ABSTRACT

Title of Dissertation: A NEUROCOMPUTATIONAL MODEL OF CAUSAL REASONING AND COMPOSITIONAL WORKING MEMORY FOR IMITATION LEARNING

Gregory Patrick Davis
Doctor of Philosophy, 2022

Dissertation Directed by: Professor James A. Reggia
Department of Computer Science

Although contemporary neural models excel in a surprisingly diverse range of application domains, they struggle to capture several key qualities of human cognition that are considered crucial for human-level artificial intelligence (AI). Some of these qualities, such as compositionality and interpretability, are readily achieved with traditional symbolic programming, leading some researchers to suggest hybrid neuro-symbolic programming as a viable route to human-level AI. However, the cognitive capabilities of biological nervous systems indicate that it should be possible to achieve human-level reasoning in artificial neural networks without the support of non-neural symbolic algorithms. Furthermore, the computational explanatory gap between cognitive and neural algorithms is a major obstacle to understanding the neural basis of cognition, an endeavor that is mutually beneficial to researchers in AI, neuroscience, and cognitive science. A viable approach to bridging this gap involves "programmable neural networks" that learn to store and evaluate symbolic expressions directly in neural memory, such as the recently developed "Neural Virtual

Machine" (NVM). While the NVM achieves Turing-complete universal neural programming, its assembly-like programming language makes it difficult to express the complex algorithms and data structures that are common in symbolic AI, limiting its ability to learn human-level cognitive procedures.

This dissertation presents an approach to high-level neural programming that supports human-like reasoning using only biologically-plausible neural computations. First, I introduce a neural model that represents graph-based data structures as systems of dynamical attractor states called *attractor graphs*. This model serves as a temporally-local compositional working memory that can be controlled via top-down neural gating. Then, I present a programmable neural network called *NeuroLISP* that learns an interpreter for a subset of Common LISP. NeuroLISP features native support for compositional data structures, scoped variable binding, and a shared memory space in which programs can be modified as data. Empirical experiments demonstrate that NeuroLISP can learn algorithms for multiway tree processing, compositional sequence manipulation, and symbolic unification in first-order logic. Finally, I present *NeuroCERIL*, a neural model that performs hierarchical causal reasoning for robotic imitation learning and successfully learns a battery of procedural maintenance tasks from human demonstrations. NeuroCERIL implements a cognitively-plausible and computationally-efficient algorithm for hypothetico-deductive reasoning, which combines bottom-up abductive inference with top-down predictive verification. Because the hypothetico-deductive approach is broadly relevant to a variety of cognitive domains, including problem-solving and diagnostic reasoning, NeuroCERIL is a significant step toward human-level cognition in neural networks.

A NEUROCOMPUTATIONAL MODEL OF CAUSAL REASONING AND COMPOSITIONAL WORKING MEMORY FOR IMITATION LEARNING

by

Gregory Patrick Davis

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2022

Advisory Committee:
Professor James A. Reggia, Chair
Professor Rodolphe J. Gentili
Professor Yiannis Aloimonos
Professor Dana S. Nau
Professor Donald Perlis

© Copyright by
Gregory Patrick Davis
2022

Mystery of life

May we one day see you shine

Through eyes of machines

Acknowledgments

Six years ago I boarded a train and moved across the country to begin my journey as a graduate student. Thanks to my lifelong friend Joey for accompanying me on that trip. Isn't it remarkable how far we've both come?

Long before then, countless individuals fostered my growth and encouraged me to realize my potential. Thanks to all the teachers and professors I've had over the years who pushed me to excel in my studies. Most of all, thanks to my parents and brothers for the love and support. I'm blessed to have you as my family.

Since arriving in Maryland, I've met several intelligent and considerate friends that I'm grateful for. Thanks especially to Matt, Ryen, and Felix for being close companions and peers that I can both brainstorm and commiserate with. Thanks also to James, Carley, Liz, Mike, Polina, Jon, Lauren, Kevin, Z, and Roo; it's been a pleasure spending time with you and staying up irresponsibly late playing board games and drinking whiskey.

I've been incredibly fortunate to participate in a research group with kind and effective mentors. Jim, Garrett, and Rodolphe, thank you very much for providing me with a healthy environment in which I could grow as a scientist, and for tolerating my sometimes outlandish ideas so they could be shaped into productive research. Jim, thank you for being an encouraging yet critical advisor, and for having such high standards of respect and humility. You are an excellent role model that I hope to emulate one day with my own students.

Finally, thanks to ONR (award N00014-19-1-2044), UMD COMBINE program (NSF Award DGE1632976), and the UMD Graduate School (Ann G. Wylie Dissertation Fellowship) for financial support.

Table of Contents

List of Tables

List of Figures

1

Introduction

Recent advancements in machine learning have renewed interest in artificial neural networks, propelling neurocomputational methods such as "deep learning" to the forefront of artificial intelligence (AI) research. Despite remarkable progress in areas as diverse as natural language production and visual scene understanding, deep neural networks struggle to capture several important hallmarks of human cognition that are recognized as key ingredients for human-level AI [113, 124, 144]. Furthermore, concerns have been raised about their opacity and dependence on massive data sets and computing resources [37, 123, 138, 186], both of which are also at odds with the efficiency and explainability of human learning and reasoning.

Several recently proposed strategies for developing human-level AI share a number of common elements, including a focus on causal reasoning, compositionality, meta-learning, and internal modeling of the physical environment and the mentality of other agents [113, 124, 144]. Many of these qualities are more readily expressed in symbolic algorithms than artificial neural networks, motivating proposals for further work in hybrid neuro-symbolic models that integrate neural and non-neural components, such as neural-guided search algorithms [5, 23, 32, 34, 92, 103, 124, 179]. Although there is a long-standing debate in the AI community about the relative merits of symbolic and sub-symbolic methods [61], the recent successes of hybrid systems demonstrate

the complementarity of both approaches by leveraging their unique benefits: neural networks offer their capacity for learning and generalization in exchange for the reliable and interpretable algorithmic control provided by traditional symbolic computations.

The difficulty of capturing human-level cognitive control in artificial neural networks is somewhat puzzling given the cognitive capabilities of biological nervous systems. This discrepancy illustrates a *computational explanatory gap* between cognitive and neural computations that impedes development of human-level AI and stifles efforts to understand the neural basis of cognition [157, 159]. Although hybrid systems are reasonable and effective from an engineering standpoint, they fail to address this issue because they do not explain how symbolic procedures can be implemented in neural networks. Bridging the computational explanatory gap requires the development of purely neural architectures capable of carrying out the high-level cognitive control that is characteristic of human intelligence.

One possible solution to this problem is to encode symbolic information as dynamical attractors in recurrent neural networks [3, 169, 210]. This approach allows purely neurocomputational systems to reliably learn cognitive procedures such as top-down control of working memory, attentional direction, and motor engagement [191, 193]. Although their current domain of application merely scratches the surface of human-level cognition, recent work on universal neural programming demonstrates that attractor neural networks can capture the full breadth of computational capabilities spanned by symbolic programming methods [97], making them particularly promising for further development.

This dissertation is motivated by the hypothesis that attractor neural networks can capture key qualities of human cognition that exceed the capabilities of existing neurocomputational systems. Specifically, I consider the following features:

- *Compositionality* refers to the storage and manipulation of structured representations made up of reusable parts [82, 111, 119, 133]. This ability promotes generalization and underpins the productivity of thought that is evident in domains as diverse as language comprehension [2, 12, 145], behavioral planning and imitation [12, 31, 164], visual perception [25, 166, 206], and concept learning [21, 93, 148].

