Compositional Memory in Attractor Neural Networks with One-Step Learning

Gregory P. Davis[a,*], Garrett E. Katz[b], Rodolphe J. Gentili[c], James A. Reggia[a]

[a]*Department of Computer Science, University of Maryland, College Park, MD, USA*
[b]*Department of Elec. Engr. and Comp. Sci., Syracuse University, Syracuse, NY, USA*
[c]*Department of Kinesiology, University of Maryland, College Park, MD, USA*

Abstract

Compositionality refers to the ability of an intelligent system to construct models out of reusable parts. This is critical for the productivity and generalization of human reasoning, and is considered a necessary ingredient for human-level artificial intelligence. While traditional symbolic methods have proven effective for modeling compositionality, artificial neural networks struggle to learn systematic rules for encoding generalizable structured models. We suggest that this is due in part to short-term memory that is based on persistent maintenance of activity patterns without fast weight changes. We present a recurrent neural network that encodes structured representations as systems of contextually-gated dynamical attractors called attractor graphs. This network implements a functionally compositional working memory that is manipulated using top-down gating and fast local learning. We evaluate this approach with empirical experiments on storage and retrieval of graph-based data structures, as well as an automated hierarchical planning task. Our results demonstrate that compositional structures can be stored in and retrieved from neural working memory without persistent maintenance of multiple activity patterns. Further, memory capacity is improved by the use of a fast store-erase learning rule that permits controlled erasure and mutation of previously learned associations. We conclude that the combination of top-down gating and fast associative learning provides recurrent neural networks with a robust functional mechanism for compositional working memory.

Keywords: Compositionality, Working memory, Itinerant attractor dynamics, Multiplicative gating, One-step learning, Programmable neural networks

1. Introduction

Compositionality refers to the ability of an intelligent system to construct representations out of reusable parts. The principle of compositionality states that "the meaning of a complex expression is determined by its structure and the meanings of its constituents" (Fodor and Pylyshyn, 1988; Nefdt, 2020; Szabó, 2012). Despite widespread debate over the precise definition and interpretation of compositionality, there is substantial evidence that structured representation plays a critical role in human reasoning. Compositional representations are useful because they can be systematically generalized and reorganized to facilitate rapid comprehension in novel circumstances. For example, a person who knows how to prepare tea can learn to prepare coffee with minimal difficulty by decomposing the process and recognizing familiar behaviors (e.g., boiling water). Compositional reasoning is considered crucial in domains as diverse as language comprehension (Baayen, 1994; Pelletier, 1994; Aizawa, 2003), behavioral planning and imitation (Baayen, 1994; Reverberi et al., 2012; Botvinick, 2008), visual perception (Riesenhuber and Poggio, 1999; Witkin and Tenenbaum, 1983; Bienenstock et al., 1997), and concept learning (Kamp and Partee, 1995; Barsalou, 1993; Piantadosi et al., 2016).

*Corresponding author.
Email addresses: grpdavis@umd.edu (Gregory P. Davis), gkatz01@syr.edu (Garrett E. Katz), rodolphe@umd.edu (Rodolphe J. Gentili), reggia@umd.edu (James A. Reggia)

Published journal article available at https://doi.org/10.1016/j.neunet.2021.01.031

Compositionality is readily achieved in cognitive systems capable of symbolic manipulation, but is much more challenging for sub-symbolic systems such as artificial neural networks. This has fueled a long-standing controversy over whether neural networks can represent compositional structures (Fodor and Pylyshyn, 1988; Marcus, 2018; Lake et al., 2017). Recent advances in neural machine translation and sequence-to-sequence modeling have demonstrated remarkable progress toward compositional learning in neural systems. This is due to several innovations that improve short term memory in neural networks, including recurrent processing units (Hochreiter and Schmidhuber, 1997; Chung et al., 2014), attention mechanisms (Vaswani et al., 2017; Hupkes et al., 2018), and external memory resources (Graves et al., 2014; Sukhbaatar et al., 2015; Pham et al., 2018). These techniques are often combined because they provide complementary benefits, allowing neural networks to maintain activation states over time and model dependencies between distant representations (Bahdanau et al., 2014; Lake, 2019).

Despite this progress, and although there are disagreements about how compositionality should be evaluated in neural models (Hupkes et al., 2020; Nefdt, 2020), empirical studies demonstrate that state-of-the-art deep neural networks struggle to learn systematic rules that permit generalization beyond training data (Lake and Baroni, 2018; Loula et al., 2018; Hupkes et al., 2020). This is in stark contrast to the ease with which such rules can be implemented in symbolic programs. In our view, this discrepancy is due to a lack of *compositional working memory* in neural networks and an inability to encode structured representations. Working memory is a form of short term memory containing information that is actively manipulated by cognitive processes (Baars, 2007; Oberauer, 2009). Although its capacity is limited, working memory is capable of storing structured or "chunked" representations that provide access to a broad range of information (Cowan, 2010; Campitelli et al., 2007).

One way to provide neural networks with compositional working memory is to integrate them into hybrid systems with non-neural symbolic algorithms that manipulate compositional data structures (Besold et al., 2015; Marcus, 2020; Kipf et al., 2019; Burke et al., 2019; Andreas et al., 2016). For example, some neural-guided search algorithms maintain compositional data structures in non-neural symbolic memory and use neural processing to inform their construction (Bunel et al., 2018; Silver et al., 2016; Kalyan et al., 2018). From an engineering standpoint, this is a reasonable and effective approach, but it does not address how compositional structures can be encoded in purely neural models. An alternative approach is to develop purely neural computers with general-purpose memory arrays that incorporate key aspects of symbolic computation (Graves et al., 2016, 2014). While these systems have an impressive ability to learn algorithmic procedures from data, they require unconstrained access to large sets of activation patterns maintained in external memory, which is considered highly implausible from a biological perspective.

Many purely neural models of working memory employ spatially localized representations, such as localized attractors of Dynamic Field Theory (Sandamirskaya et al., 2013; Erlhagen and Schöner, 2002), and minimally overlapping cell assemblies in neural blackboard architectures (Van der Velde and Kamps, 2006). Localist representations introduce an undesirable correspondence between memory and architecture that limits representational flexibility, and often requires task-specific circuitry. In contrast, vector-space approaches employ fully distributed representations that can be composed using superposition and binding operations (Gayler, 2003; Plate, 1995). For example, the Semantic Pointer Architecture uses symbol-like vector representations that can be recursively composed to store structured information (Blouw et al., 2016; Eliasmith et al., 2012). Notably, operations for composing semantic pointers can be learned using biologically-plausible learning rules in spiking neural networks (Stewart et al., 2011). However, a significant disadvantage to this approach is that the encodings of structured representations are semantically related to their constituent elements. This means that additions to a compositional data structure involve changes to the semantic pointer that encodes it, as well as any semantic pointers for super-structures that it is contained in. For example, adding a leaf node to a tree would require reconstruction of the semantic pointers for each ancestor of the new node. This makes semantic pointer encodings of data structures effectively immutable, as modifications involve constructing new encodings.

Performance on working memory tasks is correlated with measures of general intelligence (Conway et al., 2003; Colom et al., 2004; Jaeggi et al., 2008), and working memory operations are considered to be consciously reportable (Baars and Franklin, 2003; Baddeley, 1993). For these reasons, biologically-plausible models of compositional working memory may lead to significant advances in AI systems and contribute to a deeper understanding of consciousness and cognition (Reggia et al., 2020, 2019), including cognitive-motor control (Hauge et al., 2020, 2019). Substantial evidence indicates that activity-silent mechanisms such as rapid synaptic plasticity play a critical role in working memory (Manohar et al., 2019; Mongillo et al., 2008; Rose et al., 2016; Stokes, 2015; Barbosa et al., 2019). This suggests that persistent activity maintenance alone is not sufficient for modeling the complexities of human working

memory, and may explain in part why compositional learning is difficult for artificial neural networks.

Contemporary neural networks typically undergo a training phase during which weights are updated via iterative gradient descent and fixed during task performance. Recent exceptions to this demonstrate that fast associative learning greatly improves short-term memory in neural networks because it permits storage without active maintenance (Ba et al., 2016; Danihelka et al., 2016; Miconi et al., 2018). Furthermore, models based entirely on fast associative learning and itinerant dynamical attractors can learn to perform complex working memory tasks (Sylvester and Reggia, 2016; Sylvester et al., 2013), and can simulate Turing machines without the need for consistent maintenance of memory activation (Katz et al., 2019). However, such models have focused on storage of individual memories and temporal sequences of memories. To our knowledge, fast associative learning has not yet been applied to explicit encoding of hierarchical compositional structures (e.g., trees) in working memory as we do here.

In this paper, we introduce a neural model of compositional memory based on context-dependent itinerant attractors in recurrent neural networks. We refer to such models as *attractor graph networks* or AGNs. Attractor graphs are composed of fixed-point dynamical attractors (vertices) and transitions between attractors (edges) that are learned using a combination of multiplicative gating and one-step associative learning. The use of multiplicative gating is motivated by evidence that it contributes to functional grouping in neural populations (Baan et al., 2019; Vecoven et al., 2020; Masse et al., 2018; Chabuk and Reggia, 2013; Rikhye et al., 2018). In an AGN, contextual gating signals select learned associations to govern the dynamics of the network over time, directing a traversal through the attractor graph that is analogous to the operation of a finite state machine. We show how compositional data structures such as associative arrays (i.e., dictionaries or maps), linked lists, and trees can be encoded in AGNs and retrieved by programmatic procedures that control sequential iteration through attractor graphs. In contrast to semantic pointers and other vector space representations, structured representations in AGNs are made up of learned associations between activity states, and can therefore be freely modified by changing synaptic weights without changing activity states.

AGNs are inspired by neurobiological studies of working memory and are based on several biologically-inspired principles. Representations in attractor graphs are composed of distributed activation patterns (dynamical attractors) that are supported by learned connection weights, and can be reactivated as needed without persistent activity maintenance. This means that the capacity of working memory is not directly limited by network architecture, and is instead determined by the organization of both the underlying attractor model and the control signals used for memory retrieval. We propose that the phenomenon of "chunking" is supported by compositional structuring, which organizes the contents of working memory according to systematic procedures for top-down control.

2. Methods

In this section we describe our method for representing compositional data structures as systems of dynamical attractors in recurrent neural networks. We illustrate this with a small multi-region model shown in Figure 1. The core memory region (*mem*) is a recurrent neural network with attractor states that represent general-purpose elements in short-term working memory. We refer to these elements as *memory states*. Memory states are linked together with context-dependent transitions that are controlled by inputs from the context region (*ctx*). Patterns of activity in the lexicon region (*lex*) represent symbolic tokens that can be "stored" in memory states and used as names for variable pointers to memory states.

Compositional data structures are encoded in the *mem* region as systems of itinerant dynamical attractors called *attractor graphs*. Vertices in these graphs are fixed-point dynamical attractors, and edges are context-dependent transitions between attractor states. We refer to recurrent networks with attractor graph dynamics as *attractor graph networks*, or AGNs. Section 2.1 describes the structure and dynamics of AGNs in detail. Section 2.2 shows how compositional data structures can be represented as attractor graphs, and describes the interactions between the *mem*, *ctx*, and *lex* regions of the model. Section 2.3 shows how the model shown in Figure 1 can be embedded into a larger programmable neural network that constructs and manipulates data structures according to learned programmatic procedures.

2.1. Attractor Graph Networks

Attractor graphs are systems of itinerant dynamical attractors with context-dependent transitions. Attractor itinerancy refers to a temporal process in which a dynamical system undergoes transitions through a sequence of attractor

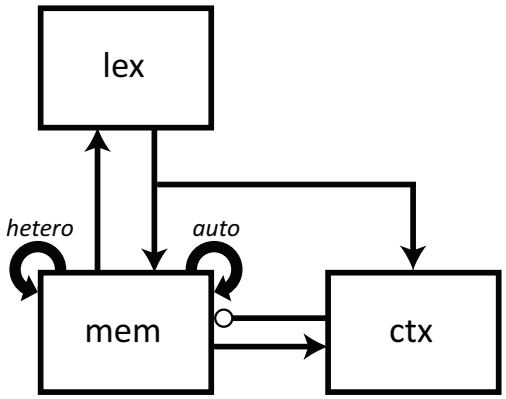

Figure 1: Neural model with compositional memory. Each box is a neural region and solid lines indicate connectivity between regions. Each connection is controlled by a binary gate (not shown) that determines whether it contributes to neural dynamics during each timestep. The memory region (*mem*, bottom left) is an attractor graph network with dense hetero-associative and auto-associative connectivity (bold looped arrows). Patterns of activity in *mem* represent general-purpose memory states that "store" symbolic tokens represented by distributed activity patterns in the lexicon region (*lex*, top left). Associations between memory states and symbolic tokens are learned in the pathway from *mem* to *lex*. Transitions between attractors in *mem* are contextualized by activity in the context region (*ctx*, bottom right) via a pathway from *ctx* to *mem*. The open circle at the end of this pathway indicates that it provides contextual gating inputs to *mem*. Activity patterns in *ctx* may be derived from *mem* or *lex* patterns, and serve as labels for relations between memory states that are used to construct compositional data structures. Finally, *lex* patterns can represent variable names that point to memory states via the pathway from *lex* to *mem*.

states, settling momentarily at each state before rapidly transitioning to its successor (Miller, 2016; Hoshino et al., 1997). In an AGN, each attractor state (vertex of graph) may have multiple outgoing transitions (edges of graph) with unique successors, and activity evolves according to contextual inputs (edge labels) that select which learned transition governs dynamics at each timestep. In the model shown in Figure 1, the *mem* region is an AGN that receives contextual inputs from the *ctx* region.

The branched organization of sequential attractors in AGNs makes it possible to represent a broad range of graph-based data structures, including compositional data structures such as linked lists, associative arrays (dictionaries, maps), and trees. Because attractor transitions depend on contextual inputs, data structures represented in the network's memory can be accessed via top-down control of contextual inputs over time (as shown in Section 2.2.1). This makes AGNs effective general-purpose models of compositional working memory that may be integrated into larger neural systems, such as the one shown in Figure 5.

Let r denote a particular region in a neural architecture, such as *mem* or *ctx* in Figure 1. A region r is used as an AGN by equipping it with two types of recurrent connectivity that play distinct roles in attractor itinerancy. A dense auto-associative weight matrix A_r encodes patterns of activation as fixed-point attractor states that represent retrievable memories, as in a Hopfield network. Each learned attractor resides in a unique orthant of activation space. Transitions between attractor states are encoded in a dense hetero-associative weight matrix H_r. Errors introduced by transition dynamics are corrected by auto-associative dynamics, which bring activity into the target orthant. Errors in attractor convergence are corrected by self-connectivity that saturates activation within the current orthant. Thus, as long as the transition brings activity into the correct orthant, the target pattern can be perfectly recalled.

The dynamics of the model are controlled by a set of binary gates that determine which connections are active at each timestep. For example, when gate $g_r^A = 1$, activity evolves according to the auto-associative matrix A_r, which causes convergence to the nearest attractor state. A temporal sequence of connection gate values implements the multi-step procedure of attractor itinerancy described above (i.e., hetero-associative transition, auto-associative convergence, saturation). These gate values may be provided by a dedicated region of binary threshold neurons, such as a controller in a programmable neural network (Sylvester and Reggia, 2016; Katz et al., 2019), as discussed in Section 2.3.

In addition to recurrent connectivity, AGNs receive two types of extrinsic inputs: additive and multiplicative. Additive inputs $\mathbf{I}_r$ contribute to the summation of synaptic inputs and can be used to initialize the activity state of the network. In contrast, multiplicative inputs $\mathbf{c}$ are binary signals that enable or disable specific neurons by gating

cumulative synaptic activity. These signals enable functional branching in attractor transitions by modulating hetero-associative dynamics, as described below, and are distinct from connection gates (e.g., g_r^A).