- *Causal reasoning* refers to reasoning about logical relationships between causes and their consequential effects, which may be chained together into causal networks that describe complex situations [37, 113, 123, 144, 177]. This is the case in many real-world problem-solving domains, including scientific inquiry, which requires the construction and manipulation of causal explanations that yield experimentally verifiable predictions.

- *Interpretability* means that outside observers can make sense of an agent's behavior, allowing them to diagnose and correct mistakes [36, 68, 117, 118, 165]. One way to achieve interpretability in machine learning is to build models that provide meaningful explanations that justify their actions.

- *Meta-learning*, or *learning to learn*, involves learning general-purpose skills that accelerate learning across domains [79, 199, 204]. This typically involves generalizing knowledge so that it can be recycled across tasks and applied to novel situations.

- *Intuitive physics and psychology* refers to the capacity for reasoning about the dynamics of physical objects and mental states such as goals and beliefs [51, 109, 113]. This is especially critical for robotic systems that interact with the physical environment and collaborate with humans and/or other machines.

To explore this hypothesis, I consider the domain of imitation learning, in which new skills are learned by observing demonstrations. The ability to imitate emerges in humans at an early age and plays a crucial role in early cognitive development, but remains a natural and intuitive method for acquiring new skills throughout the lifespan [90, 129]. Importantly, human-level imitation involves not only replicating observable motor behavior, but also inferring the underlying goals and causal intentions of the demonstrator and relating them to changes in the physical environment. This allows learners to generalize demonstrated skills to novel environments by abstracting away details that are circumstantial to the demonstration.

Robotic imitation learning has been proposed as a solution to the complexity and limited accessibility of robotic programming [26, 83, 155, 176]. Despite recent progress, it remains difficult to develop systems that generalize well to novel circumstances and adapt learned behavior to situations that deviate from the demonstration environment. Furthermore, contemporary robotic imitation learning systems suffer from the limitations of deep learning, including a lack of interpretability, which makes it difficult to diagnose and debug errors and creates barriers in trustworthiness and explainability. Safe and effective robotic imitation learning requires human-level algorithms for understanding demonstrated actions, adapting learned skills to novel environments, and constructing explanations of planned behavior that can be understood by end-users.

A recently developed causal imitation learning system called CERIL learns by inferring the underlying intentions of the demonstrator and generalizing them to novel environments [95]. CERIL reasons about the causal implications of plausible intentions to construct an interpretable hierarchical explanation that is consistent with observed behavior and its environmental consequences. While effective and provably correct, CERIL's algorithms are implemented with traditional non-neural symbolic programming and have a limited degree of cognitive plausibility.

Specifically, CERIL's inference algorithm involves exhaustive enumeration of plausible causal explanations, which places unrealistic demands on working memory. It also processes demonstrations in an offline fashion, rather than iteratively as humans do. However, CERIL provides useful target behavior for a neurocomputational model of compositional causal reasoning.

In this dissertation, I present NeuroCERIL, a purely neural system capable of compositional causal reasoning for robotic imitation learning. Unlike deep neural networks, NeuroCERIL is composed of attractor neural networks that learn to produce interpretable explanations using fast local learning rules. NeuroCERIL captures CERIL's ability to perform hierarchical causal inference, but does so using a novel hypothetico-deductive algorithm that combines bottom-up abductive inference with top-down predictive verification, which more closely resembles human problem-solving. This is made possible by novel methods that I developed for representing compositional data structures and learning high-level algorithmic procedures in attractor neural networks. More specifically, this work was guided by the following three objectives:

1. Develop a neurocomputational model of compositional working memory that can be controlled by internal cognitively-directed signals. This model should be capable of storing hierarchical behavioral plans and environmental models for causal inference during imitation learning. In addition, it should be based on biologically-plausible neural processes such as attractor dynamics, local learning, and multiplicative gating.

2. Develop a programmable attractor neural network that can learn algorithms expressed in a high-level programming language. This network should make use of the compositional working memory described above, and be suitable for implementing the cognitive procedures necessary for causal imitation learning.

3. Develop a cognitively-plausible causal reasoning algorithm for intentional inference during imitation learning that parallels the functionality of CERIL. This algorithm should be implemented in the programmable neural network described above, and achieve computational efficiency to respect the constraints of human working memory.

The remainder of this dissertation is organized as follows.

Chapter 2 surveys relevant prior research. Section 2.1 covers robotic imitation learning and details the functionality of CERIL, which provides the target functionality for NeuroCERIL, the model developed in this dissertation. Section 2.2 covers neural network models of algorithmic learning, including a recently developed Neural Virtual Machine capable of universal neural programming.

Chapter 3 presents a neural model of working memory that represents compositional data structures as systems of itinerant attractors called *attractor graphs*. This model uses a novel combination of multiplicative gating and asymmetric associative learning to establish transitions between attractors that depend on contextual gating signals. This makes it possible to encode a general class of data structures based on labeled directed multigraphs, including linked lists, trees, and associative arrays. Empirical experiments evaluate the model's memory capacity and show that it can be effectively controlled by a programmable neural network that implements a basic hierarchical planning algorithm, satisfying the first objective described above.

Chapter 4 presents *NeuroLISP*, a programmable neural network that can represent and evaluate programs written in a subset of Common LISP, a high-level programming language with an extensive history in AI research. NeuroLISP uses attractor graphs to store nested program expressions, and implements several high-level programming constructs such as scoped variable binding,

recursion, and the ability to manipulate programs as data. Empirical results demonstrate that NeuroLISP can learn algorithms for multiway tree traversal, compositional sequence manipulation, and symbolic pattern matching, and thus satisfies the second objective.

Chapter 5 presents *NeuroCERIL*, a purely neural system that implements a novel hypothetico-deductive algorithm for hierarchical causal inference during imitation learning. NeuroCERIL's architecture extends NeuroLISP to support classes and exception handling, which ease the implementation of complex data structures and algorithms. NeuroCERIL successfully learned the same battery of procedural tasks that was used to validate CERIL, demonstrating that it successfully reproduces CERIL's capacity for causal inference. Additional analysis of NeuroCERIL's memory usage demonstrates its computational efficiency, supporting its plausibility as a model of human-level reasoning and satisfying the third and final objective of this dissertation.

Chapter 6 concludes with a brief discussion of NeuroCERIL's implications for AI research, as well as its limitations and directions for future work.

2

Background

This section reviews some past research that is relevant to the work presented in this dissertation. Section 2.1 provides an overview of imitation learning and details on CERIL, an imitation learning system based on cause-effect reasoning. Section 2.2 describes neurocomputational methods for algorithmic learning, highlighting an architecture for universal neural programming based on gated itinerant attractor dynamics.

2.1 Imitation Learning

Humans readily teach and learn using demonstration and imitation. Although its origin remains unclear, cognitive-level imitation emerges in early childhood and plays a crucial role in human cognitive development, but remains a natural and intuitive method for acquiring new skills throughout the lifespan [15, 17, 90, 128, 195]. Imitation is thought to be supported by neural mechanisms that establish shared representations for perceptually observable behavior and cognitive-motor control processes (i.e., the mirror neuron system), facilitating perspective-taking and interpersonal collaboration [62, 85, 107, 140].