The following equations describe the dynamics of an AGN region such as *mem* over time. In Section 2.2, we return to the neural circuit shown in Figure 1 to explain how these dynamics contribute to compositional working memory. Here we use subscript r to refer to a generic region. First, synaptic input is aggregated from gated recurrent connectivity and external inputs:

$$\mathbf{s}_r(t) = \underbrace{g_r^S(t)\ \omega_r\ \mathbf{v}_r(t)}_{\text{saturation}} + \underbrace{g_r^A(t)\ A_r\ \mathbf{v}_r(t)}_{\text{convergence}} + \underbrace{g_r^H(t)\ H_r\ \mathbf{v}_r(t)}_{\text{transition}} + \mathbf{I}_r(t) \tag{1}$$

where

- $\mathbf{s}_r(t)$ is a vector of cumulative synaptic input to region r at time t.
- $\mathbf{v}_r(t)$ is a vector of neural activity of region r at time t.
- ω_r is a scalar self-weight that causes saturation and maintenance of neural activity in region r when $g_r^S(t) = 1$.
- A_r is an auto-associative weight matrix for region r that causes convergence to a nearby fixed-point attractor when $g_r^A(t) = 1$.
- H_r is a hetero-associative weight matrix for region r that causes a transition between attractor states when $g_r^H(t) = 1$.
- $\mathbf{I}_r(t)$ is a vector of external (non-recurrent) synaptic input to region r at time t. This input may be provided by other neural regions, as described in Section 2.2, or from outside the model for initialization purposes, as shown in Section 2.1.1.

Next, multiplicative activation is determined based on gated contextual input:

$$\mathbf{x}_r(t) = \begin{cases} \mathbf{c}_r(t), & \text{if } g_r^C(t) = 1 \\ \mathbf{1}, & \text{otherwise} \end{cases} \tag{2}$$

where

- $\mathbf{c}_r(t)$ is a vector containing the net multiplicative input to region r at time t. This input may be provided by other neural regions such as *ctx* in Figure 1.
- $\mathbf{1}$ is a vector of ones of the same size as $\mathbf{c}_r(t)$.
- $\mathbf{x}_r(t)$ is a vector of active multiplicative input to region r at time t. When $g_r^C(t)$ is enabled, $\mathbf{x}_r(t) = \mathbf{c}_r(t)$. Otherwise, $\mathbf{x}_r(t)$ defaults to $\mathbf{1}$, and synaptic input $\mathbf{s}_r(t)$ passes into the activation function regardless of $\mathbf{c}_r(t)$.

Finally, neural activation is computed for the next timestep by combining synaptic and multiplicative input:

$$\mathbf{v}_r(t+1) = \sigma_r\Big(\mathbf{x}_r(t) \odot \mathbf{s}_r(t)\Big) \tag{3}$$

where

- σ_r is a sign-preserving neural activation function in region r (i.e., $sgn(\sigma_r(x)) = sgn(x)$). For the experiments reported in Section 3, we use the hyperbolic tangent activation function for the *mem* AGN shown in Figure 1.
- $\odot$ is the Hadamard (element-wise) product.

An attractor transition is carried out in four stages, starting with an activity pattern that may be initialized by $\mathbf{I}_r(t)$, or by a previous transition. 1) The transition begins with application of a context pattern $\mathbf{c}$, which disables a subset of neurons and "masks" the source activation pattern. This is done by enabling saturation and context gates ($g_r^C(t)$ and $g_r^S(t)$) for one timestep. 2) Next, while context gate $g_r^C(t)$ remains enabled, hetero-associative gate $g_r^H(t)$ is enabled for one timestep, and activity transitions to a new pattern of non-zero activation in the participating neurons. 3) Once the initial transition is complete, $g_r^A(t)$ is enabled, causing auto-associative dynamics across the entire network. Over several timesteps, activity converges to the nearest fixed-point attractor. Note that because context gate $g_r^C(t)$ is disabled, all neurons participate in auto-associative dynamics. 4) Finally, saturation gate $g_r^S(t)$ is enabled, causing activity to saturate within the current orthant of activity space over several timesteps. This makes up for any errors in convergence, and is successful as long as auto-associative dynamics resulted in an activity pattern in the correct orthant. Note that saturation dynamics are only necessary for continuous models (e.g., when σ_r is the hyperbolic tangent), and can be omitted in models with a threshold activation function (e.g., the sign/signum function). The connection gate values for each stage of an attractor transition are shown in Table 1. Stages 1 and 2 take one timestep each, while Stages 3 and 4 each occur over several timesteps.

Table 1: Connection gate values for each stage of an AGN attractor transition

Stage	g_r^S	g_r^C	g_r^H	g_r^A
1. Mask	1	1		
2. Transition		1	1	
3. Converge				1
4. Saturate	1			

Each attractor state learned by the AGN may have multiple hetero-associative transitions to several other attractor states. Each of the transitions from a given state must be learned in the context of a unique multiplicative input pattern $\mathbf{c}_r(t)$. During attractor transition dynamics, the choice of $\mathbf{c}_r(t)$ determines which learned association will govern the transition. This process is depicted in Figure 2, which shows the activation space for an AGN with three neurons. Trajectories are shown for two transitions from an attractor state $\mathbf{m}^{(0)}$ using unique context patterns $\mathbf{c}^{(1)}$ and $\mathbf{c}^{(2)}$.

Learned attractor states are patterns in $\{-\rho_r, +\rho_r\}^{N_r}$, where N_r is the number of neurons in the AGN, and ρ_r is a parameter that determines the magnitude of learned activity states. These activation patterns are randomly generated using a Bernoulli process with equal probabilities (i.e., a fair coin toss). For models using a threshold activation function, activation patterns are discrete and bipolar, and $\rho_r = 1$. When the hyperbolic tangent activation function is used, $0 << \rho_r < 1$ (typically 0.9999) because $\sigma_r^{-1}(1) = \infty$. Once ρ_r is set, the value of the saturation self-weight ω_r is determined according to the following relation:

$$\rho_r = \sigma_r(\omega_r \rho_r)$$

This ensures that $\pm\rho_r$ is a stable fixed-point when saturation dynamics are enabled ($g_r^S(t) = 1$).

Multiplicative gating patterns $\mathbf{c}_r(t)$ that contextualize transitions are in $\{0, 1\}^N$, where N is the number of neurons in the network. The density of these patterns is an important parameter that determines how many neurons participate in each transition. This parameter is referred to as λ, the probability used in the Bernoulli process that generates context patterns.

The auto-associative weight matrix for a region r (A_r) is learned using traditional Hebbian learning, starting with an initial matrix with zero entries, and updating the weights for each learned pattern:

$$\Delta A_r = \frac{1}{\rho_r^2 N_r} \sigma_r^{-1}(\mathbf{v}) \mathbf{v}^\top \tag{4}$$

where $\mathbf{v}$ is the pattern to be learned, σ_r is the neural activation function, ρ_r is the stable activation level of learned attractor patterns, and N_r is the number of neurons in the region. When the sign/signum activation function is used ($\sigma_r = sgn(x)$), its inverse is defined as:

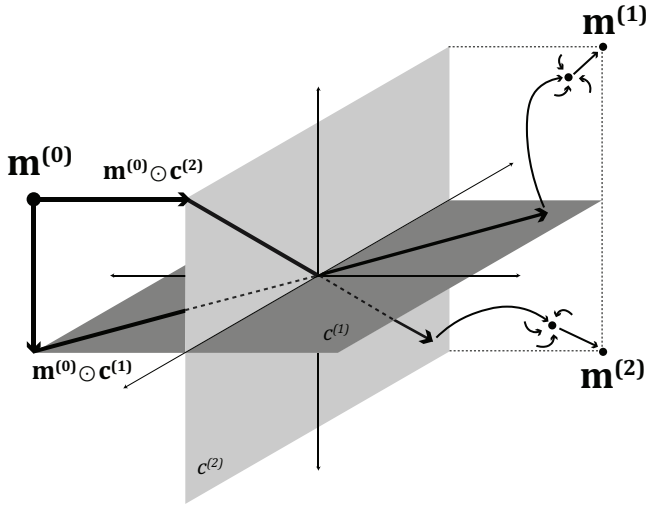

Figure 2: Visual depiction of context-dependent attractor transitions in the activation space of an AGN with three neurons. Each axis represents the activation level of one neuron (from -1 to +1). Two transitions from state $\mathbf{m}^{(0)}$ (left side) are shown, each with a distinct context pattern ($\mathbf{c}^{(1)}$ or $\mathbf{c}^{(2)}$). Each context pattern is a binary vector that selects subsets of neurons to participate in hetero-associative dynamics, and corresponds to a subspace in activation space (shaded planes labeled $c^{(1)}$ and $c^{(2)}$). Transitions are executed over several steps. 1) First, a context pattern is used to "mask" the source activation pattern, disabling a subset of neurons and collapsing activity into the context-specific subspace (arrows leaving $\mathbf{m}^{(0)}$ to $\mathbf{m}^{(0)} \odot \mathbf{c}^{(1)}$ or $\mathbf{m}^{(0)} \odot \mathbf{c}^{(2)}$, left side). 2) Next, while the context masking remains active, hetero-associative dynamics cause activation to transition into a new orthant of the subspace (center arrows within shaded planes). 3) Then, the context masking is disabled, and auto-associative dynamics cause convergence to the nearest fixed-point attractor state over several timesteps (curved arrows, right side). 4) To correct any errors in convergence, saturation dynamics push activity to the corner of the current orthant of activation space over several timesteps (straight arrows to $\mathbf{m}^{(1)}$ and $\mathbf{m}^{(2)}$, right side).

$$sgn^{-1}(x) = \begin{cases} +1, & \text{if } x > 0 \\ -1, & \text{if } x < 0 \\ 0, & \text{if } x = 0 \end{cases}$$

The hetero-associative weight matrix for a region r (H_r) is also learned using local one-step learning. However, because transitions are context-dependent, they must be learned with the corresponding contextual gating pattern. Each transition is learned with the following update rule, starting with a zero matrix:

$$\Delta H_r = \frac{1}{\lambda \rho_r^2 N_r} \sigma_r^{-1}(\mathbf{c} \odot \mathbf{v})(\mathbf{c} \odot \mathbf{u})^\top \tag{5}$$

where $\mathbf{u}$ is the source pattern, $\mathbf{v}$ is the target pattern, $\mathbf{c}$ is the context pattern, λ is the density of contextual gating patterns, and σ_r, ρ_r, and N_r are as defined above. Note that the normalization factor $\lambda \rho_r^2 N$ is approximately equivalent to the squared Euclidean length of $\mathbf{c} \odot \mathbf{u}$.

Other associative learning rules can be used to learn A_r and H_r. To adapt a learning rule for gated hetero-associative learning, it must 1) only update weights connecting neurons that participate in the transition, as determined by contextual gating pattern $\mathbf{c}$, and 2) renormalize weight updates according to λ, the density of $\mathbf{c}$. In this paper, we consider both traditional Hebbian learning (Equations 4 and 5) and the fast store-erase learning rule (Katz et al., 2019):

$$\Delta A_r = \frac{1}{\rho_r^2 N_r} \Big(\sigma_r^{-1}(\mathbf{v}) - \big(A_r \mathbf{v}\big) \Big) \mathbf{v}^\top \tag{6}$$

$$\Delta H_r = \frac{1}{\lambda \rho_r^2 N_r} \left(\underbrace{\sigma_r^{-1}(\mathbf{c} \odot \mathbf{v}) - (\mathbf{c} \odot H_r (\mathbf{c} \odot \mathbf{u}))}_{\text{masked target delta}} \right) \underbrace{\left(\mathbf{c} \odot \mathbf{u}^\top \right)}_{\text{source}} \tag{7}$$

where $\mathbf{c}$ is a binary contextual gating pattern, $\mathbf{u}$ is the initial source activity pattern, $\mathbf{v}$ is the final target activity pattern, and λ, σ_r, ρ_r, and N_r are as defined above. The additional context masking in the target delta of Equation 7 (just before H_r) ensures that weights with deactivated post-synaptic neurons are not updated.

The store-erase learning rule contains an anti-Hebbian component that erases previously stored associations. This makes it particularly advantageous for neural working memory, as it can be used to overwrite relations between memories and modify learned data structures. However, this learning rule has not been evaluated with auto-associative memory, and has not been systematically compared with traditional Hebbian learning. This is addressed in Section 3, where we present empirical results that compare memory storage and retrieval in AGNs learned with either traditional Hebbian or store-erase learning.

2.1.1. Contextual Gating Supports Functional Branching

Contextual gating signals in AGNs make it possible to store multiple transitions from an attractor state using a single hetero-associative weight matrix. We refer to this branching of attractor transitions as *functional branching* because it depends on functional multiplicative inputs. This is in contrast to *structural branching*, in which each branch in the attractor sequence is stored in a distinct hetero-associative weight matrix. Structural branching imposes an undesirable correspondence between network architecture and memory because the number of outgoing transitions from any given attractor state (i.e., its out-degree or branching factor) is limited by the number of hetero-associative weight matrices. In contrast, functional branching requires only a single hetero-associative weight matrix, and does not impose constraints on the structure of learned associations. This novel aspect of AGNs makes it possible to represent arbitrary directed graphs as systems of functionally-branched itinerant attractor sequences.

To clarify how AGNs work, next we illustrate how contextual gating signals support functional branching with a simple toy example. For this example, we consider an AGN with $N = 4$ neurons that use the sign/signum activation function ($\rho = 1$ and $\omega = 1$). Learned attractor states are bipolar patterns in $\{-1, +1\}^4$. The network, shown in Figure 3a, learns two attractor transitions (Figure 3b) that make up a simple attractor graph (Figure 3c). The learning procedure and transition dynamics are described in detail below, with the subscript r omitted for ease of presentation.

Three activity patterns are learned as attractors in auto-associative matrix A_r using Equation 4: $\mathbf{v}^{(0)} = [+1, -1, +1, -1]^\top$, $\mathbf{v}^{(1)} = [+1, +1, -1, -1]^\top$, and $\mathbf{v}^{(2)} = [-1, +1, -1, +1]^\top$. For simplicity, we omit the normalization term $\frac{1}{\rho^2 N}$ from the learning rule.

$$A = \sum_i \mathbf{v}^{(i)} \mathbf{v}^{(i)\top} = \mathbf{v}^{(0)} \mathbf{v}^{(0)\top} + \mathbf{v}^{(1)} \mathbf{v}^{(1)\top} + \mathbf{v}^{(2)} \mathbf{v}^{(2)\top} = \begin{bmatrix} +3 & -1 & +1 & -3 \\ -1 & +3 & -3 & +1 \\ +1 & -3 & +3 & -1 \\ -3 & +1 & -1 & +3 \end{bmatrix}$$

Note that these patterns were chosen for illustration purposes, and that AGNs are not restricted to learning orthogonal/complementary activation patterns.

Two attractor transitions are learned in hetero-associative matrix H using Equation 5: $\mathbf{v}^{(0)}$ to $\mathbf{v}^{(1)}$, and $\mathbf{v}^{(0)}$ to $\mathbf{v}^{(2)}$. Each of these transitions is contextualized by a distinct context pattern that selects a subset of neurons in the network. The first transition is contextualized by $\mathbf{c}^{(1)} = [1, 1, 0, 0]^\top$, which enables the first and second neurons, while the second transition is contextualized by $\mathbf{c}^{(2)} = [0, 0, 1, 1]^\top$, which enables the third and fourth neurons. As above, we omit the renormalization term $\frac{1}{\lambda \rho^2 N}$ for simplicity.

$$H = (\mathbf{c}^{(1)} \odot \mathbf{v}^{(1)})(\mathbf{c}^{(1)} \odot \mathbf{v}^{(0)})^\top + (\mathbf{c}^{(2)} \odot \mathbf{v}^{(2)})(\mathbf{c}^{(2)} \odot \mathbf{v}^{(0)})^\top = \begin{bmatrix} +1 & +1 & 0 & 0 \\ -1 & -1 & 0 & 0 \\ 0 & 0 & -1 & +1 \\ 0 & 0 & +1 & -1 \end{bmatrix}$$

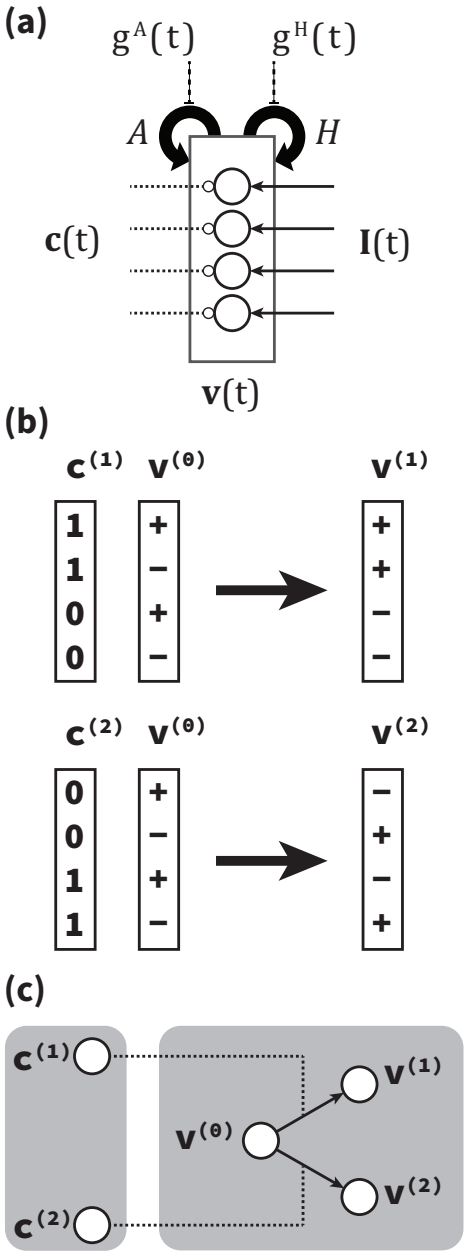

Figure 3: Toy example of transitions in an AGN (see text for details). (a) AGN with four neurons (center circles). Bold looped arrows indicate auto-associative (A) and hetero-associative (H) recurrent connectivity matrices. Dashed lines on the left indicate multiplicative gating signals ($\mathbf{c}(t)$) that contextualize attractor transitions. Solid lines on the right indicate external synaptic input ($\mathbf{I}(t)$) that initializes activation patterns ($\mathbf{v}(t)$). Binary gates ($g^A(t)$ and $g^H(t)$, top) determine which connections are active at each timestep ($g^S(t)$ and $g^C(t)$ not shown). (b) Two contextual transitions learned in the network. Each transition begins with activity pattern $\mathbf{v}^{(0)}$ and transitions to a unique successor ($\mathbf{v}^{(1)}$ or $\mathbf{v}^{(2)}$) in the context of a unique context pattern ($\mathbf{c}^{(1)}$ or $\mathbf{c}^{(2)}$). (c) Graphical depiction of the learned attractor graph. Gating patterns (circles in left rectangle) contextualize transitions between distributed activity states in the AGN (circles in right rectangle).

The resulting weight matrix H encodes two transitions from $\mathbf{v}^{(0)}$ that are segregated to different sub-populations of neurons and weights. The disjoint context patterns used in this example are solely for illustrative purposes; in general, context patterns can be randomly generated and can enable overlapping sets of neurons, introducing interference

between weight updates for different context patterns. Empirical results presented in Section 3 demonstrate successful transitions despite this interference.

When a transition is executed, the corresponding context pattern ($\mathbf{c}^{(1)}$ or $\mathbf{c}^{(2)}$) is presented to select the subset of neurons that learned the transition. Consequently, only one quadrant of H governs the first step of transition dynamics, updating activity in the context-specific subset of neurons associated with the transition. To complete the transition, auto-associative dynamics cause convergence to the nearest attractor, completing the target activity pattern. This process is illustrated in Table 2 for the transition from $\mathbf{v}^{(0)}$ to $\mathbf{v}^{(1)}$ using context signal $\mathbf{c}^{(1)}$ (blank cells indicate zero values).

Table 2: Timing of example attractor transition dynamics

Timestep	$g^S(t)$	$g^C(t)$	$g^H(t)$	$g^A(t)$	$\mathbf{I}(t)$	$\mathbf{c}(t)$	$\mathbf{x}(t)$	$\mathbf{s}(t)$	$\mathbf{v}(t)$
Initialization (t=0)					$\mathbf{v}^{(0)}$		$\mathbf{1}$	$\mathbf{v}^{(0)}$	$\sigma_r(\mathbf{v}^{(0)}) = \begin{bmatrix}+1\\-1\\+1\\-1\end{bmatrix}$
Masking (t=1)	1	1				$\mathbf{c}^{(1)}$	$\mathbf{c}^{(1)}$	$\mathbf{v}(0)$	$\sigma_r(\mathbf{c}^{(1)} \odot \mathbf{v}(0)) = \begin{bmatrix}+1\\-1\\0\\0\end{bmatrix}$
Transition (t=2)		1	1			$\mathbf{c}^{(1)}$	$\mathbf{c}^{(1)}$	$H\mathbf{v}(1)$	$\sigma_r(\mathbf{c}^{(1)} \odot H\mathbf{v}(1)) = \begin{bmatrix}+1\\+1\\0\\0\end{bmatrix}$
Convergence (t=3)				1			$\mathbf{1}$	$A\mathbf{v}(2)$	$\sigma_r(A\mathbf{v}(2)) = \begin{bmatrix}+1\\+1\\-1\\-1\end{bmatrix}$

This procedure is carried out in an AGN using a temporal sequence of inputs and connection gate values shown at the top of each timestep block (e.g., $\mathbf{I}(0) = \mathbf{v}^{(0)}$). The second transition from $\mathbf{v}^{(0)}$ to $\mathbf{v}^{(2)}$ can be executed by substituting context pattern $\mathbf{c}^{(2)}$ for $\mathbf{c}^{(1)}$. Note that saturation dynamics are omitted after convergence because the sign activation function has discrete outputs. The saturation connection is, however, necessary for activity maintenance during timestep 1. Although this simple example only requires one timestep of auto-associative dynamics to recover the final pattern, in general, several timesteps will be necessary.

2.2. Compositional Memory

The AGN model outlined in the previous section describes the dynamics of the *mem* region of the model shown in Figure 1. In this section, we describe the interactions between the *mem*, *ctx*, and *lex* regions of this model, and explain how they encode compositional data structures.

Attractor states in the *mem* region represent discrete generic memory states, and are organized into structured representations with context-dependent attractor transitions. For example, a linked list is represented by a sequence of memory states connected by transitions with a shared list-specific context. Each memory state may be associated with a pattern of activity in the lexicon region (*lex*) via the pathway from *mem* to *lex*. Patterns in *lex* represent symbolic tokens that are "stored" in memory states using associative learning. These tokens may represent words, for example, and a sentence could be represented as a linked list of memory states, each associated with the *lex* pattern representing the corresponding word in the sentence.

Transitions between memory states are contextualized by activity patterns in the context region (*ctx*). The *ctx* region uses the heaviside activation function, and provides the binary context signals that select subsets of *mem* neurons to participate in attractor transitions ($\mathbf{c}(t)$ in Equation 2). Patterns in *ctx* may be derived from *mem* and *lex* patterns via pathways from those regions to *ctx*. This makes it possible for memory states or symbolic tokens to indirectly contextualize memory transitions through an intermediate associated *ctx* activity pattern.

While the *mem* region is an AGN with multiple recurrent pathways and attractor dynamics, the *ctx* and *lex* regions have no recurrent connectivity, and only include internal saturation dynamics that maintain activity patterns over time. With the exception of the pathway from *ctx* to *mem*, which provides unweighted contextual gating signals for attractor transitions in *mem*, inter-regional pathways (solid lines with arrows in Figure 1) have dense connectivity matrices that are learned using one-step associative learning.

Activity in the *mem* region evolves according to Equations 1 - 3. External multiplicative input to the *mem* region $\mathbf{c}_{mem}(t)$ comes from neural activation in the *ctx* region:

$$\mathbf{c}_{mem}(t) = \mathbf{v}_{ctx}(t)$$

where $\mathbf{v}_{ctx}(t)$ is the neural activation of the *ctx* region at time t. When the context gate for *mem* region dynamics is enabled ($g^C_{mem} = 1$), activity in *mem* is contextualized by activity in *ctx*. All other inputs to *mem* from other regions are absorbed into the $\mathbf{I}_{mem}(t)$ term:

$$\mathbf{I}_{mem}(t) = \sum_q g_{mem,q}(t) W_{mem,q}(t) \mathbf{v}_q(t)$$

where $\mathbf{v}_q(t)$ is the neural activation of region q at time t, $W_{mem,q}(t)$ is the connectivity matrix from region q to *mem* at time t, and $g_{mem,q}(t)$ is a binary connection gate on the pathway.

Activity in the *ctx* and *lex* regions evolves according to the following equations, which are simplified versions of Equations 1 and 3 that do not include contextual gating:

$$\mathbf{s}_r(t) = g^S_r \omega_r \mathbf{v}_r(t) + \mathbf{I}_r(t) + \sum_q g_{r,q}(t) W_{r,q}(t) \mathbf{v}_q(t) \tag{8}$$

$$\mathbf{v}_r(t+1) = \sigma_r\left(\mathbf{s}_r(t)\right) \tag{9}$$

where

- $\mathbf{s}_r(t)$ is a vector of cumulative synaptic input to region r at time t.
- $g^S_r(t)$ is a binary gate that enables or disables activity saturation and maintenance in region r at time t. As in Equation 1, ω_r is a scalar self-weight that establishes the fixed-point of saturated neural activity.
- $\mathbf{I}_r(t)$ is a vector of external synaptic input to region r at time t.
- $W_{r,q}(t)$ is a weight matrix for the pathway from region q to region r at time t, which is controlled by a binary connection gate $g_{r,q}(t)$.
- $\mathbf{v}_r(t)$ is a vector of neural activity in region r at time t.
- σ_r is the neural activation function for region r. The *ctx* region uses the Heaviside activation function, while the *lex* region uses the sign/signum activation function.

Inter-regional weight matrices in the model ($W_{r,q}$) are learned with one-step associative learning. As noted in Section 2.1, we consider both traditional Hebbian learning (Equation 10) and the store-erase rule (Equation 11):

$$\Delta W_{r,q} = \frac{1}{\rho_q^2 N_q} \sigma_r^{-1}(\mathbf{v}) \mathbf{u}^\top \tag{10}$$

$$\Delta W_{r,q} = \frac{1}{\rho_q^2 N_q} \left(\sigma_r^{-1}(\mathbf{v}) - W_{r,q}\mathbf{u}\right) \mathbf{u}^\top \tag{11}$$

where $\mathbf{u}$ is the source (initial) pattern in region q, $\mathbf{v}$ is the target (final) pattern in region r, $W_{r,q}$ is the associative weight matrix for the pathway from q to r, σ_r is the neural activation function of r, ρ_q is the activation magnitude of neurons in q, and N_q is the number of neurons in q. The context region (*ctx*) must use the heaviside activation

function, and learn binary patterns ($\rho = 1$), while the *lex* region may use any sign-preserving activation function such as the hyperbolic tangent or sign/signum function.

The pathway from *mem* to *lex* ($W_{lex,mem}$) learns associations between memory states and lexical symbols stored in those states. This pathway is used to "read" the contents of the currently active state in the *mem* region. The pathways from *mem* and *lex* to *ctx* ($W_{ctx,mem}$ and $W_{ctx,lex}$) learn associations with patterns that contextualize memory transitions. Each memory state and lexical symbol corresponds to a unique randomly generated context state. These associations are critical for construction of compositional data structures in attractor graphs, as described below.

2.2.1. Representing Compositional Data Structures

The graph-organized memory represented by the *mem* region attractor graph is suitable for encoding compositional data structures. Here we focus on associative arrays, linked lists, and trees, each of which can be represented by a particular organization of attractors. Several instances of these data structures may be encoded simultaneously as subgraphs of a single attractor graph. Individual elements of a compositional data structure are accessed by sequences of attractor transitions resembling iteration through data structures in conventional computer memory. Each memory state can be associated with a *lex* activity pattern that represents the symbolic token stored in that memory state.

Representation of associative arrays (dictionaries, maps) in attractor graphs is straightforward. A map is represented by a dedicated memory state, and each entry in the map is represented by a transition to a target memory state (value) that is contextualized by a *ctx* pattern (key). Keys may be derived from patterns in other regions via pathways into *ctx* (e.g., *lex* or *mem*). To access the value associated with a key, the key pattern is initialized in *ctx*, and the memory pattern representing the map is initialized in *mem*. Then the attractor transition is executed using the *ctx* pattern as context. The resulting activity pattern in *mem* represents the memory state (value) associated with the key, which may be a complex data structure or a single memory state with a corresponding symbol that is retrieved via the pathway from *mem* to *lex*.

Accessing a map with a key that has no corresponding value results in undefined behavior. When erroneous lookups are possible, the data structure must be modified to allow validation. One possibility is to learn self-transitions for each value that use the key as a context pattern. Thus, to validate a map lookup, an additional transition can be executed after the lookup, and the resulting activity state can be compared with the value state. If they do not match, the lookup was not successful.

A linked list is encoded as a trajectory through the attractor graph, as shown in Figure 4a. The trajectory begins with a *mem* pattern that represents the list object (referred to as the *head*), and includes zero or more additional *mem* patterns that represent the list elements. The end of the list is marked by a self-loop transition in the final memory state in the sequence. Although a list cannot contain repeat memory states, two states in a list may be associated with the same activity pattern in *lex* via the pathway from *mem* to *lex*, representing storage of the same symbolic token in two positions of the list. Each transition in the trajectory is contextualized by a shared *ctx* pattern that is specific to the list, and is associated with the head pattern in *mem* via the pathway from *mem* to *ctx*. To iterate through a list, the head pattern is first initialized in *mem* and used to retrieve the list-specific pattern in *ctx*. This context pattern is then used to contextualize a sequence of attractor transitions that terminates when the post-transition state is identical to the pre-transition state indicating a complete traversal of the sequence. Note that an empty list is represented by a head pattern in *mem* that transitions directly to itself (i.e., a self-loop trajectory), and therefore terminates after a single transition.

Because transitions in a linked list are contextualized by a list-specific context pattern, a memory state may be contained in several lists. In this case, the shared memory state has distinct transitions to list-specific successors. This means that memory states are *reusable components* that may be used in multiple compositional data structures. Our approach is similar to Borisyuk et al. (2013) in that a list-specific context signal resolves ambiguities in sequence recall that occur when an element is contained in more than one sequence. However, context signals in AGNs are multiplicative masks that select subsets of the neural population to participate in sequence transitions. As a result, the transitions for distinct sequences are learned in distinct but overlapping sets of connection weights.