Imitation can take on a variety of forms depending on the degree to which the imitator understands the demonstrated behavior. The simplest form of imitation involves directly repli-

cating the motor trajectories of the teacher, and requires only a superficial understanding that is relatively inflexible. However, humans are capable of a higher form of imitation that involves reasoning about the demonstrator's goals and intentions. This form of "cognitive-level" imitation allows an imitator to grasp the underlying purpose of the demonstration and isolate factors that are essential to the task from those that are circumstantial to the demonstration environment. As a result, the imitator can generalize the learned skill to perform the task in a novel environment. Cognitive-level imitation is therefore far more flexible and adaptive than lower-level imitation that focuses on concrete motor actions rather than than abstract goals and intentions.

Programming robots to carry out complex tasks in a human-like fashion is difficult and typically requires laborious programming by an experienced roboticist. One promising solution to this problem is to develop robots that learn from demonstrations (i.e., robotic imitation learning) [26, 83, 155, 176]. This eases the burden of teaching robots and makes it accessible to domain experts without experience in robotic programming. Despite recent progress, it remains difficult to develop systems that generalize well to novel circumstances and adapt learned behavior to situations that deviate from the demonstration environment [27, 139]. Much past work has focused on low-level tasks such as manual coordination [20, 59, 207] and mobile navigation in simple environments [1, 6, 49]. Because these models primarily address low-level motor control, they do not generalize well to novel environments. This requires higher-level cognitive abilities such as planning and reasoning that are qualitatively more complex than motor reproduction.

Symbolic approaches to cognitive-level imitation learning incorporate explicit planning procedures that operate on symbolic models of goals and behaviors and relate them to changes in the environment [40, 50, 87]. However, these models have mostly employed domain-specific procedures and knowledge for constrainted tasks such such as manipulation of objects on two-

dimensional surfaces and virtual grid-worlds. In addition, they often produce plans with limited hierarchical structure, and do not address unbounded composition of structured knowledge.

Another approach to imitation learning is reinforcement learning, in which agents learn policies for selecting behaviors to reach reward states. This is most effective in models that introduce hierarchical structure for planning at multiple timescales [64, 108, 116]. However, learning policies that generalize well requires significant training data that captures environmental variability, and traditional reinforcement learning requires an explicit reward signal that reflects the demonstrator's goals. When a reward signal is not available, the imitator must infer the goals directly from observed behavior. This is done in inverse reinforcement learning models, which infer a cost function that the demonstrator might be using to plan behavior [76, 120, 139]. While this eliminates the need for explicit reward signals, these models still require substantial training, and they typically have limited or no hierarchical structure.

When demonstrations are treated as execution traces of "behavioral programs", imitation learning can be approached as a problem of program synthesis or induction. Because algorithmic procedures can be expressed as hierarchies of functional routines, this approach naturally addresses the compositionality of intentional behavior. Although there is significant prior work using symbolic methods [72], recent work has shown some success with machine learning models. For example, Sun et al. [189] present a neural achitecture that is trained end-to-end using target programs as training labels. This architecture contains an internal "summarizer" modulate that aggregates encodings of multiple demonstrations into a compact representation that is used to sequentially generate an output program. Similarly, Xu et al. [208] present a Neural Task Programming (NTP) architecture that uses an external program stack and key-value memory to execute hierarchical programs. NTP models learn to interpret "task specifications" that describe the task procedure and

may be in the form of sequential demonstrations, permitting one-shot imitation at test time. However, NTP requires extensive pre-training using rich execution traces that are not typically available during imitation learning. Both of these approaches therefore share limitations with reinforcement learning: they require a substantial amount of training data that includes explicit indications of the task in the form of target programs or execution traces.

One way to alleviate the data intensity of machine learning algorithms is by incorporating them into hybrid systems with non-neural components. For example, Balog et al. [16] train a deep neural network to infer program attributes, such as the presence of specific primitives operations, from input/output examples. Predicted attributes are then used to guide a variety of search methods to produce a final program. Similarly, Bunel et al. [32] generate programs using beam search guided by the predictions of a model trained with reinforcement learning. Verma et al. [203] combine reinforcement learning and search in an imitation learning paradigm: demonstrations are generated by sampling a policy learned via deep Q-learning, and Bayesian optimization is used to synthesize a program that produces similar outputs. These approaches demonstrate the value that symbolic algorithms provide to machine learning, but still fall short of the data-efficient generalization of human imitators.

2.1.1 CERIL: Cause-Effect Reasoning for Imitation Learning

Inferring intentions during imitation learning can be viewed as a process of causal reasoning, in which observable behaviors are treated as the effects of hidden causal intentions. These intentions can be inferred using *abductive inference*, which proceeds from observed effects to plausible causes by inverting causal relations. Abduction is a difficult and powerful procedure

because of its inherent uncertainty. In contrast to deductive methods, abductive inference yields defeasible hypotheses that may be subject to verification via experimental testing. In addition, because it can result in a mutually-exclusive set of possible causes, abductive inference requires a method for resolving conflict among competing hypotheses. While challenging to model, abduction is suitable for real-world applications because it accommodates the uncertainty of complex environments and provides a principled method for navigating compositional search spaces.

Abductive inference has been explored in various areas of artificial intelligence such as concept learning [91], program synthesis [48, 60, 150, 184], and diagnostic reasoning [146]. Most closely related to cognitive imitation learning is plan recognition, which has mostly been approached using probabilistic methods such as Bayesian inference that become computationally intractable with large models [39, 69, 152, 153, 180] (but see [135]).

Recently, a novel cause-effect reasoning framework called CERIL (Cause-Effect Reasoning for Imitation Learning) has been developed specifically for robotic imitation learning [95]. CERIL focuses on the role of high-level cognition in imitation and uses causal reasoning to infer the intentions governing the demonstrator's behavior and assemble plans for achieving them in similar circumstances. CERIL generalizes complex skills by breaking them down into a hierarchy of intentions at multiple levels of abstraction. Intentions at the bottom of the hierarchy guide concrete behaviors that are adapted to the particulars of the demonstration environment, while those at the top of the hierarchy correspond more generally to the goals of the task. By isolating these high-level goals from demonstrations, CERIL is able to learn generalizable skills that can be adapted to novel environments during planning.

CERIL focuses on the use of existing causal knowledge during imitation, and is agnostic to the source of this knowledge, which may be programmed by a domain expert or learned from

data using inductive methods. However, because CERIL makes use of causal chaining to construct explanations and plans, this knowledge-base need only contain elementary causal relationships, such as simple specifications of how abstract intentions are realized by sub-intentions and concrete actions. For example, relocating an object can be accomplished by grasping the object, moving the grasping hand to the desired location, and releasing the grasp. This elementary skill may be used to implement more abstract skills, such as swapping the locations of two objects, and may be achievable by alternative means such as pushing the object without grasping. This redundancy promotes generalization because abstract goals can be satisfied in various ways according to the demands of the imitation environment.

Imitation begins with a demonstration of the target skill (Figure 2.1, left side). Because extensive prior work has addressed action recognition in computer vision [57, 200, 205], CERIL processes transcripts of demonstrations recorded in a virtual environment called SMILE (Figure 2.2). These transcripts contain sequences of parameterized primitive actions such as grasping objects and simple interactions with control components such as switches and knobs. CERIL constructs an explanation for the demonstration out of causal relations with effects that match subsequences of demonstrated actions. Using abductive causal chaining, CERIL matches actions with increasingly abstract intentions in a hierarchical fashion, often resulting in competing alternative explanations. CERIL evaluates explanations using parsimony criteria that favor minimal complexity, often referred to as "Occam's razor" [94, 158]. Thus, inference results in the simplest plausible explanation that covers the observed behavior. This explanation is generalized by extracting the intentions at the top of the hierarchy, which capture the abstract goals of the task that are furthest removed from the particular demonstration environment.

During the next stage of imitation, CERIL uses the inferred top-level intentions to plan a

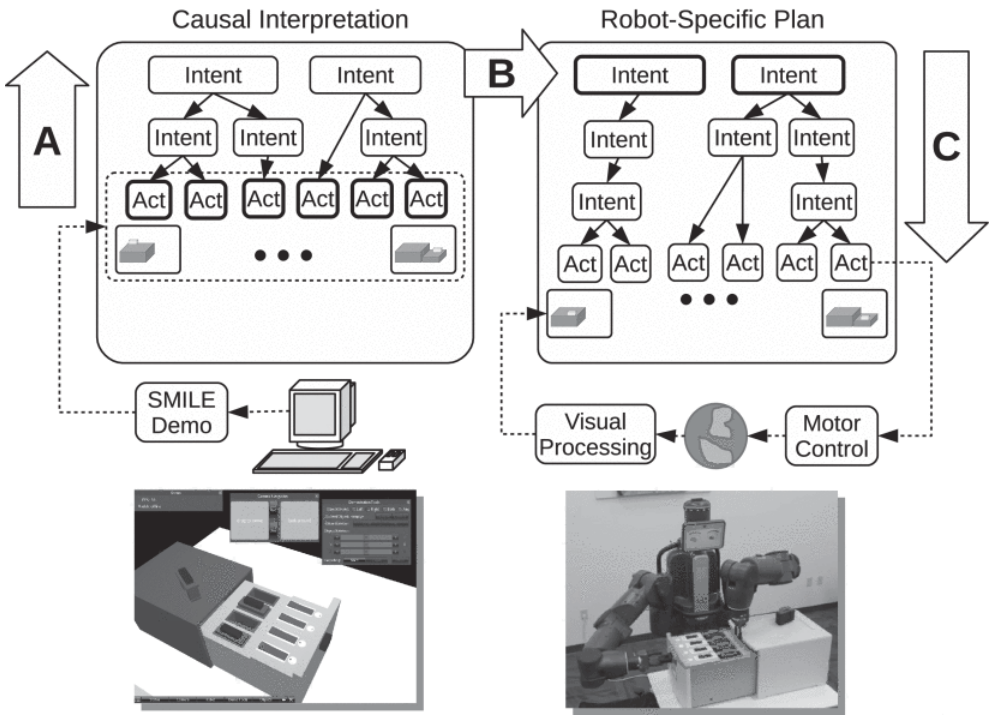


Figure 2.1: Overview of CERIL's cause-effect reasoning framework for imitation learning. Demonstrations of a procedural skill are recorded in a virtual environment (SMILE, bottom left) and interpreted to construct a hierarchical causal explanation for observed behavior (left side, **A**). This explanation is then adapted during imitation to construct a robot-specific plan that implements the learned skill (right side, **B** and **C**). CERIL issues low-level motor commands that manipulate objects identified in the environment using visual processing (bottom right). Figure reproduced from [95].

novel sequence of actions to carry out the task in an imitation environment (Figure 2.1, right side). Planning proceeds deductively from top-level goals to concrete effects using the same knowledge-base that was used to construct explanations during abductive inference. In order to successfully reproduce these goals, the plan must be tailored to the imitation environment, which may differ from the demonstration environment (Figure 2.3). For example, objects may be placed in different locations, or there may be additional irrelevant obstacles that must be removed. CERIL accomplishes this with low-level visual processing to match objects in the environment to objects that were manipulated during the demonstration. Adapting behavior to this environmental model may

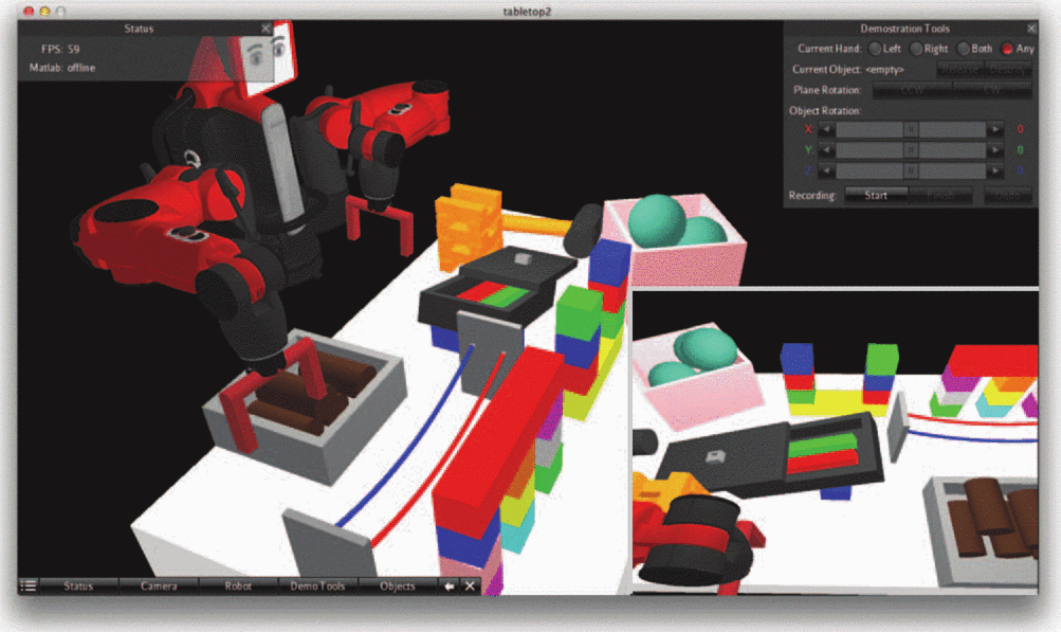


Figure 2.2: Virtual 3D environment used for recording procedural demonstrations for imitation learning (SMILE: Simulator for Maryland Imitation Learning Environment). The environment contains a tabletop with various objects that can be manipulated, as well as an avatar for a bimanual robot, Baxter (Rethink Robotics). The user can pick up, rotate, move, and release objects on the tabletop to create a sequential demonstration to be used for imitation learning. Figure reproduced from [80].

require constraining action selection (e.g. pushing rather than grasping an object) or inserting additional actions (e.g. removing obstacles).

CERIL's reuse of causal knowledge for inference and planning has an important consequence: plans can be used to generate explanations for behavior using the same principles that are used for inference [96]. When prompted for an explanation for an action during imitation, CERIL can consult the generated plan to extract causal chains that justify the action in the context of the goals of the task. promotes transparency and trustworthiness that is crucial for effective human-robot collaboration, and significantly eases the burden of debugging faulty behavior.

CERIL is a significant step forward in cognitive imitation learning for several reasons.