Trees can be encoded in attractor graphs by recursive composition of either associative arrays or linked lists, each with distinct advantages. Tree nodes represented by associative arrays have direct parent-child relations that are labeled by *ctx* patterns, which must be provided during tree traversal. However, this organization permits random access of child nodes during traversal. Trees represented by linked lists, on the other hand, can be traversed without external provision of context patterns. This is because each node is associated with a unique *ctx* pattern that contextualizes

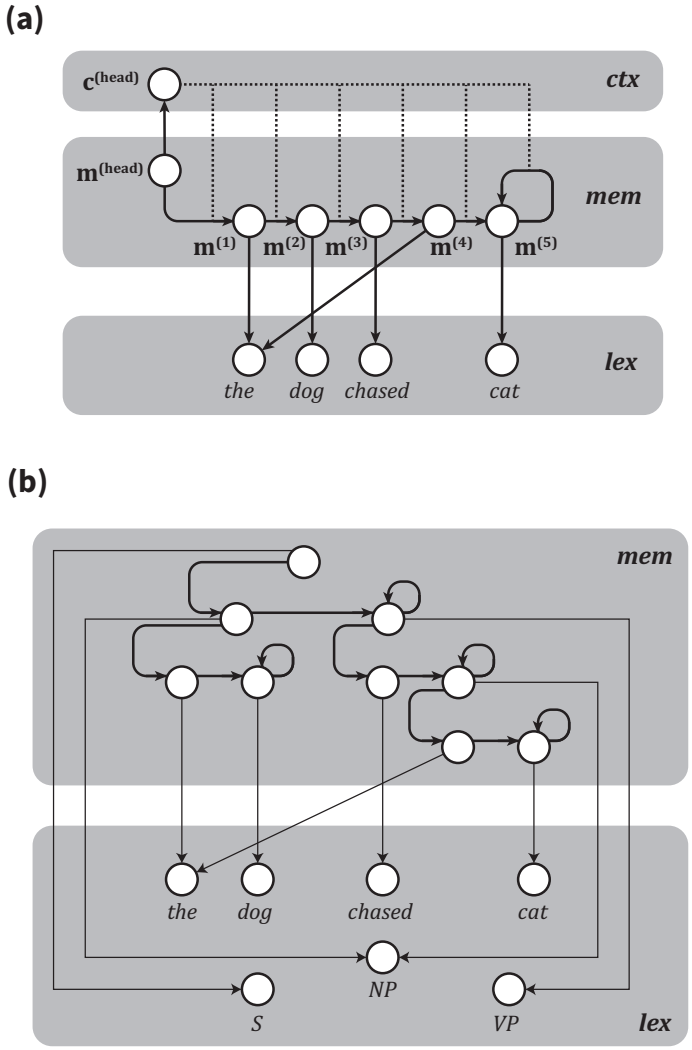

Figure 4: Graphical depiction of compositional data structures. Each gray rectangle represents the activity space of a region of the neural model (*ctx*, *mem*, and *lex*), and each circle represents a unique activity pattern (distributed representation). (a) A linked list representing the sentence "the dog chased the cat". The list is represented by a trajectory through memory states (middle gray rectangle) that terminates with a self-loop transition. Each transition in the trajectory is contextualized by a list-specific context state ($\mathbf{c}^{(head)}$, top left) that is associated with a list head memory state ($\mathbf{m}^{(head)}$, middle left). Each element in the list is represented by a unique memory state ($\mathbf{m}^{(1)}$ through $\mathbf{m}^{(5)}$, center) that is associated with a pattern of activity in *lex* representing the corresponding word in the sentence (bottom). (b) A parse tree for "the dog chased the cat" represented as a list of lists. Each internal node of the tree is represented by a memory state that serves as the head of a list containing its child nodes (trajectories in top rectangle). The context patterns for these trajectories are omitted for clarity. Each node is associated with a pattern in *lex* representing its symbolic content (*S* for "sentence", *NP* for "noun phrase", *VP* for "verb phrase", or a word in the sentence).

the trajectory through its children and can be retrieved during traversal. However, child nodes cannot be accessed in arbitrary order, and parent-child relations do not have associated labels. This organization is shown in Figure 4b.

2.3. Programmatic Control of Compositional Memory

The compositional data structures described in the previous section can be constructed and manipulated via top-down control of the *mem*, *ctx*, and *lex* regions over time. Figure 5 shows a programmable neural network with additional regions that provide this control. In this section we first describe the functionality of these control mechanisms, including the representation and execution of learned programs that make use of compositional memory. Then

we describe a planning task that the model learns to perform, which involves constructing and modifying complex hierarchical data structures in memory.

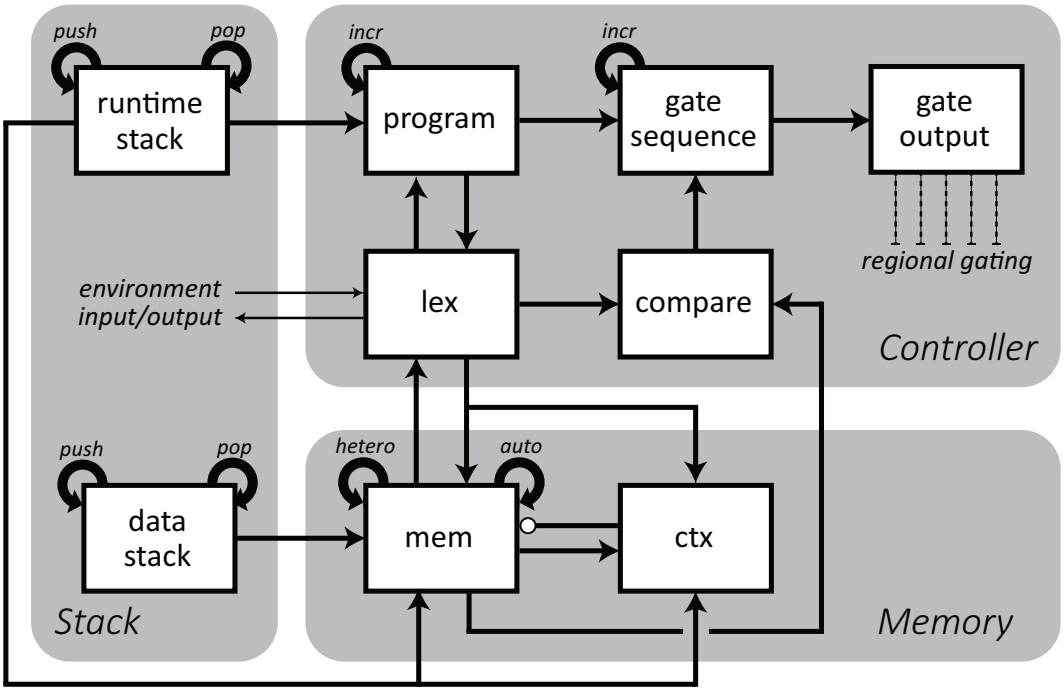

Figure 5: Programmable neural network with compositional memory. Neural regions (boxes) are interconnected with gated pathways (solid lines). The architecture of the model resembles a stack machine, and includes three subnetworks. **Controller:** The controller subnetwork (top right) controls model execution based on learned programs. Neurons in the *gate output* region (top right) determine which pathways are active during each timestep of model execution. This region is controlled by the *gate sequence* region, which encodes a sequence of gating operations that control the flow of information through the model over time. These gate sequences correspond to instruction opcodes for programs encoded in the *program* region. Instruction operands are represented in the *lex* region (center), which encodes a lexicon of recognized symbols as activity patterns. This region serves as a bridge between the controller and memory subnetworks, and is used to pass symbolic information into and out of the model (left center). **Memory:** The core region of the memory subnetwork is the *mem* region (bottom center), which is an attractor graph network with dense hetero-associative and auto-associative connectivity (bold looped arrows). Patterns of activity in *mem* represent general-purpose memory states that "store" symbolic tokens represented in *lex*. Associations between memory states and symbolic tokens are learned in the pathway from *mem* to *lex*. Transitions between attractors in *mem* are contextualized by activity in the context region (*ctx*, bottom right) via a pathway from *ctx* to *mem*. The open circle at the end of this pathway indicates that it provides multiplicative contextual gating inputs to *mem*. Activity patterns in *ctx* may be derived from *mem* or *lex* patterns, and serve as labels for relations between memory states that are used to construct compositional data structures. **Stack:** The stack subnetwork (left) contains two regions with bidirectionally associated activity patterns that represent stack frames. The *runtime stack* region stores and retrieves pointers to program instructions, memory states, and context states. These are stashed when program subroutines are called, and retrieved upon return to the caller. The *data stack* region stores pointers to memory states that are used for operations involving multiple memory states, and is used to pass arguments between subroutines.

The complete model depicted in Figure 5 is a programmable neural network with program-independent circuitry that is based on the Neural Virtual Machine (NVM) (Katz et al., 2019), but has several key differences. Most notably, the sequential tape-like memory of the NVM is replaced by attractor graph memory (*mem* and *ctx*), permitting storage of compositional data structures via direct context-dependent associations between memory states. In addition, our model functions as a stack machine rather than a register machine, simplifying its instruction set and program circuitry. Symbols are represented in a single region (*lex*) rather than multiple register and operand regions as in the NVM, and a *data stack* region is used for operations with multiple operands.

Our model includes three major subnetworks, shown as large gray rectangles in Figure 5. The memory subnetwork contains the *mem* and *ctx* regions, which implement compositional memory with attractor graphs, as described in Sections 2.1 and 2.2. The *lex* region is contained in the controller subnetwork, and serves as a bridge between several model components. Patterns in the *lex* region serve several functions, some of which were described in Section 2.2.

They represent symbolic tokens that can:

- be stored in memory states (*mem* to *lex*)
- be variable names that refer to data structures in memory (*lex* to *mem*)
- label transitions in *mem* attractor graphs that represent key-value relations in associative arrays (*lex* to *ctx*)
- be used as program instruction operands (*program* to *lex*)
- refer to program subroutines (*lex* to *prog*)
- be printed to or read from the environment (*lex* to/from environment)

The controller subnetwork also contains several regions that control connection gates based on learned programs. Neurons in the *gate output* region determine which pathways in the model are active during each timestep (one neuron per connection gate). This region is controlled by the *gate sequence* region, which encodes sequences of gating operations that control the flow of information through the model over time, much like in conventional computer architectures. These gate sequences correspond to instruction opcodes for programs encoded in the *program* region, while instruction operands are represented by *lex* activity patterns.

The *compare* region encodes true and false patterns that are used to perform conditional jumps in programs based on comparisons between memory states or lexical symbols. For example, an instruction might jump to a subroutine if two memory states store the same symbol. If the comparison yields the *true* state, the model jumps to the subroutine specified by the jump instruction's operand. Otherwise, it advances to the next instruction in the sequence.

The stack subnetwork contains two regions with bidirectionally associated activity patterns that represent stack frames. The *runtime stack* region stores a call-stack that maintains pointers to program instructions, memory states, and context states. These are pushed when program subroutines are called, and popped upon return to the caller. The *data stack* region maintains pointers to memory states that are used for operations involving multiple memory states. For example, when a transition is learned, the target memory state is assumed to be currently active in *mem*, and a pointer to the source memory state is stored on the top of the data stack. These stacks are explained further in the following section.

2.3.1. Program Storage and Execution

As mentioned above, programs are represented by sequences of activity patterns in the *program* region. Each activity pattern represents an individual instruction with an opcode and optional operand. Each opcode corresponds to a sequence of patterns in the *gate sequence* region that implements the operation as a temporal sequence of connection gates. Each instruction is associated with the first pattern in the corresponding opcode sequence via the pathway from *program* to *gate sequence*. Similarly, an instruction with an operand is associated with a *lex* activity pattern representing the operand value.

Gating operations and program sequences are established during a one-time associative learning procedure that is analogous to firmware "flashing" in a non-volatile microcontroller memory. The model architecture supports operations that can be expressed as temporal sequences of active connection gates, such as the attractor transition procedure specified in Table 1. Each timestep of an operation is represented by a randomly generated activity pattern in the *gate sequence* region, and is associated with a *gate output* pattern that specifies the active connection gates. The final pattern of most operation sequences is associated with a common gating sequence that advances the *program* region to the next instruction, and opens the pathway from *program* to *gate sequence*, initiating the gating sequence for the next instruction. The exception is jump instructions, which have conditional behavior that depends on comparison operations (explained in Section 2.3.3).

In addition to the above associations, the "flashing" procedure also establishes associations in the stack regions (*runtime stack* and *data stack*). Activity patterns in these regions represent individual stack frames that can be associated with activity patterns in other regions. Each stack frame is associated with the frame above and below it in the stack using distinct recurrent hetero-associative matrices (*push* and *pop* loops in Figure 5).

Finally, the model is "flashed" with a lexicon of recognizable symbols in the lex region. Each symbol pattern is associated with a unique *ctx* pattern, allowing the symbol to serve as an attractor transition label (i.e., an associative array key).

2.3.2. Online Learning

Program execution involves online updating of connectivity matrices. These updates are determined by distinct gates that control plasticity:

$$W_{r,q}(t+1) = W_{r,q}(t) + g^{\ell}_{r,q}(t)\Delta W_{r,q}(t) \tag{12}$$

where $W_{r,q}(t)$ is a weight matrix connecting region q to region r at time t, and $g^{\ell}_{r,q}(t)$ is a learning gate that determines when this matrix is updated. For pathways connecting distinct regions, $\Delta W_{r,q}(t)$ can be computed using the store-erase learning rule with the active patterns in r and q:

$$\Delta W_{r,q}(t) = \frac{1}{\rho_q^2 N_q}\Big(\sigma_r^{-1}(\mathbf{v}_r(t)) - W_{r,q}(t)\mathbf{v}_q(t)\Big)\mathbf{v}_q(t)^{\top} \tag{13}$$

where $W_{r,q}(t)$ is a weight matrix connecting region q to region r at time t, $\mathbf{v}_r(t)$ and $\mathbf{v}_q(t)$ are the activation patterns of r and q, σ_r is the activation function of neurons in r, ρ_q is the stable activation level of neurons in region q, and N_q is the number of neurons in region q.

The recurrent auto-associative matrix in *mem* is also updated using the currently active pattern:

$$\Delta A_r(t) = \frac{1}{\rho_r^2 N_r}\Big(\sigma_r^{-1}(\mathbf{v}_r(t)) - A_r(t)\mathbf{v}_r(t)\Big)\mathbf{v}_r(t)^{\top} \tag{14}$$

where $A_r(t)$ is the auto-associative weight matrix for region r at time t, and all other terms are as defined above.

Online learning of recurrent hetero-associative matrices involves distinct source and target patterns that cannot be simultaneously active. To address this, we introduce an eligibility trace $\epsilon_r(t)$ that stores a target activity pattern for subsequent learning:

$$\epsilon_r(t+1) = \begin{cases} \mathbf{c}_r(t) \odot \mathbf{s}_r(t), & \text{if } g^{\epsilon}_r(t) = 1 \\ \epsilon_r(t), & \text{otherwise} \end{cases} \tag{15}$$

where $\epsilon_r(t)$ is the eligibility trace of region r at time t, $g^{\epsilon}_r(t)$ is a gate that determines when the eligibility trace is updated, and $\mathbf{c}_r(t)$ and $\mathbf{s}_r(t)$ are synaptic and multiplicative inputs at time t (defined in Equations 1 and 2). When $g^{\epsilon}_r(t) = 1$, the current gated inputs are stashed in the eligibility trace. To learn a transition to the stashed pattern, a source pattern is activated, and the hetero-associative learning gate is opened:

$$\Delta H_r(t) = \frac{1}{\lambda_r \rho_r^2 N_r}\Big(\epsilon_r(t) - \big(\mathbf{c}_r(t) \odot H_r(t)\mathbf{v}_r(t)\big)\Big)\mathbf{v}_r(t)^{\top} \tag{16}$$

where $H_r(t)$ is the hetero-associative weight matrix for region r, $\epsilon_r(t)$ is the eligibility trace defined above, $\mathbf{c}_r(t)$ is the multiplicative input to region r, λ_r is a context density for multiplicative gating of region r, and all other terms are as defined above. Note that the contextual gating is already applied to the source pattern $\mathbf{v}_r(t)$ (Equation 1).

2.3.3. Comparisons

The comparison region has unique dynamics that allow it to memorize an input pattern from another region for subsequent recognition. When an input pattern is memorized, it is associated with a *cmp* activity pattern representing *true* ($\mathbf{v}^{true}_{cmp}$). Unlike other connectivity updates, this overwrites the corresponding weight matrix rather than incrementally updating it:

$$\Delta W_{cmp,q}(t) = \frac{1}{\rho_q^2 N_q}\sigma^{-1}_{cmp}\Big(\mathbf{v}^{true}_{cmp}\Big)\mathbf{v}_q(t)^{\top} - W_{cmp,q}(t) \tag{17}$$

where $W_{cmp,q}(t)$ is a weight matrix connecting region q to the compare region at time t, $\mathbf{v}_q(t)$ is the activation pattern of region q, σ_{cmp} is the activation function of neurons in *cmp*, ρ_q is the stable activation level of neurons in region q, and N_q is the number of neurons in region q. This learning rule resembles Equation 13, except that it completely overrides the existing weights and associates the input pattern with a fixed pattern ($\mathbf{v}^{true}_{cmp}$).