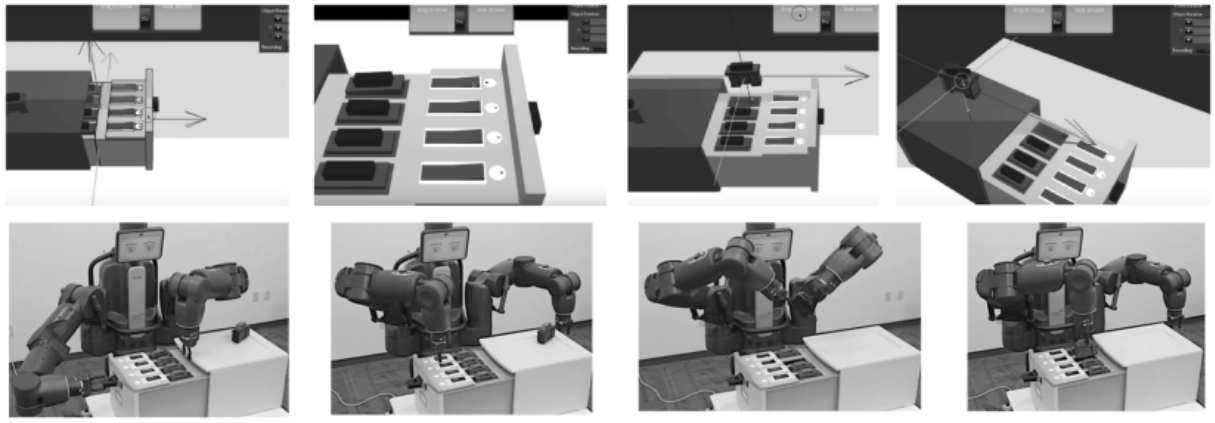


Figure 2.3: Real-world imitation of a virtually demonstrated procedure. The demonstration is generalized to the imitation environment by taking into account its differences from the demonstration environment. Here, Baxter (Rethink Robotics) learns to replace a broken disk drive located in different slots in the demonstration (top) and imitation environment (bottom). Figure from [95].

1) By using domain-general causal reasoning algorithms, CERIL is broadly applicable and can seamlessly integrate tasks across domains using common causal knowledge. This has important implications for transfer learning, which is a significant challenge in machine learning [196]: new skills can be learned more efficiently if they can be broken down into previously learned skills, which is achieved in CERIL's knowledge-base of elementary causal relations. 2) Causal chaining with compositional knowledge allows CERIL to construct hierarchical plans and explanations with arbitrary depth. The exponentially increasing size of compositional search spaces makes several approaches such as Bayesian inference intractable. 3) CERIL is agnostic to the source of its causal knowledge, which can be provided by a human domain author or learned from data using inductive methods. In addition, this knowledge can be provided incrementally without disturbing the integrity of prior knowledge, which is a significant limitation in machine learning [63, 104].

However, CERIL's correspondence with human imitation learning is limited in two ways. 1) CERIL's algorithms for intention inference rely on dynamic programming methods that may not

be cognitively plausible. While this approach is effective and reasonably efficient, it is unlikely that they can be supported by human working memory, which has significant capacity limitations. CERIL also requires multiple passes through a demonstration, whereas human imitators reason about demonstrated behavior as it occurs to construct partial explanations before a demonstration is complete. 2) Although CERIL is compatible with neurocomputational methods for sensorimotor processing, it is currently implemented using only symbolic programming. A neurocomputational implementation of comparable cognitive behaviors would benefit from the powerful learning techniques available to artificial neural networks, and would contribute to research on the neurocomputational basis of cognition and consciousness [160, 161].

This dissertation addresses these limitations with *NeuroCERIL*, a purely-neural imitation learning system that reproduces CERIL's ability to explain demonstrated behavior during imitation learning (Chapter 5). NeuroCERIL implements a novel causal inference algorithm based on the hypothetico-deductive approach, an influential model of diagnostic and scientific reasoning [114, 115, 122, 158, 181]. Hypothetico-deductive reasoning involves a combination of bottom-up abductive inference and top-down predictive verification, which obviates the need for exhaustive search by focusing cognitive processing on relevant causal knowledge. This approach also allows NeuroCERIL to process demonstrations in an online fashion by iteratively constructing efficient data structures in memory that can be used to generate plausible explanations for observed behavior. In other words, NeuroCERIL's cognitive processes are much more human-like than CERIL's, and they are supported by neurocomputational mechanisms that more closely resemble those used by people during cause-effect reasoning. These mechanisms are inspired by prior work on algorithmic learning in neural networks that is reviewed in the next section below.

2.2 Algorithmic Learning in Neural Networks

Recent progress in artificial intelligence has largely been driven by advances in machine learning, particularly in neurocomputational methods such as deep learning. As demonstrated in the previous section, deep neural networks have achieved modest success in program synthesis and induction. However, significant limitations complicate deep learning approaches to human-level artificial intelligence, including dependence on large labeled datasets, architectural constraints (differentiability, restricted recurrence, etc), and the difficulty of interpreting and debugging trained models. Furthermore, while deep neural networks are remarkably effective at learning low-level sensory processing and motor control, they struggle to capture critical aspects of high-level cognition such as compositionality and causal reasoning [113, 123, 144]. For example, Neural Turing Machines [70, 71] are able to learn simple algorithms such as copying and sorting, but induced programs have a limited capacity for generalization, and it is unclear to what degree they exhibit compositionality, in part because of their lack of interpretability.

Many of these limitations are alleviated by integrating neural networks with symbolic methods in hybrid systems such as neural-guided search algorithms [5, 23, 32, 34, 92, 103, 124, 179]. The success of hybrid approaches demonstrates that neural networks benefit from robust symbolic control and memory structures that are difficult to capture in neurocomputational systems, which is puzzling given that cognitive control is carried out in the human brain. This discrepancy between artificial and biological neural networks illustrates a *computational explanatory gap* between cognitive and neurocomputational algorithms that impedes development of human-level artificial intelligence as well as efforts to understand the neural basis of cognition [157, 159]. Bridging this gap requires the development of neural architectures capable of carrying out high-level cognitive

control processes that are more readily captured by symbolic methods [160, 161].

One promising approach to neurocognitive control involves recurrent neural networks that encode symbolic information in discrete dynamical attractors [3, 169, 210]. Dynamical attractors are regions of state space toward which a dynamical system tends to evolve (e.g. patterns of activation states in a recurrent neural network). Importantly, systems occupying attractor states resist minor perturbations and remain near the attractor.

Dynamical attractors share several desirable properties with symbolic representations. 1) Because they are stable by definition, they resist perturbations that may compromise the integrity of information processing. However, their stability may be contingent upon conditions that can be controlled, making them suitable for computation. 2) Attractors are discrete entities that emerge in continuous systems. This makes them easy to distinguish from within the system (during internal computations) and from outside of the system (during inspection, input/output, etc). 3) The underlying representation (e.g. the specific pattern of activity) can be arbitrary and acquire meaning solely from its relationship with other attractors. This means that computational processes operating in attractor spaces are not constrained by the particular encoding of the symbols, which may be flexibly configured as necessary.

At the same time, the neurodynamical substrate of attractors provides unique advantages over traditional symbolic encodings. Most notably, attractor-based memory retrieval is noise tolerant, and memories can be effectively recovered from partial or degraded input patterns. This built-in error correction makes attractor networks robust to perturbations that can compound in dynamical systems, and provides an inherent capacity to generalize across similar inputs. In addition, attractor spaces need not be fixed or predetermined, and can evolve over time during learning. This may be necessary to reliably implement complex computational procedures that are sensitive to specific

activation patterns, such as the famous exclusive-or problem.