The *cmp* region has an input bias toward a *false* activity pattern ($\mathbf{v}^{false}_{cmp}$) that can be overcome by the memorized input. Thus, if an input pattern is close enough to the memorized pattern, the resulting *cmp* activation pattern is $\mathbf{v}^{true}_{cmp}$; otherwise it is $\mathbf{v}^{false}_{cmp}$ The following equation describes the synaptic input term for the *cmp* region:

$$\mathbf{s}_{cmp}(t) = \underbrace{g^S_{cmp}(t)\ \omega_{cmp}\ \mathbf{v}_{cmp}(t)}_{\text{saturation}} - \underbrace{\Big(1 - g^S_{cmp}(t)\Big)\ \theta\ \omega_{cmp}\mathbf{v}^{false}_{cmp}}_{\text{bias to false pattern}} + \underbrace{\sum_q \Big(g_{cmp,q}(t)\ W_{cmp,q}(t)\ \mathbf{v}_q(t)\Big)}_{\text{inter-regional input}} \tag{18}$$

where $\mathbf{v}^{false}_{cmp}$ is the *false* activity pattern in *cmp*, θ is a comparison similarity threshold (typically > 0.95), and all other terms are as defined for Equation 8 (with $r = cmp$). The synaptic input $\mathbf{s}_{cmp}(t)$ is transformed to neural activation $\mathbf{v}_{cmp}(t)$ by Equation 9 (again with $r = cmp$). The threshold θ determines how similar an input pattern needs to be to the memorized input pattern in order to activate the *true* pattern ($\mathbf{v}^{true}_{cmp}$). Specifically, a value of $\theta = 0.95$ means that an input pattern must have a cosine similarity exceeding 0.95 in order to activate $\mathbf{v}^{true}_{cmp}$.

The *true* and *false* states in *cmp* are associated with distinct sequences in the *gate sequence* region. When a jump instruction is executed, the pathway from *cmp* to *gate sequence* is opened, and the *program* region is advanced according to the result of the most recent comparison. The *true* gate sequence activates the first instruction of the subroutine indicated by the jump instruction operand. This is done by opening the pathway from *program* to *lex*, followed by the pathway from *lex* back to *program*. The *false* gate sequence simply opens the recurrent hetero-associative connection gate in the *program* region, advancing it to the next instruction in the current subroutine.

2.3.4. Generating Memory States

The model's instruction set includes operations that allocate memory for construction of data structures. Memory states are learned attractors in the *mem* region's auto-associative connectivity. We include a gated noise term in the *mem* and *ctx* regions. When a noise gate is opened, a random pattern of activation is established in the corresponding region:

$$\mathbf{v}_{mem}(t+1) = \sigma_{mem}\Big(\mathbf{x}_{mem}(t) \odot \mathbf{s}_{mem}(t) + \underbrace{g^N_{mem}\omega_{mem}\mathbf{n}_{mem}(t)}_{\text{gated noise}}\Big) \qquad \mathbf{n}_{mem}(t) \sim Bernoulli(0.5) \tag{19}$$

$$\mathbf{v}_{ctx}(t+1) = \sigma_{ctx}\Big(\mathbf{s}_{ctx}(t) + \underbrace{g^N_{ctx}\omega_{ctx}\mathbf{n}_{ctx}(t)}_{\text{gated noise}}\Big) \qquad \mathbf{n}_{ctx}(t) \sim Bernoulli(\lambda) \tag{20}$$

where g^N_{mem} and g^N_{ctx} are noise gates for the *mem* and *ctx* regions, and $\mathbf{n}_{mem}(t)$ and $\mathbf{n}_{ctx}(t)$ are random vectors. All other terms are as defined in Equation 3 (for *mem*) and Equation 9 (for *ctx*). Random vectors are generated by a Bernoulli process with probabilities 0.5 and λ (context density, defined in Section 2.1). Random patterns in *mem* are in $\{-\rho_{mem}, +\rho_{mem}\}^{N_{mem}}$, where N_{mem} is the number of neurons in *mem*, and ρ_{mem} is the steady-state magnitude of saturation dynamics in *mem* ($\rho_{mem} = \sigma_{mem}(\omega_{mem})$). Random patterns in *ctx* are binary patterns.

2.3.5. I/O

Environmental inputs to the model are provided to the *lex* region via the external input variable $\mathbf{I}_{lex}$ (Equation 8). The model indicates when it is ready to receive input by activating a dedicated read gate g^R_{lex}. Environmental input is specified as a sequence of symbols from an alphabet that the model is pre-trained to recognize. Each alphabet symbol is mapped to a unique activity pattern. When the environment detects that $g^R_{lex}(t) = 1$, $\mathbf{I}_{lex}(t)$ is set to the activation pattern corresponding to the next unread symbol in the input sequence.

Similarly, a dedicated write gate g^W_{lex} indicates to the environment when the model is ready to provide output. When $g^W_{lex}(t) = 1$, the environment captures the current *lex* activity pattern $\mathbf{v}_{lex}(t)$ and translates it into a symbol by identifying the closest activity pattern in the alphabet mappings.

2.3.6. Planning Task

Planning is a high-level executive task that involves reasoning about actions and organizing them to achieve goals (Ghallab et al., 2004; Erol, 1996). Behavioral plans are often hierarchically structured and require compositional reasoning (Botvinick, 2008; Dehaene and Changeux, 1997). To further test our model's ability to construct and maintain compositional data structures in memory, we trained it to perform an automated planning task using hierarchical task networks (HTNs) (Ghallab et al., 2004; Erol, 1996). An HTN is a tree representing the decomposition of a high-level compound task (root node) into concrete primitive actions (leaf nodes). Each internal node is broken down into subactions (compound or primitive) according to learned rules in a knowledge-base that may depend on environmental states. For example, opening a door may involve different motor behaviors depending on what type of door it is (e.g., pushing, pulling, sliding, etc). During planning, an agent recursively decomposes a top-level task to produce a sequence of primitive actions that is appropriate for the given environment.

We implemented a version of HTN planning with the following restrictions. The environment is represented by an associative array of named feature bindings (e.g., "door type" = "sliding"). Each compound action is decomposed according to the value of a specific environmental feature (e.g., "open door" is decomposed according to the value of "door type"). When there is no rule for a given action and feature value, the action is treated as a primitive action, and is not further decomposed. The task is performed as follows. First, the model reads in the knowledge-base rules and environmental bindings and stores them in memory. Then, it reads in a sequence of top-level actions for planning. An HTN is constructed by recursively decomposing each top-level action into primitive actions based on the knowledge-base and environmental bindings. Once the plan is complete, the model performs a pre-order traversal of the HTN and prints out the action for each node.

The knowledge-base is stored in attractor graph memory as a nested map (associative array). The keys of the top-level map are compound actions, and the values are inner maps containing decomposition rules for each action. As mentioned above, each action is decomposed according to the value of a specific environmental binding. The key for this binding is stored in the inner map memory state, and is used to query the environment for the binding's value. This value is then used as a key for the inner map to retrieve the corresponding decomposition rule, which is represented as a linked list of sub-actions. During decomposition, knowledge-base lookups are validated as described in Section 2.2, and actions are only decomposed if a rule is successfully retrieved.

For testing purposes, we designed a planning domain to simulate a simple repair task involving a mechanical assembly unit (see Appendix for details). The unit has a door on the front, an indicator for the status of the unit, and an interaction point for performing repairs. The full task involves opening the door according to its type, performing a repair according to the status indicator, and closing the door. Each stage of the task is performed based on a set of environmental bindings describing a specific unit, including the type of door, status indicator, and repair interaction point (e.g., sliding door, LED indicator, keypad interaction point).

3. Results

The model outlined above was implemented in Python using the NumPy scientific computing library, and was tested in several stages. The first three stages evaluated the memory capacity of the AGN model, and involved learning attractor graphs in the *mem* region. Specifically, we sought to empirically determine 1) the number of attractor states that can be learned and reliably retrieved from partial patterns, 2) the number of unique context-dependent transitions from a single source pattern that can be learned (i.e., the branching factor of attractor graphs), and 3) the total number of transitions that can be learned in an attractor graph.

The fourth and fifth stages of testing evaluated storage and retrieval of compositional data structures, and included learning in the inter-regional pathways of the model. First, the model was trained with attractor graphs representing linked lists to determine whether errors in attractor transitions compound during traversal, and whether memory states can be effectively reused in multiple distinct list structures. Then, the model was trained with attractor graphs representing sentence parse trees. Each node in a tree was represented as a pattern of activity in *mem*, and each symbol stored in the node was represented by a pattern in *lex*. The associations between nodes and symbols were learned in the pathway from *mem* to *lex*.

In the final stage of testing, we evaluated the model's ability to manipulate compositional data structures using the HTN planning task described in Section 2.3.6. Each test involved learning a knowledge-base of decomposition rules,

a set of environmental bindings, and a sequence of top-level compound actions. These top-level actions were decomposed into HTNs according to the knowledge-base and environmental bindings. We evaluated runtime complexity by varying the top-level action sequence to vary the size of the constructed HTN. Three conditions were evaluated: 1) a baseline condition using a small knowledge-base (9 rules spread across 5 compound actions) and small environment (4 bindings), 2) an extended knowledge-base of 15 rules across 7 compound actions, and 3) an extended environment containing 8 bindings. We validated that the model constructed the correct HTN in each test, and measured the number of timesteps taken to complete the task. The specific knowledge-bases, environments, and top-level sequences used are listed in the Appendix.

In all stages of testing, performance was evaluated by comparing patterns of activity with learned target patterns. For example, to evaluate a learned attractor transition from memory pattern $\mathbf{m}_A$ to memory pattern $\mathbf{m}_B$, the trained model would be initialized with $\mathbf{m}_A$ in the *mem* region, the transition would be executed (including attractor convergence), and the resulting *mem* activity pattern would be compared with $\mathbf{m}_B$. Because saturation dynamics can correct any convergence errors within an orthant of activity space, two patterns are considered identical if they reside within the same orthant (i.e., the sign of each neuron's activation matches). For each experiment, results are reported as the percentage of activity patterns that matched their corresponding targets.

Experiments in Sections 3.1 - 3.5 evaluated the model shown in Figure 1 using either traditional Hebbian learning or the fast store-erase learning rule. Pathways with context-dependent dynamics were learned with the gated versions of these rules (Equations 4 - 7). All other pathways were learned with non-gated versions of these rules (Equations 10 and 11. Each region of the model contained $N = 1024$ neurons. The *mem* region used the hyperbolic tangent activation function and learned activity patterns with magnitude $\rho = 0.9999$. The *ctx* and *gate output* region used the heaviside activation function, and learned binary patterns ($\rho = 1$). The *lex* region used the sign/signum activation function ($\rho = 1$). The experiment in Section 3.6 evaluated the full programmable network shown in Figure 5. In this experiment, regions were sized according to the number of patterns to be learned for the planning task described in Section 2.3.6. Regions not present in Figure 1 used the sign/signum activation function ($\rho = 1$).

3.1. Attractor Convergence

Attractor transitions are carried out in multiple steps, starting with contextually-gated hetero-associative dynamics, followed by auto-associative attractor convergence and activity saturation. Because the hetero-associative step results in a partial pattern of activity (some neurons have zero activation), successful transitions depend on accurate auto-associative pattern completion. We therefore sought to determine the number of activity patterns that can be learned as attractors and successfully recovered from partial patterns.

The density of partial patterns (i.e., the number of non-zero elements) encountered during attractor transitions depends on the parameter λ, the probability used to generate context patterns (Section 2.1). This parameter indicates the number of neurons in the *mem* region that participate in hetero-associative dynamics, and consequently the number of active neurons prior to attractor convergence.

The *mem* region AGN was trained with sets of randomly generated memory states, and evaluated for pattern completion. One trial of testing involved learning a set of M memory states generated by a Bernoulli process with probability 0.5 (i.e., a fair coin toss for each neuron's activation). Each learned memory pattern was evaluated by initializing the network with a partial version of the pattern, running auto-associative dynamics and saturation, and comparing the resulting activity pattern with the original learned pattern. The number of timesteps for auto-associative dynamics was set to 10, which was found to be sufficient for attractor convergence in preliminary testing. Partial patterns were produced by randomly setting $N(1 - \lambda)$ elements to zero, simulating contextual gating. This process was repeated 8 times for each memory state, and the value of λ was varied experimentally.

Results are shown in Figure 6. Each plot shows results for networks trained with a common learning rule (store-erase or traditional Hebbian learning) and various values of λ. Each line shows accuracy of attractor convergence as the number of learned memory states M increases. Accuracy deteriorated as the density of the context pattern (λ) was decreased and the masked partial patterns became sparser. Results for the two learning rules were comparable, but the store-erase rule showed slightly more gradual degradation with increasing numbers of stored patterns. Perfect accuracy can be achieved with the store-erase rule when $M = 64$ memory patterns are stored. In subsequent experiments, we limit the number of learned attractors to 64 to avoid errors in attractor convergence.

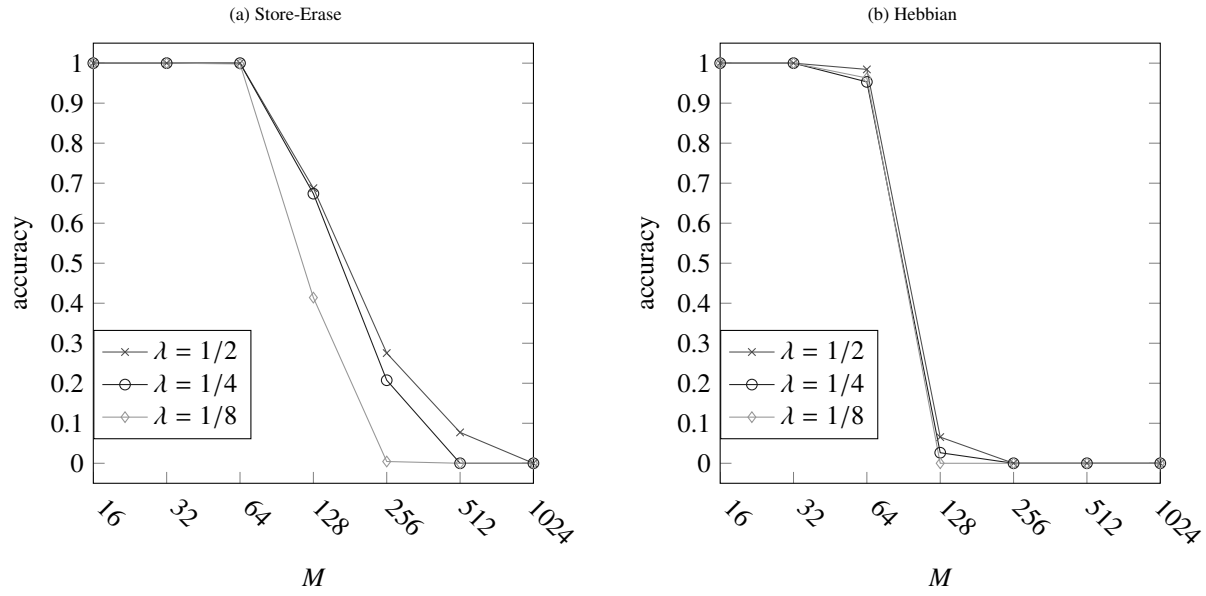

Figure 6: Accuracy of attractor convergence (pattern recovery/completion). Each plot shows the recall accuracy (y-axis) of an AGN with $N = 1024$ neurons trained with either the fast store-erase learning rule (left) or traditional Hebbian learning (right). The network was trained with sets of memory patterns of various sizes (M, x-axis). Each learned memory pattern was tested 8 times by initializing the network with a partial version of the pattern, running auto-associative dynamics for 10 timesteps, and comparing the resulting activity pattern with the learned pattern. The density of the partial version was determined by the parameter λ, and each line indicates results for a single value of λ. Each data point indicates the percentage of convergence trials resulting in perfect pattern recall, for a total of $8M$ trials per data point.

3.2. Transition Branching

Functional branching is a novel aspect of AGNs that makes it possible to learn multiple transitions from a single attractor state using a single hetero-associative weight matrix (Section 2.1.1). We refer to the number of transitions from an attractor as the attractor's *branching factor*. We evaluated learning of attractor graphs with various branching factors to determine how many transitions can be learned from a single source attractor. Specifically, we trained AGNs with attractor graphs organized as directed stars, where one attractor serves as an internal node with transitions to several leaf nodes.