Most past work on attractor networks has focused on storage and retrieval of individual memories or sequences of memories. However, recent work has explored using cognitively-directed multiplicative gating signals to control attractor-based memory. This principle has been successfully used to program neural networks to carry out tasks requiring cognitive control of working memory, such as running memory span, n-back, card matching, and the Wisconsin Card Sorting Task [190, 191, 192, 193]. These networks are based on a framework called GALIS ("Gated Attractors Learning Instruction Sequences") that is inspired by three prominant neuroscientific hypotheses about cortical control of working memory. The first is that the cerebral cortex is organized as a distributed network of interacting regions encoding task-relevant information and procedures. This makes behavior dependent on the activity patterns stored in the network's memory and permits reprogramming without architectural modifications. Second, regions are recurrent neural networks that learn sequences of dynamical attractors using temporally asymmetric learning, which can be used to encode sequential programs and data structures. Finally, cortical regions can not only exchange information with one another, but can also control each others' functionality via region-level gating. GALIS networks use gating to transiently reconfigure their own connectivity by opening and closing pathways between regions, much like modern computer architectures control multiplexers based on instruction opcodes. In addition, adaptive gates can control the plasticity of regions, allowing the network to decide when to learn and update working memory.

The GALIS framework offers a number of benefits over contemporary neurocomputational methods. In contrast to deep learning, attractors can be rapidly constructed using biologically plausible one-shot Hebbian learning rules rather than slow iterative gradient descent. The discrete nature of attractors makes them ideal for encoding interpretable symbolic information, including

sequential programs that capture serial procedures. Because function is separated from architecture, attractor networks can be reprogrammed without reconfiguring their structure, much like digital computers. The representation of programs as data makes it possible for GALIS networks to dynamically modify their own behavior and provides an opportunity for autonomous programming. Finally, in contrast to hybrid approaches, GALIS networks can capture symbolic procedures in a purely neural architecture that does not rely on non-neural symbolic components such as external stacks and memory systems. This makes the GALIS framework relevant not only to artificial intelligence, but also to computational studies of the brain and mind [160, 161].

2.2.1 A Programmable Neural Virtual Machine

Building gated attractor networks that carry out specific tasks requires non-trivial design of neural "hardware" and "software". The network's architecture must be suitable for its application environment, and it must implement a programming language capable of expressing the task as a sequence of neural gating operations. This process could be greatly simplified by an architecture for *universal neural programming* that is compatible with a broad range of peripheral components for environmental interaction (e.g. visual processing and embodied sensorimotor control). We have recently addressed this with a programmable *Neural Virtual Machine* (NVM) that captures the functionality of conventional computer architectures in a purely neurocomputational system [97].

The NVM deviates from previous neurocognitive models in its use of distributed representations and local learning rules. Rather than assigning variables or symbols to individual neurons, the NVM uses dynamical attractors to store discrete symbolic information. As in GALIS net-

works, these attractors are chained into itinerant sequences that encode sequential programs and data structures, and are controlled by internally generated multiplicative gating signals. Attractors and their connective links are learned using a novel variant of Hebbian learning called the "Fast Store-Erase Learning Rule". In contrast to gradient-based learning, this rule can form robust associations in a single timestep using spatially and temporally local information. In addition, this rule encompasses both Hebbian and anti-Hebbian learning to overwrite old associations with new ones (hence the term "store-erase").

The workflow for building, programming, and running NVM instances is shown in Figure 2.4. First, the user specifies values for architectural parameters such as layer sizes and activation functions that are used during a one-time construction process. This produces a blank NVM instance that implements a universal programming language, akin to hardware and firmware in a conventional computer. Next, human-authored programs are assembled and loaded into the NVM instance using local learning. Importantly, new programs can be loaded without reconfiguring the architecture, and without retraining with previously stored programs. Finally, stored programs can be executed by running neural dynamics. To interface with the outside world, a "codec" is maintained that stores mappings between attractors and their assigned symbols. This allows users to translate symbolic input into activation patterns that the NVM understands, and to decode the NVM's output into human-readable form. However, the NVM operates independently of this codec, which is only used for interface and interpretation.

The NVM emulates symbolic processing in a Harvard computer architecture, which maintains separate storage locations for instructions and data [171]. Figure 2.5a shows the components of the symbolic machine with labels for some neural regions (opc, co, r1, etc). Interface with the external environment is provided through general-purpose registers that maintain activation

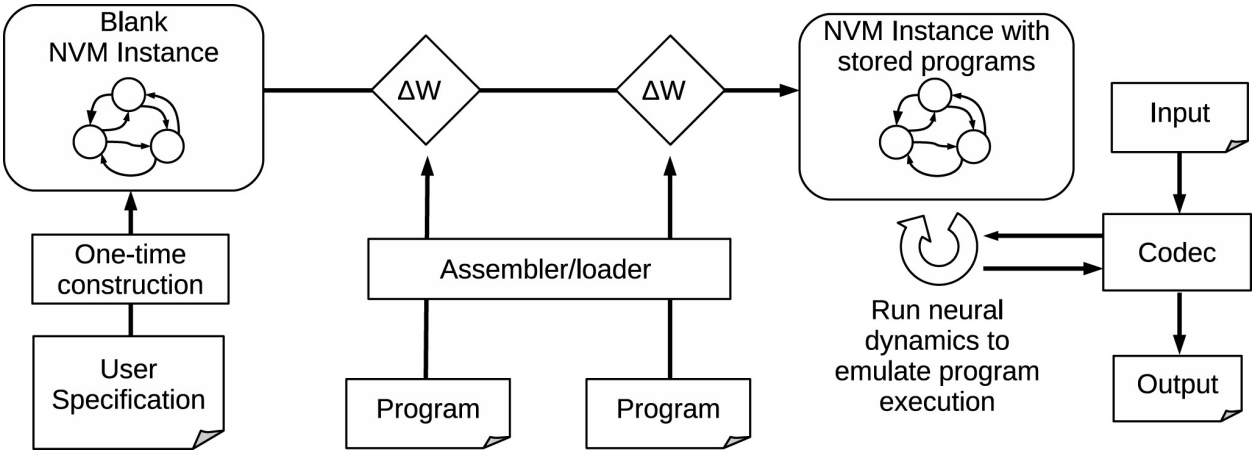


Figure 2.4: Neural Virtual Machine (NVM) workflow. Following user-configurable one-time construction (left), an NVM instance can be programmed with code written in an assembly-like language. New programs are assembled and loaded into the NVM using local learning rules without erasing previous stored programs (center). A "codec" is maintained for translating input/output symbols to/from their corresponding activation patterns during program execution (right). Figure from [97].