The results for attractor convergence above indicate that a network of $N = 1024$ neurons can reliably learn $M = 64$ attractors. We therefore generated attractor graphs with $M = 64$ attractors and varying numbers of transitions. One attractor was designated as the internal node, and each transition targeted an attractor randomly chosen from the remaining 63 attractors. Each transition was learned with a unique context pattern generated by a Bernoulli process with probability λ (varied experimentally). Note that there may be multiple transitions from the internal node to the same leaf node.

Results are shown in Figure 7. Each data point indicates the percentage of transitions that were successfully executed after learning. In contrast to the pattern recovery results, accuracy was higher with smaller λ. This is likely due to decreased weight sharing across contexts. The store-erase rule significantly outperformed traditional Hebbian learning, yielding high accuracy (over 97%) for branching factors up to 1024 for $\lambda = \frac{1}{4}$ and $\lambda = \frac{1}{8}$.

3.3. Random Graphs

Having established that AGNs can learn attractor graphs with high branching factors, we sought to determine how many transitions can be stored when the transitions do not share a source node. To do so, we trained the AGN with randomly generated graphs of $M = 64$ attractors and varying numbers of transitions (T). Graphs were generated by randomly selecting a source attractor (vertex), target attractor (vertex), and context pattern (edge label) for each transition (edge). The number of available context patterns was set to the maximum branching factor of nodes in the

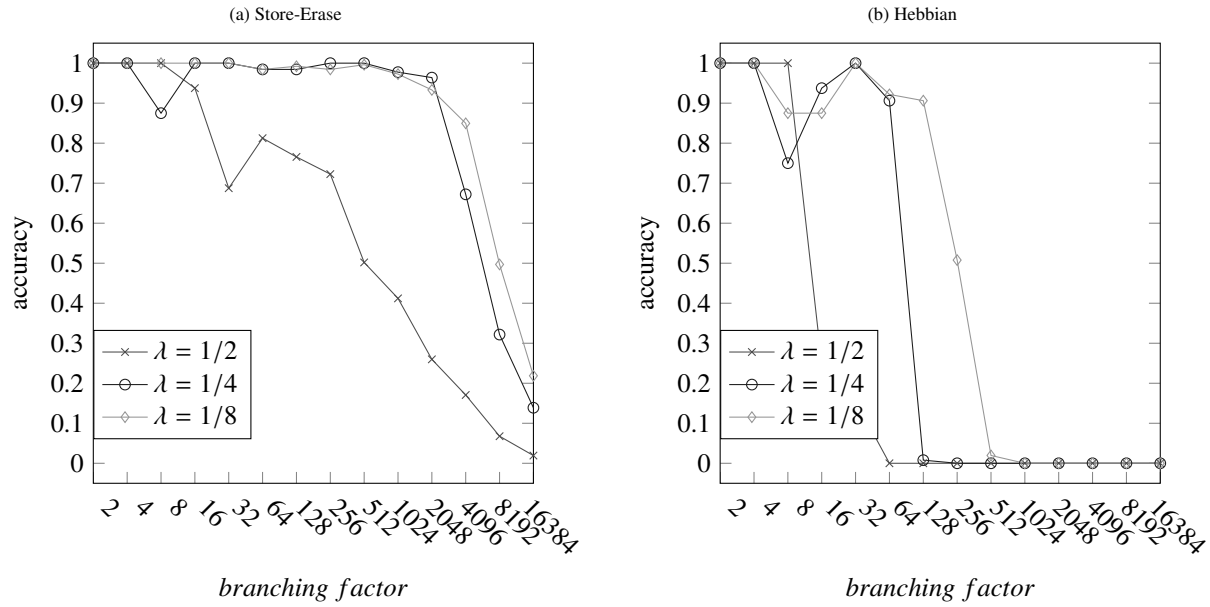

Figure 7: Accuracy of attractor graphs with various branching factors. Each plot shows the transition accuracy for an AGN with $N = 1024$ neurons trained with either the fast store-erase learning rule (left) or traditional Hebbian learning (right). The network was trained with attractor graphs containing $M = 64$ memory attractors and various numbers of transitions (x-axis). The attractor graphs were organized as directed stars, where one internal node transitions to several leaf nodes in different contexts. Multiple transitions to the same leaf node in different contexts were allowed, making it possible to test branching factors larger than the total number of leaf nodes. Each line indicates results for one value of λ, the density of context patterns used for transitions. Each data point indicates the percentage of successful transitions.

graph, ensuring that no two transitions shared the same source and context patterns. Transitions from different source patterns were allowed to share context patterns.

Results are shown in Figure 8. Store-erase learning was more reliable than traditional Hebbian learning, with less variable accuracy. With $\lambda = \frac{1}{4}$, both learning rules yielded high accuracy with $T = 1024$ transitions. In subsequent experiments, we limit the number of learned attractor transitions to 1024.

3.4. Linked Lists

The above results evaluate the integrity of individual attractor transitions and convergence events in an AGN. Iteration through attractor graphs representing compositional data structures involves sequences of transitions in which errors might compound. We therefore evaluated retrieval of linked lists with itinerant traversals, allowing any errors in transitions to compound.

The model was trained with attractor graphs of $M = 64$ attractors encoding linked lists (Section 2.2). Each attractor served as the head of a unique list containing E elements (varied experimentally) drawn from the remaining 63 attractors. Each list is encoded as a trajectory containing $E + 1$ transitions, for a total of $64(E + 1)$ transitions in the graph. Note that an attractor (memory state) may be contained in more than one list, as each list's transitions use a unique list-specific context pattern. These context patterns were learned in the pathway from the *mem* to *ctx* regions (Figure 1), and were retrieved at the beginning of each traversal at testing time.

Results are shown in Figure 9. Accuracy drops sharply after $E = 15$ elements, or a total of $T = 1024$ transitions. This corresponds to the point at which accuracy declines in Figure 8. The drop in accuracy after $E = 15$ is therefore likely due to limits in transition capacity. These results show that below this capacity, traversals through attractor graphs can be executed without compounding errors. In addition, large numbers of linked lists can be encoded in attractor graphs, and memory states can be successfully shared between lists.

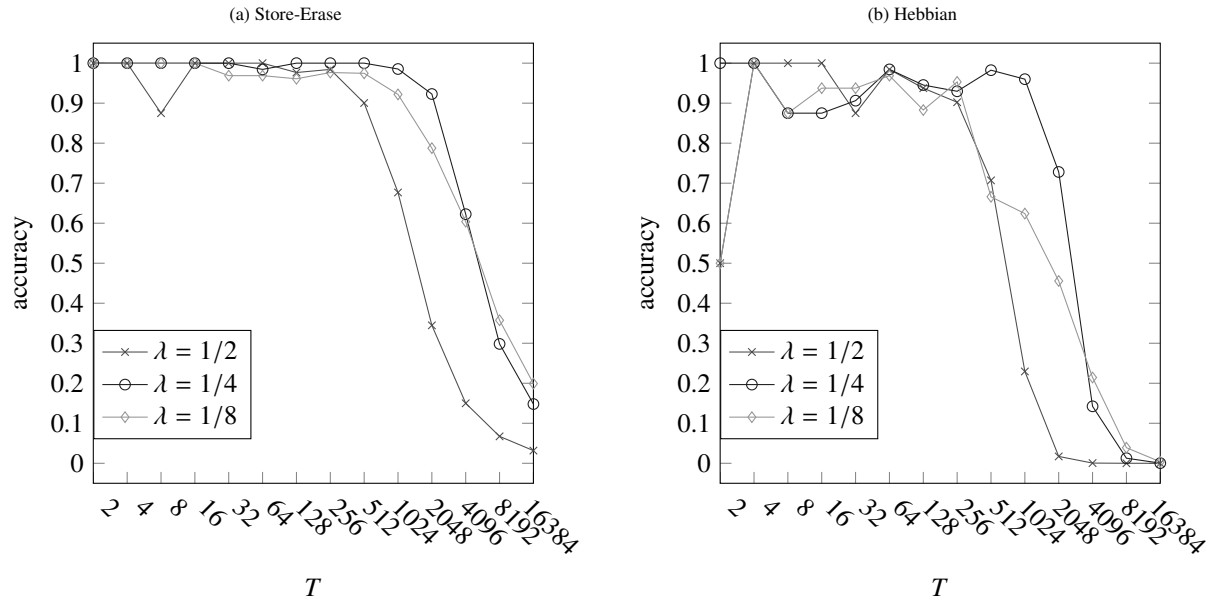

Figure 8: Accuracy of randomly generated attractor graphs. Each plot shows the transition accuracy for an AGN with $N = 1024$ neurons trained with either the fast store-erase learning rule (left) or traditional Hebbian learning (right). The network was trained with attractor graphs containing $M = 64$ memory attractors and various numbers of transitions (x-axis). Each transition in the graph connected two randomly selected attractors using a randomly generated context pattern. Each line indicates results for one value of λ, the density of context patterns used for transitions. Each data point indicates the percentage of successful transitions.

3.5. Parse Trees

The results above show that list traversals can be successfully carried out without compounding errors when the total number of attractors is limited to $M = 64$. That experiment did not evaluate storage and retrieval of symbolic information via the pathway from *mem* and *lex*, and did not involve retrieval of context patterns in *ctx* using memory states that may contain minor errors. To address these limitations, we evaluated the model with storage of parse trees with a symbol stored at each node.

Parse trees were randomly selected from the Penn Treebank corpus[1]. Each tree was encoded as an attractor graph using lists of lists, where each node is represented as a list of its children. Each node in the tree was assigned to a memory attractor, and each symbol in the parse tree was assigned to a randomly generated pattern of activity in the *lex* region using a Bernoulli process with probability 0.5. Associations between nodes and symbols were learned in the pathway from *mem* to *lex* using either the store-erase learning rule or traditional Hebbian learning.

The model learned one tree at a time, and weights were reset after learning and evaluating each tree. Each tree was evaluated with a traversal starting with initialization of *mem* with the root node pattern. As with list testing, errors were allowed to propagate during traversal, and context patterns were retrieved using the pathway from *mem* to *lex*. After each transition, the symbol stored in the current node was retrieved using the pathway from *mem* to *lex*, and results are reported as the percentage of symbol patterns in *lex* that were correctly recovered. An external queue was used to store and retrieve intermediate activity patterns and perform a breadth-first traveral. Note that this queue is not considered part of the model, and is only used for evaluation purposes.

Results are shown in Figure 10. A total of 100 randomly selected trees were learned. Each mark indicates the percentage of perfectly recalled symbol patterns across all nodes in a single tree (y-axis), and the x-axis (log-scale) indicates the number of nodes in the tree. In accordance with the results in Section 3.1, accuracy begins to deteriorate with trees containing more than 64 nodes. Past this point, accuracy degrades more gradually with the store-erase rule than with traditional Hebbian learning.

[1]Parse trees were retrieved from the *Penn Treebank Sample* dataset of the Python Natural Language Toolkit, found at *http://www.nltk.org/nltk_data/*

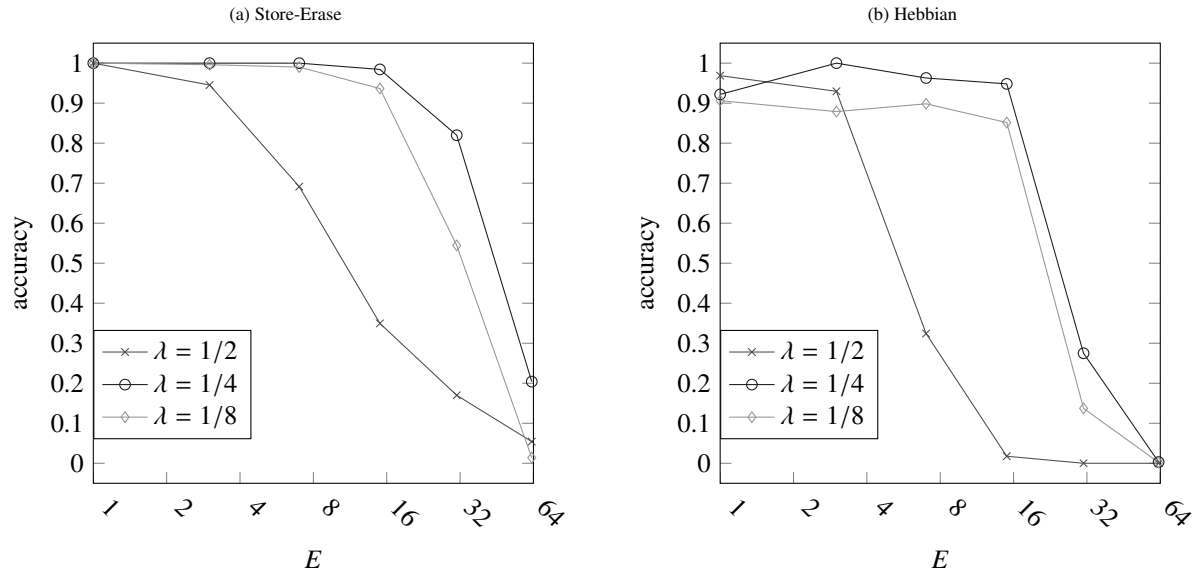

Figure 9: Accuracy of attractor graphs encoding multiple linked lists. Each plot shows the transition accuracy for an AGN with $N = 1024$ neurons trained with either the fast store-erase learning rule (left) or traditional Hebbian learning (right). The network was trained with attractor graphs containing $M = 64$ memory attractors, each representing the head of a linked list containing E elements (x-axis). Each list contained a random permutation of E attractors (not including the list's head attractor), and was encoded as a trajectory through the attractor graph. The total number of transitions in the attractor graph is $64(E + 1)$. Performance was evaluated by executing traversals through each list starting with the head pattern, and errors in transitions were allowed to propagate during traversal. Each line corresponds to a unique context density λ, and each data point indicates the percentage of successful transitions.

The store-erase learning rule contains an anti-Hebbian component that erases previously learned associations. To evaluate this unique contribution, we performed a second experiment with parse trees of at most 64 nodes each. In this experiment, the model weights were not reset in between learning each tree. A set of $M = 64$ attractors was learned and made available for construction of each tree, which contained a unique set of transitions between attractors, as well as associations between attractors and context/lexicon patterns.

Results are shown in Figure 11 for 30 trials. The left plot shows accuracy of *lex* pattern recovery for the two learning rules with $\lambda = \frac{1}{4}$. Accuracy for Hebbian learning (dashed line) drops to zero after the first trial, as learned associations compound and interfere with one another. Because the store-erase rule allows overwriting of associations via controlled erasure, accuracy remains fairly high across the trials, but dips to as low as 75% (solid line).

To determine the extent of pattern deterioration, patterns in *lex* representing stored symbols were compared with a fine-grained similarity metric rather than all-or-nothing comparison. This similarity metric measured the percentage of *lex* neurons with activation matching the target symbol's activation pattern. Results are shown in the right plot of Figure 11 for $\lambda = \frac{1}{4}$. Each data point indicates the average similarity of *lex* activity patterns retrieved from tree nodes. The solid line shows that the average similarity with the store-erase rule is nearly perfect (average similarity of 0.9998 across trials) despite the dips in overall accuracy in the left plot. This indicates that errors in pattern retrieval are minimal, and involve very small numbers of neurons with activation that did not match the target pattern. In contrast, the average similarity with Hebbian learning drops rapidly to around 50%, which is the expected similarity for two randomly generated patterns.