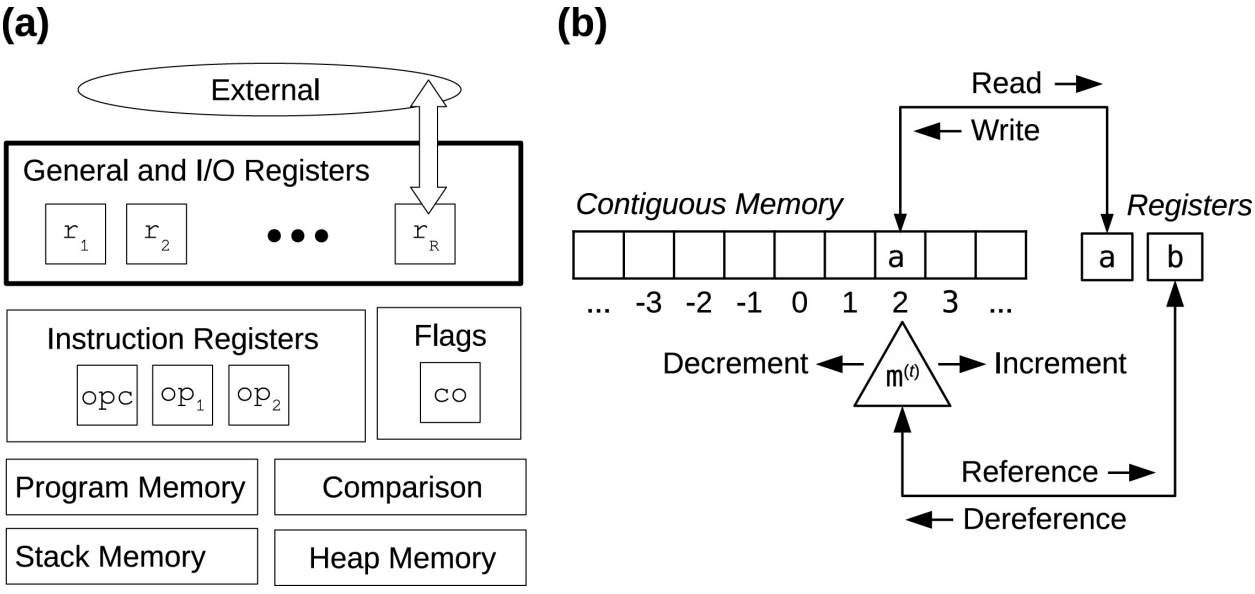


Figure 2.5: (a) Symbolic machine emulated by the NVM. External input/output runs through registers that are connected with a contiguous long-term memory resembling the tape of a Turing machine. Instructions encode sequential gating operations that control the flow of information between regions. (b) Sequential memory model depicting read/write, increment/decrement, and reference/dereference operations. Figure from [97].

patterns encoding symbols. Programs that use I/O must be written to handle asynchronous register interactions, which can be done using mutual exclusion principles. Registers also interact with a long-term contiguous memory resembling the tape of a Turing machine, as shown in Figure 2.5b. Interactions between registers and memory are mediated by instructions stored in the instruction registers. These instructions encode sequential gating operations that control information flow between regions and implement read/write and reference/dereference operations. Additional instructions increment/decrement the active memory location, simulating movement of a tape head.

The full neural architecture of the NVM is shown in Figure 2.6. Regions and pathways are indicated by small labeled squares connected by arrows. The control flow regions function as a control unit that opens and closes connection gates based on program instructions and the results of computations. During one-time construction of the NVM instance, these regions are "flashed" with "firmware" that implements the instruction set architecture. Instructions are encoded in the program memory regions, which are updated during assembly and loading. Each instruction is encoded as an attractor in the `ip` ("instruction pointer") layer that is linked to attractors encoding its opcode and operands in the `opc`, `op1`, and `op2` layers. Instruction execution begins by opening the gates on outgoing connections from `ip`, which "unpacks" the instruction by activating the corresponding attractors in the `op` layers. The opcode is then sent to the `gh` ("gate hidden") layer to begin the control sequence appropriate for the instruction. For example, read and write operations involve gating different pathways between registers and heap memory. Which operation to perform is indicated by the opcode, while the source/destination register is indicated by the first operand. The dynamics of `gh` are initialized by the opcode, but are also informed by operands to ensure that the appropriate gates are controlled to execute the instruction. Upon completion of an instruction, `gh` advances `ip` to the next instruction attractor and prepares to unpack it for the next

execution cycle.

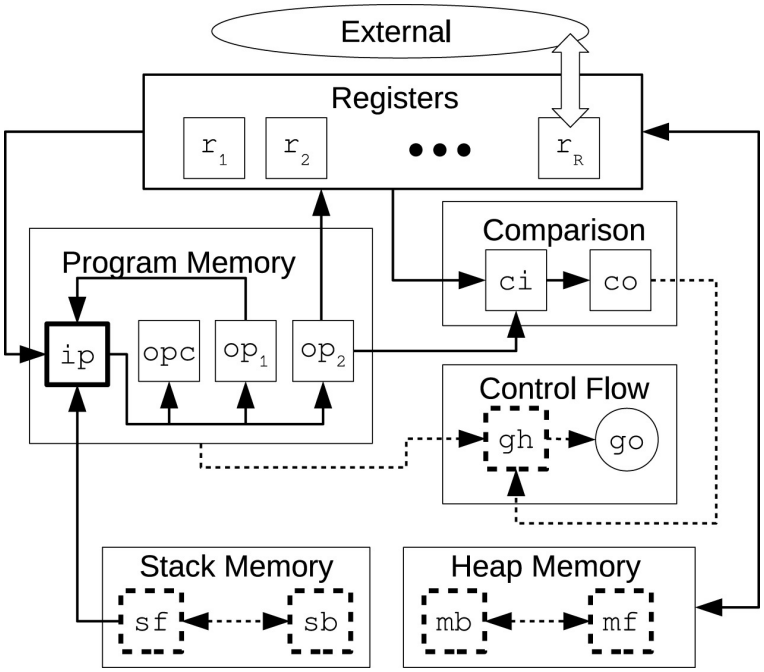


Figure 2.6: Full architecture of the NVM. Small labeled squares depict neural regions that are interconnected with gated pathways (arrows). Dashed pathways are initialized and fixed during one-time construction, whereas solid pathways may change via learning during program loading and execution. Bold squares indicate regions with recurrent connectivity. Pathways are gated by activation in the `go` ("gate output") region. For simplicity, these gates are omitted from the diagram, along with interconnections between registers. Figure from [97].

Program execution typically proceeds sequentially through instructions, which are chained together as an itinerant attractor sequence in `ip`. However, several instruction opcodes cause branching of program execution using associations between `op1` and `ip` that are learned during program assembly and loading. For example, a jump instruction includes an operand that refers to another target instruction, and can be executed by opening the pathway from `op1` to `ip` after the jump is unpacked. Jumps can be contingent on the results of prior comparison operations, which are used to compare the contents of two registers, or to compare the contents of one register with an "immediate" value encoded in the comparison instruction itself. The result of the comparison is

encoded in co ("compare output") and is available to gh via a gated pathway. When a conditional jump instruction is executed, gh follows one of two activity trajectories based on the contents of co: if the comparison was true, the target instruction is loaded from op1, otherwise ip is simply advanced to the next instruction in the program sequence.

In addition to jumps, the NVM uses stack memory to support sub-routine execution. Like jumps, these instructions encode target instructions using associations between op1 and ip. However, calling a sub-routine involves stashing the current instruction in stack memory so the sub-routine can return to it when it completes. Like heap memory, stack memory is implemented as a bidirectional chain of attractor states that emulates a tape with a read/write head. Pushing an instruction onto the stack involves advancing the head and binding the resulting attractor to the contents of ip using gated learning. The learned association can then be used to recover the instruction by opening the pathway to ip, and the head is moved back to its prior location. Stack memory supports nested sub-routine calls by storing a record of trace of calls that have not yet been returned from, providing a principled mechanism for managing compositional procedures.