3.6. Planning Task

To evaluate autonomous construction, access, and manipulation of compositional data structures, we tested the model using the HTN planning task outlined in Section 2.3.6. First, the controller regions of the model were "flashed" using the store-erase rule with an instruction set and set of program subroutines that implement the task (Katz et al., 2019). During testing, the model was provided with a sequence of inputs encoding a knowledge-base, environmental

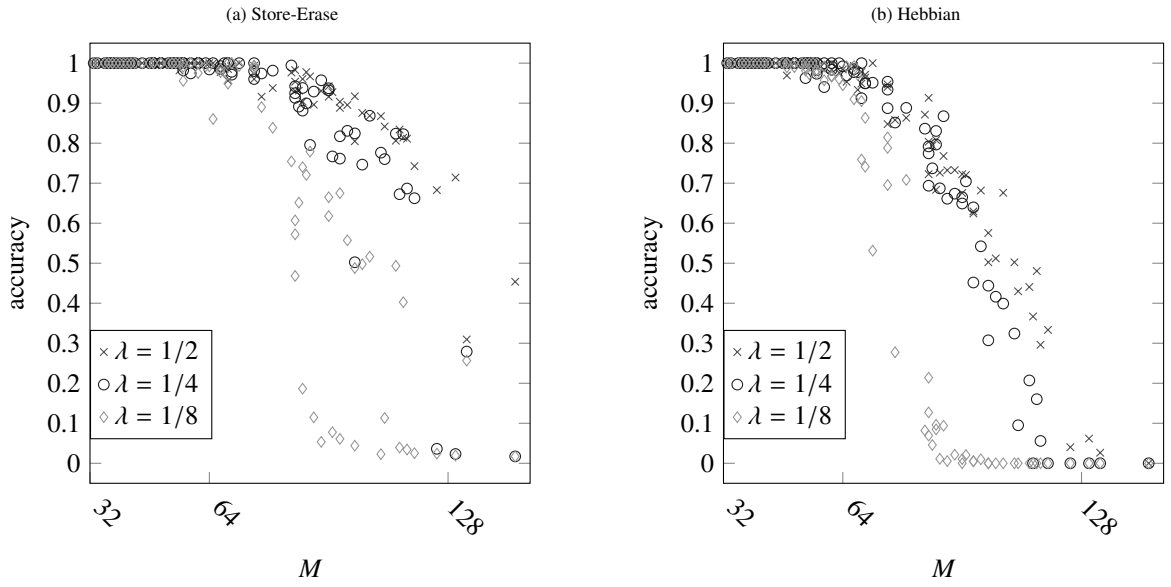

Figure 10: Accuracy of attractor graphs encoding parse trees. Each plot shows the accuracy for an AGN with $N = 1024$ neurons trained with either the fast store-erase learning rule (left) or traditional Hebbian learning (right). The network was trained with attractor graphs representing sentence parse trees drawn from the Penn Treebank. Each node was represented by an attractor in the *mem* region, and contained a symbol represented by a pattern of activity in the *lex* region. Each data point indicates the percentage of *lex* patterns successfully retrieved (y-axis) during traversal of a tree with M nodes (x-axis). Errors in attractor transitions were allowed to propagate during traversal, and an external queue was used to maintain and retrieve intermediate *mem* activation patterns to perform a breadth-first traversal.

bindings, and top-level actions for planning. After decomposing these actions, the model performed a pre-order traversal, printing out the resulting HTN tree.

The model was tested with top-level action sequences that corresponded to HTNs of various sizes. In the *baseline* condition, these tests were performed with a small knowledge-base (9 rules across 5 compound actions) and small environment (4 bindings). To determine the impact of these data structures on planning runtime, two additional conditions were considered: one with a larger knowledge-base (*extended KB*, 15 rules across 7 compound actions), and one with a larger set of environmental bindings (*extended env*, 8 bindings). We ensured that the constructed HTNs did not differ across these conditions.

The size of each network region was set according to the number of patterns to be represented in that region. The *mem* region contained $N_{mem} = 9216$ neurons to ensure successful storage of task-relevant data structures. The context density λ was set to 0.5, and the store-erase rule was used for online learning (Section 2.3.2).

The model successfully performed the task in all cases, and produced the sequence of outputs corresponding to the correct HTN tree. Figure 12 shows the number of timesteps taken during each test (multi-timestep attractor convergence events are collapsed into individual timesteps). These results show that the computational complexity of planning scales linearly with the size of the constructed HTN, independently of the size of the knowledge-base or environmental binding set. This demonstrates that associative arrays represented in attractor graph memory can be efficiently accessed, as lookups require a constant number of timesteps that is independent of the number of learned key-value pairs.

4. Discussion

We have presented a recurrent neural network model that represents compositional data structures as systems of itinerant attractors called *attractor graphs*. Our model learns context-dependent attractor transitions using a novel combination of top-down gating and one-step associative learning. Notably, this training method makes it possible to learn multiple outgoing transitions from a single attractor state using a single hetero-associative matrix. These tran-

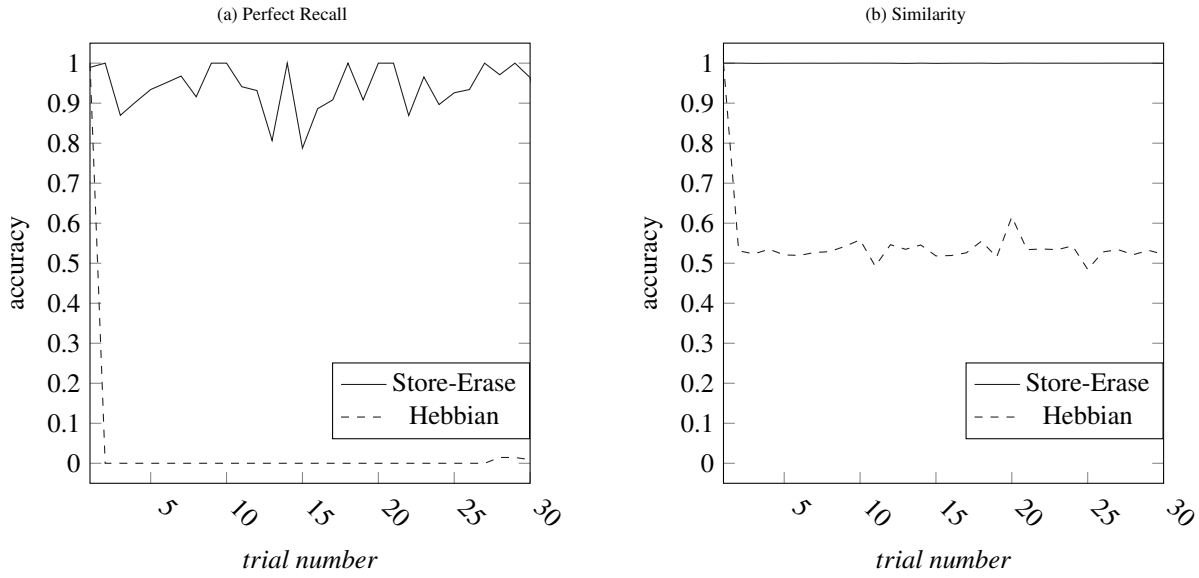

Figure 11: Learning parse trees without weight resets. The plot shows the performance an AGN with $N = 1024$ neurons trained with either the fast store-erase learning rule (solid lines) or traditional Hebbian learning (dashed lines), reported as the average similarity for *lex* activity patterns representing symbols stored in tree nodes. Each trial involved learning a single parse tree, and evaluating symbol recall. Attractor states were recycled between trees, but each tree was represented by a unique set of attractor transitions and inter-regional associations (from *mem* to *ctx* and *lex*). In between trials, the weights of the model were not reset. The size of randomly selected parse trees was limited to 64 nodes to prevent errors in attractor convergence.

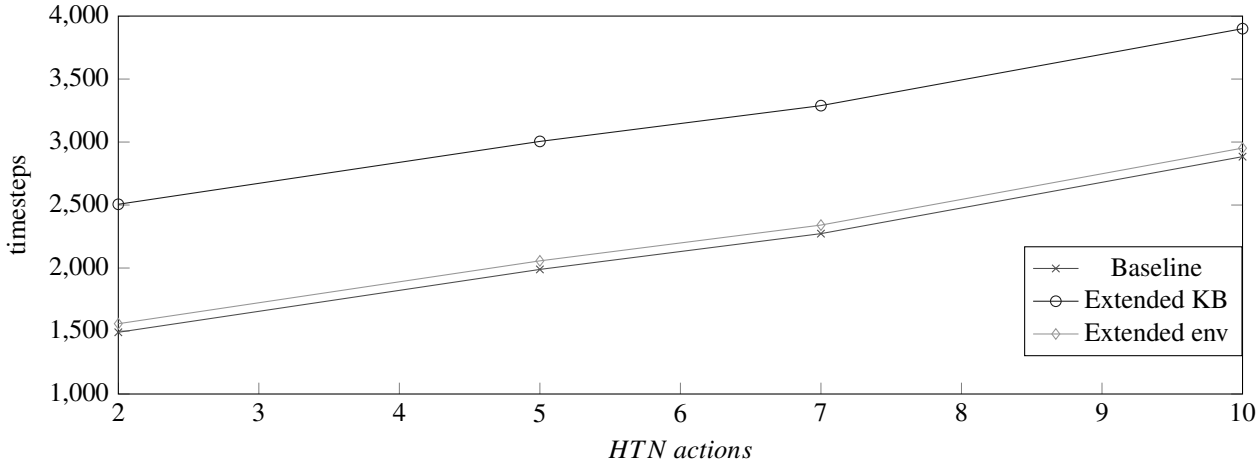

Figure 12: HTN planning runtime. The full programmable neural network was pre-trained to perform the HTN planning task, and evaluated on inputs representing a simple repair task domain (Section 2.3.6). Each line shows the number of timesteps taken to parse the domain knowledge-base and environment bindings, and perform decomposition of a sequence of high-level actions (y-axis). The x-axis indicates the number of actions in the target HTN (including internal and leaf nodes). Three conditions were evaluated. The baseline condition involved a small knowledge-base and environment. The "extended KB" condition involved a knowledge-base that was roughly twice the size of the baseline knowledge-base. The "extended env" condition involved twice the environmental bindings as the baseline condition. The results show that the computational complexity of the planning stage of the task is independent of the size of the knowledge-base and environmental bindings, and depends only on the resulting HTN tree.

sitions are selected during model execution by multiplicative contextual gating signals that control memory retrieval and iteration through learned data structures. We refer to this as *functional branching*, as the branches in attractor sequences are determined by patterns of activity and are not stored in distinct connectivity matrices.

Empirical results demonstrate that our model can reliably store and retrieve attractor graphs representing compositional data structures such as associative arrays, linked lists, and trees. While the number of learned attractor states is limited, our model can learn attractor graphs with large numbers of transitions (edges), and with very high branching factors (vertex degrees), and individual attractors may be used as components of several data structures. We evaluated two forms of one-step associative learning: traditional Hebbian learning and the fast store-erase learning rule (Katz et al., 2019). While the two learning rules yielded similar memory capacities, the store-erase rule significantly outperformed Hebbian learning on attractor graphs with very high branching factors. This reflects a reduction of interference across contexts that is likely due to the anti-Hebbian component of the store-erase rule, which also enables controlled erasure of learned associations. The ability to erase and overwrite transitions permits rapid reorganization of attractor graphs, making the network an effective model of reusable working memory.

We also showed that compositional data structures can be efficiently manipulated via procedural gating control in a programmable neural network. The network successfully performed a hierarchical planning task involving rule-based decomposition of action sequences. A significant limitation of this model is that it does not leverage compositional memory to store programmatic procedures, and instead relies upon a comparably simple assembly-like language with linear program sequences (Katz et al., 2019). Because attractor graphs can represent tree data structures, these processes may instead be represented as abstract syntax trees for programs written in a high level programming language. In addition, a unified program/data memory would allow implementation of homoiconic programming languages such as Lisp and Scheme, making it possible for the model to modify learned programs and synthesize new ones.

Another significant limitation of our model is that it does not identify opportunities to reuse existing memory structures, and does not have a stable long-term memory. As mentioned in the Introduction, the limited capacity of human working memory is offset by the ability to organize and store structured representations that afford access to a broad range of information. We have shown how representations in working memory can be structured or "chunked" according to learned programmatic procedures. The capacity of working memory would be greatly enhanced by a long-term memory containing structures that can be integrated with the contents of working memory. Structures in working memory could then be replaced by pointers to existing long-term memory structures, effectively "chunking" them into compressed units to reduce working memory load.

Our model differs from contemporary machine learning approaches to compositionality in several ways. Most notably, individual representations are fixed-point dynamical attractors learned with one-step associative learning rather than the error-based gradient descent learning methods common in deep learning. These attractor states are composed into complex structures with context-dependent transitions that represent relations between discrete elements in memory. Because these relations are stored in connectivity weights, our model does not rely on persistent maintenance of multiple activity patterns. Instead, memories are retrieved as needed via top-down control of attractor transitions. This "activity-silent" form of working memory has a strong basis in neuroscientific theory (Manohar et al., 2019; Mongillo et al., 2008; Rose et al., 2016; Stokes, 2015; Barbosa et al., 2019), and has not previously been used for compositional learning in artificial neural networks.

AGNs do not require specialized operations for compressing elements into structured representations, such as circular convolutions. Instead of creating summary vectors, AGNs learn direct relations between elements with arbitrary encodings (activity patterns) using one-step associative learning. Because structure is learned in connectivity weights, compositional data structures can be modified without changing any activity state encodings. This is particularly advantageous for nested structures: modifications to a sub-structure do not require modifications to encapsulating structures. For example, a leaf may be added to a tree without modifying the encodings of the leaf's ancestor nodes. In contrast, semantic pointers are semantically related to the content they represent, and cannot be modified without creation of new semantic pointers (Blouw et al., 2016).

Attractor graphs are capable of representing any labeled directed multigraph that does not contain two edges with a shared source node and edge label. This represents a very general class of possible data structures, including associative arrays, linked lists, and trees, but also graphs with cycles that cannot be represented as semantic pointers due to recursive dependencies. This expressive capability exceeds that of other attractor-based models that focus on sequence learning (Rajan et al., 2016; Yamashita and Tani, 2008; Jensen, 2006), or that have architecturally separated

representations of each hierarchical level, as in Dynamic Field Theory (Durán et al., 2012). In our model, attractor graphs in the *mem* region can represent arbitrarily nested hierarchical structures without distinct regions for each level of the hierarchy.

Acknowledgements

This work was supported by ONR award N00014-19-1-2044.

Appendix A. Planning Task Domain

The hierarchical planning task described in Section 2.3.6 involves rule-based decomposition of sequential behaviors according to environmental conditions. Here we provide the decomposition rules, environmental bindings, and sequences of top-level actions used for the experiments in Section 3.6.

Rules in the knowledge-base specify how compound actions can be decomposed into sequences of sub-actions according to properties of the environment. For example, opening a door involves different actions that depend on the type of door being opened (e.g., a sliding door is opened by grasping the handle, sliding the door open, and releasing the handle). We use the following notation to express these rules:

$$compound\_action(env\_value) = \begin{cases} (sub\_action_1^a, sub\_action_2^a, ...) & \text{if } env\_val = val^a \\ (sub\_action_1^b, sub\_action_2^b, ...) & \text{if } env\_val = val^b \\ ... \end{cases}$$

where *compound_action* is the action to be decomposed, *env_value* is the value of the environmental binding that determines the applicable decomposition rule, $(sub\_action_1^a, sub\_action_2^a, ...)$ is a sequence of sub-actions for rule a, and val^a is the required environmental binding value to apply rule a. The full set of rules is enumerated below. Rules marked with a ⋆ are only learned in the "extended knowledge-base" condition in Section 3.6, and are not used during decomposition of the top-level sequences that were tested.

$$open\_door(door\_type) = \begin{cases} (grasp, slide\_open, release) & \text{if } door\_type = sliding \\ (enter\_passcode, enter\_open) & \text{if } door\_type = electronic \\ \star\ (unlatch, grasp, pull\_open, release) & \text{if } door\_type = latch \end{cases}$$

$$close\_door(door\_type) = \begin{cases} (grasp, slide\_closed, release) & \text{if } door\_type = sliding \\ (enter\_close) & \text{if } door\_type = electronic \\ \star\ (grasp, push\_closed, release, latch) & \text{if } door\_type = latch \end{cases}$$

$$check\_component(indicator) = \begin{cases} (check\_led) & \text{if } indicator = led \\ \star\ (check\_pressure\_gauge) & \text{if } indicator = pressure\_gauge \\ \star\ (check\_display) & \text{if } indicator = display \end{cases}$$

$$check\_led(led\_color) = \begin{cases} (report\_working) & \text{if } led\_color = green \\ (repair\_component) & \text{if } led\_color = yellow \\ \star\ (report\_broken) & \text{if } led\_color = red \end{cases}$$

$$check\_display(display\_reading) = \begin{cases} \star\ (report\_working) & \text{if } display\_reading = ok \\ \star\ (repair\_component) & \text{if } display\_reading = warning \\ \star\ (report\_broken) & \text{if } display\_reading = error \end{cases}$$

$$check\_pressure\_gauge(pressure\_reading) = \begin{cases} \star\ (report\_working) & \text{if } pressure\_reading = high \\ \star\ (repair\_component) & \text{if } pressure\_reading = medium \\ \star\ (report\_broken) & \text{if } pressure\_reading = low \end{cases}$$

$$repair\_component(interaction\_point) = \begin{cases} (flip\_switch) & \text{if } interaction\_point = switch \\ (press\_button) & \text{if } interaction\_point = button \\ \star\ (screw\_valve) & \text{if } interaction\_point = valve \end{cases}$$

Environmental bindings are stored in an associative array, and each binding takes the form of a simple key-value pair. The full environment is listed below. Items marked with a $\star$ are only learned in the "extended environment" condition in Section 3.6, and are not accessed during decomposition of the test top-level sequences.

$$\begin{aligned} door\_type &: electronic \\ indicator &: led \\ led\_color &: yellow \\ interaction\_point &: button \\ \star\ weather &: cloudy \\ \star\ pants &: jeans \\ \star\ time &: evening \\ \star\ mood &: tired \end{aligned}$$

Four different top-level action sequences were used for testing. These sequences are listed below, along with the total number of actions contained in the resulting HTN (including internal and leaf nodes):

$$\begin{aligned} &2\ actions : (repair\_component) \\ &5\ actions : (open\_door, close\_door) \\ &7\ actions : (open\_door, press\_button, report\_repaired, close\_door) \\ &10\ actions : (open\_door, check\_component, close\_door) \end{aligned}$$

References

Aizawa, K., 2003. The productivity of thought, in: The Systematicity Arguments. Springer, pp. 43–55.

Andreas, J., Rohrbach, M., Darrell, T., Klein, D., 2016. Neural module networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 39–48.

Ba, J., Hinton, G.E., Mnih, V., Leibo, J.Z., Ionescu, C., 2016. Using fast weights to attend to the recent past, in: Advances in Neural Information Processing Systems, pp. 4331–4339.

Baan, J., Leible, J., Nikolaus, M., Rau, D., Ulmer, D., Baumgärtner, T., Hupkes, D., Bruni, E., 2019. On the realization of compositionality in neural networks, in: Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP, pp. 127–137.

Baars, B.J., 2007. The global workspace theory of consciousness. The Blackwell Companion to Consciousness , 236–246.

Baars, B.J., Franklin, S., 2003. How conscious experience and working memory interact. Trends in Cognitive Sciences 7, 166–172.

Baayen, R.H., 1994. Productivity in language production. Language and Cognitive Processes 9, 447–469.

Baddeley, A., 1993. Working memory and conscious awareness. Theories of Memory , 11–28.

Bahdanau, D., Cho, K., Bengio, Y., 2014. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473 .

Barbosa, J., Stein, H., Martinez, R., Galan, A., Adam, K., Li, S., Valls-Solé, J., Constantinidis, C., Compte, A., 2019. Interplay between persistent activity and activity-silent dynamics in prefrontal cortex during working memory. bioRxiv , 763938.

Barsalou, L.W., 1993. Flexibility, structure, and linguistic vagory in concepts: Manifestations of compositional system of perceptual symbols. Theories of Memory , 29.

Besold, T.R., Garcez, A.d., Bader, S., Bowman, H., Domingos, P., Hitzler, P., Kühnberger, K.U., Lamb, L.C., Lowd, D., Lima, P.M.V., et al., 2015. Neural-symbolic learning and reasoning: A survey and interpretation, in: Knowledge Representation and Reasoning: Integrating Symbolic and Neural Approaches: Papers from the 2015 AAAI Spring Symposium.

Bienenstock, E., Geman, S., Potter, D., 1997. Compositionality, MDL priors, and object recognition, in: Advances in Neural Information Processing Systems, pp. 838–844.

Blouw, P., Solodkin, E., Thagard, P., Eliasmith, C., 2016. Concepts as semantic pointers: A framework and computational model. Cognitive Science 40, 1128–1162.

Borisyuk, R., Chik, D., Kazanovich, Y., da Silva Gomes, J., 2013. Spiking neural network model for memorizing sequences with forward and backward recall. Biosystems 112, 214–223.

Botvinick, M.M., 2008. Hierarchical models of behavior and prefrontal function. Trends in Cognitive Sciences 12, 201–208.

Bunel, R., Hausknecht, M., Devlin, J., Singh, R., Kohli, P., 2018. Leveraging grammar and reinforcement learning for neural program synthesis. arXiv preprint arXiv:1805.04276 .

Burke, M., Penkov, S., Ramamoorthy, S., 2019. From explanation to synthesis: Compositional program induction for learning from demonstration. Robotics: Science and Systems XV doi:10.15607/RSS.2019.XV.015.

Campitelli, G., Gobet, F., Head, K., Buckley, M., Parker, A., 2007. Brain localization of memory chunks in chessplayers. International Journal of Neuroscience 117, 1641–1659.

Chabuk, T., Reggia, J.A., 2013. The added value of gating in evolved neurocontrollers, in: The 2013 International Joint Conference on Neural Networks (IJCNN), pp. 1–8. doi:10.1109/IJCNN.2013.6706895.

Chung, J., Gulcehre, C., Cho, K., Bengio, Y., 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555 .

Colom, R., Rebollo, I., Palacios, A., Juan-Espinosa, M., Kyllonen, P.C., 2004. Working memory is (almost) perfectly predicted by g. Intelligence 32, 277–296.

Conway, A.R., Kane, M.J., Engle, R.W., 2003. Working memory capacity and its relation to general intelligence. Trends in Cognitive Sciences 7, 547–552.

Cowan, N., 2010. The magical mystery four: How is working memory capacity limited, and why? Current Directions in Psychological Science 19, 51–57.

Danihelka, I., Wayne, G., Uria, B., Kalchbrenner, N., Graves, A., 2016. Associative long short-term memory, in: International Conference on Machine Learning, pp. 1986–1994.

Dehaene, S., Changeux, J.P., 1997. A hierarchical neuronal network for planning behavior. Proceedings of the National Academy of Sciences 94, 13293–13298.

Durán, B., Sandamirskaya, Y., Schöner, G., 2012. A dynamic field architecture for the generation of hierarchically organized sequences, in: International Conference on Artificial Neural Networks, Springer. pp. 25–32.

Eliasmith, C., Stewart, T.C., Choo, X., Bekolay, T., DeWolf, T., Tang, Y., Rasmussen, D., 2012. A large-scale model of the functioning brain. Science 338, 1202–1205.

Erlhagen, W., Schöner, G., 2002. Dynamic field theory of movement preparation. Psychological Review 109, 545.

Erol, K., 1996. Hierarchical task network planning: formalization, analysis, and implementation. Ph.D. thesis.

Fodor, J.A., Pylyshyn, Z.W., 1988. Connectionism and cognitive architecture: A critical analysis. Cognition 28, 3–71. doi:10.1016/0010-0277(88)90031-5.

Gayler, R.W., 2003. Vector symbolic architectures answer jackendoff's challenges for cognitive neuroscience, in: International Conference on Cognitive Science, Citeseer.

Ghallab, M., Nau, D., Traverso, P., 2004. Automated Planning: Theory and Practice. Elsevier.

Graves, A., Wayne, G., Danihelka, I., 2014. Neural turing machines. arXiv preprint arXiv:1410.5401 .

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S.G., Grefenstette, E., Ramalho, T., Agapiou, J., et al., 2016. Hybrid computing using a neural network with dynamic external memory. Nature 538, 471–476.

Hauge, T.C., Katz, G.E., Davis, G.P., Huang, D.W., Reggia, J.A., Gentili, R.J., 2020. High-level motor planning assessment during performance of complex action sequences in humans and a humanoid robot. International Journal of Social Robotics , 1–18.

Hauge, T.C., Katz, G.E., Davis, G.P., Jaquess, K.J., Reinhard, M.J., Costanzo, M.E., Reggia, J.A., Gentili, R.J., 2019. A novel application of levenshtein distance for assessment of high-level motor planning underlying performance during learning of complex motor sequences. Journal of Motor Learning and Development 1, 1–20.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Computation 9, 1735–1780.

Hoshino, O., Usuba, N., Kashimori, Y., Kambara, T., 1997. Role of itinerancy among attractors as dynamical map in distributed coding scheme. Neural Networks 10, 1375–1390.

Hupkes, D., Dankers, V., Mul, M., Bruni, E., 2020. Compositionality decomposed: How do neural networks generalise? Journal of Artificial Intelligence Research 67, 757–795.

Hupkes, D., Singh, A., Korrel, K., Kruszewski, G., Bruni, E., 2018. Learning compositionally through attentive guidance. arXiv preprint arXiv:1805.09657 .

Jaeggi, S.M., Buschkuehl, M., Jonides, J., Perrig, W.J., 2008. Improving fluid intelligence with training on working memory. Proceedings of the National Academy of Sciences 105, 6829–6833.

Jensen, O., 2006. Maintenance of multiple working memory items by temporal segmentation. Neuroscience 139, 237–249.

Kalyan, A., Mohta, A., Polozov, O., Batra, D., Jain, P., Gulwani, S., 2018. Neural-guided deductive search for real-time program synthesis from examples. arXiv preprint arXiv:1804.01186 .

Kamp, H., Partee, B., 1995. Prototype theory and compositionality. Cognition 57, 129–191.

Katz, G.E., Davis, G.P., Gentili, R.J., Reggia, J.A., 2019. A programmable neural virtual machine based on a fast store-erase learning rule. Neural Networks 119, 10–30.

Kipf, T., Li, Y., Dai, H., Zambaldi, V., Sanchez-Gonzalez, A., Grefenstette, E., Kohli, P., Battaglia, P., 2019. Compile: Compositional imitation learning and execution, in: International Conference on Machine Learning (ICML), pp. 3418–3428.

Lake, B., Baroni, M., 2018. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks, in: International Conference on Machine Learning, pp. 2873–2882.

Lake, B.M., 2019. Compositional generalization through meta sequence-to-sequence learning, in: Advances in Neural Information Processing Systems, pp. 9788–9798.

Lake, B.M., Ullman, T.D., Tenenbaum, J.B., Gershman, S.J., 2017. Building machines that learn and think like people. Behavioral and Brain Sciences 40.

Loula, J., Baroni, M., Lake, B.M., 2018. Rearranging the familiar: Testing compositional generalization in recurrent networks. arXiv preprint arXiv:1807.07545 .

Manohar, S.G., Zokaei, N., Fallon, S.J., Vogels, T., Husain, M., 2019. Neural mechanisms of attending to items in working memory. Neuroscience

& Biobehavioral Reviews .
Marcus, G., 2018. Deep Learning: A Critical Appraisal. arXiv:1801.00631 .
Marcus, G., 2020. The next decade in ai: four steps towards robust artificial intelligence. arXiv preprint arXiv:2002.06177 .
Masse, N.Y., Grant, G.D., Freedman, D.J., 2018. Alleviating catastrophic forgetting using context-dependent gating and synaptic stabilization. Proceedings of the National Academy of Sciences 115, E10467–E10475.
Miconi, T., Stanley, K., Clune, J., 2018. Differentiable plasticity: training plastic neural networks with backpropagation, in: International Conference on Machine Learning, pp. 3559–3568.
Miller, P., 2016. Itinerancy between attractor states in neural systems. Current Opinion in Neurobiology 40, 14–22.
Mongillo, G., Barak, O., Tsodyks, M., 2008. Synaptic theory of working memory. Science 319, 1543–1546.
Nefdt, R.M., 2020. A puzzle concerning compositionality in machines. Minds and Machines , 1–29.
Oberauer, K., 2009. Design for a working memory. Psychology of Learning and Motivation 51, 45–100.
Pelletier, F.J., 1994. The principle of semantic compositionality. Topoi 13, 11–24.
Pham, T., Tran, T., Venkatesh, S., 2018. Graph memory networks for molecular activity prediction, in: 2018 24th International Conference on Pattern Recognition (ICPR), IEEE. pp. 639–644.
Piantadosi, S.T., Tenenbaum, J.B., Goodman, N.D., 2016. The logical primitives of thought: Empirical foundations for compositional cognitive models. Psychological Review 123, 392.
Plate, T.A., 1995. Holographic reduced representations. IEEE Transactions on Neural networks 6, 623–641.
Rajan, K., Harvey, C.D., Tank, D.W., 2016. Recurrent network models of sequence generation and memory. Neuron 90, 128–142.
Reggia, J.A., Katz, G.E., Davis, G.P., 2019. Modeling working memory to identify computational correlates of consciousness. Open Philosophy 2, 252–269.
Reggia, J.A., Katz, G.E., Davis, G.P., 2020. Artificial conscious intelligence. Journal of Artificial Intelligence and Consciousness 7, 95–107.
Reverberi, C., Görgen, K., Haynes, J.D., 2012. Compositionality of rule representations in human prefrontal cortex. Cerebral Cortex 22, 1237–1246.
Riesenhuber, M., Poggio, T., 1999. Hierarchical models of object recognition in cortex. Nature Neuroscience 2, 1019–1025.
Rikhye, R.V., Gilra, A., Halassa, M.M., 2018. Thalamic regulation of switching between cortical representations enables cognitive flexibility. Nature Neuroscience 21, 1753–1763.
Rose, N.S., LaRocque, J.J., Riggall, A.C., Gosseries, O., Starrett, M.J., Meyering, E.E., Postle, B.R., 2016. Reactivation of latent working memories with transcranial magnetic stimulation. Science 354, 1136–1139.
Sandamirskaya, Y., Zibner, S.K., Schneegans, S., Schöner, G., 2013. Using dynamic field theory to extend the embodiment stance toward higher cognition. New Ideas in Psychology 31, 322–339.
Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al., 2016. Mastering the game of go with deep neural networks and tree search. Nature 529, 484–489.
Stewart, T.C., Bekolay, T., Eliasmith, C., 2011. Neural representations of compositional structures: Representing and manipulating vector spaces with spiking neurons. Connection Science 23, 145–153.
Stokes, M.G., 2015. 'activity-silent'working memory in prefrontal cortex: a dynamic coding framework. Trends in Cognitive Sciences 19, 394–405.
Sukhbaatar, S., Weston, J., Fergus, R., et al., 2015. End-to-end memory networks, in: Advances in Neural Information Processing Systems, pp. 2440–2448.
Sylvester, J., Reggia, J., 2016. Engineering neural systems for high-level problem solving. Neural Networks 79, 37–52. doi:10.1016/j.neunet.2016.03.006.
Sylvester, J., Reggia, J., Weems, S., Bunting, M., 2013. Controlling working memory with learned instructions. Neural Networks 41, 23–38.
Szabó, Z., 2012. The case for compositionality. The Oxford Handbook of Compositionality 64, 80.
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I., 2017. Attention is all you need, in: Advances in Neural Information Processing Systems, pp. 5998–6008.
Vecoven, N., Ernst, D., Wehenkel, A., Drion, G., 2020. Introducing neuromodulation in deep neural networks to learn adaptive behaviours. PloS One 15, e0227922.
Van der Velde, F., Kamps, M.d., 2006. Neural blackboard architectures of combinatorial structures in cognition. Behavioral and Brain Sciences 29, 37–70.
Witkin, A.P., Tenenbaum, J.M., 1983. On the role of structure in vision, in: Human and Machine Vision. Elsevier, pp. 481–543.
Yamashita, Y., Tani, J., 2008. Emergence of functional hierarchy in a multiple timescale neural network model: a humanoid robot experiment. PLoS Comput Biol 4, e1000220.