The NVM stands out from prior work on universal neural computation in a number of ways. As a purely neural system, it differs from hybrid systems that augment neural networks with auxiliary components (e.g. stacks and memory units), which do not address how neurocomputation can support symbolic processing. In contrast, all of the NVM's modules are implemented using a common set of biologically-plausible neurocomputational principles, including multiplicative gating, distributed representation, and fast local learning. Each of these principles has played a significant role in both machine learning and neuroscience, but only limited prior work has explored their combination.

A notable limitation of the NVM is that it is constrained to the linear structure of itinerant

attractor sequences. This is overcome by introducing additional pathways through the architecture that enable alternative transitions between attractors. For example, program memory represents programs as linear sequences of instructions using the recurrent connectivity of the `ip` layer, and branching (jumps and sub-routine calls) requires additional links between instructions that are established in a separate pathway through the `op1` layer. Similarly, bidirectional links in linear stack and heap memory are encoded in separate forward/backward pathways that can be individually controlled by the gating mechanism during increment and decrement operations. Additional links between memory addresses are supported by reference/dereference operations that establish pointers using additional pathways through register regions.

While jumps and pointers can be used to construct complex algorithms and data structures, their current implementation in the NVM incurs significant computational costs. Complex conditional behavior (e.g., branching based on several possible conditions) requires a sequence of binary conditional operations encoded in a chain of comparison and jump instructions. Similarly, compound objects can be stored in heap memory using contiguous chains of pointers that refer to object attributes. In both cases, the cost of reaching the target state increases with the number of elements (conditions/attributes) because access depends on iteration through linear memory. Jumps and pointers also require additional memory use: jump targets are encoded in `op1`, and pointer targets (memory addresses) are encoded in register regions.

An exception to this limitation is the attractor space of the `gh` control flow region, which branches to different gating sequences according to the instruction being executed. The dynamics of this layer can be understood as an emulation of a finite state machine, where transitions between attractors are guided by external input from program memory regions (for details, see Appendix E in [97]). Although each transition requires a unique intermediate activation pattern, similar to the

auxiliary representations required for conditional jumps and heap memory pointers, transitioning to one of various targets can be done in constant time without substantial iteration through linear memory. However, this attractor space is learned during one-time construction of the NVM using a linear solver rather than local learning, and because gating behavior is program-independent, it need not be modified during program loading and execution.

In this dissertation, I address these limitations with a generalization of itinerant attractor sequences called *attractor graphs* (Chapter 3). Unlike linear sequences, attractor graphs can represent arbitrary directed graphs in neural memory, and support storage of compositional data structures with only local learning rules. This method makes it possible to represent higher-level programming languages with nested program expressions, making it easier to implement complex algorithms like those that are common in traditional symbolic AI. This possibility is realized in *NeuroLISP*, a programmable neural network that implements a subset of Common LISP (Chapter 4). NeuroLISP provides a neurocomputational foundation for learning causal reasoning algorithms for imitation learning, demonstrated with *NeuroCERIL* (Chapter 5).

3

Compositional Working Memory

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" [61, 133, 194]. 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 [2, 12, 145], behavioral planning and imitation [12, 31, 164], visual perception [25, 166, 206], and concept learning [21, 93, 148].

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 compo-

The research presented in this chapter was previously published by Elsevier in the journal *Neural Networks*: https://doi.org/10.1016/j.neunet.2021.01.031

sitional structures [61, 113, 123]. 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 [41, 77], attention mechanisms [81, 201], and external memory resources [70, 147, 188]. 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 [14, 112].

Despite this progress, and although there are disagreements about how compositionality should be evaluated in neural models [82, 133], empirical studies demonstrate that state-of-the-art deep neural networks struggle to learn systematic rules that permit generalization beyond training data [82, 111, 119]. This is in stark contrast to the ease with which such rules can be implemented in symbolic programs. This discrepancy may be 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 [10, 137]. Although its capacity is limited, working memory is capable of storing structured or "chunked" representations that provide access to a broad range of information [35, 44].

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 [5, 23, 34, 103, 124]. For example, some neural-guided search algorithms maintain compositional data structures in non-neural symbolic memory and use neural processing to inform their construction [32, 92, 179]. 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 mem-

ory arrays that incorporate key aspects of symbolic computation [70, 71]. 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 [55, 174], and minimally overlapping cell assemblies in neural blackboard architectures [198]. 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 [66, 149]. For example, recursive auto-associative memory (RAAM) networks can learn fixed-length representations of tree data structures that can be iteratively decoded to sub-trees and leaf nodes [151]. More recently, the Semantic Pointer Architecture uses symbol-like vector representations that can be recursively composed to store structured information [28, 54]. Notably, operations for composing semantic pointers can be learned using biologically-plausible learning rules in spiking neural networks [183]. 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 its representational encoding (e.g., semantic pointer), as well as any encodings for super-structures that it is contained in. For example, adding a leaf node to a tree would require reconstruction of the representations for each ancestor of the new node. This makes semantic encodings of data structures effectively immutable, as modifications involve constructing new encodings.

Performance on working memory tasks is correlated with measures of general intelligence

[42, 43, 86], and working memory operations are considered to be consciously reportable [11, 13]. 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 [161, 162], including cognitive-motor control [73, 74]. Substantial evidence indicates that activity-silent mechanisms such as rapid synaptic plasticity play a critical role in working memory [19, 121, 132, 170, 185]. 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 [8, 45, 130]. Furthermore, models based entirely on fast associative learning and itinerant dynamical attractors can learn to perform complex working memory tasks [191, 193], and can simulate Turing machines without the need for consistent maintenance of memory activation [97]. However, such models have focused on storage of individual memories and temporal sequences of memories. To my knowledge, fast associative learning has not yet been applied to explicit encoding of hierarchical compositional structures (e.g., trees) in working memory as is done here.

This chapter introduces a neural model of compositional memory based on context-dependent itinerant attractors in recurrent neural networks. This model is referred to as an *attractor graph network* or AGN. 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 [9, 38, 126, 167, 202]. 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. This chapter shows 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. This suggests 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.

3.1 Methods

This section describes a novel method for representing compositional data structures as systems of dynamical attractors in recurrent neural networks. This is illustrated with a small

multi-region model shown in Figure 3.1. The core memory region (*mem*) is a recurrent neural network with attractor states that represent general-purpose elements in short-term working memory. These elements are referred to 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.

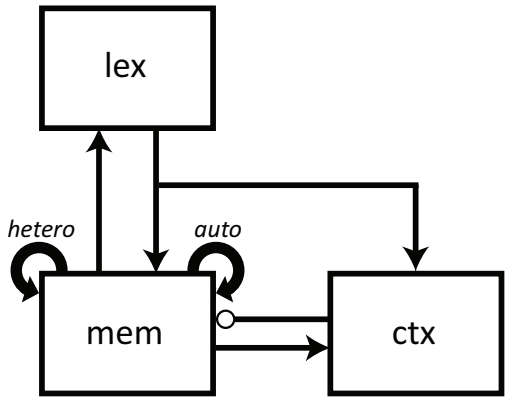


Figure 3.1: Neural model with compositional memory. Each box is a neural region and solid lines indicate connectivity between regions. Each connection is controlled by a binary gate (not shown) that determines whether it contributes to neural dynamics during each timestep. The memory region (*mem*, bottom 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*.

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. Recurrent networks with attractor graph dynamics are referred to as *attractor graph networks*, or AGNs. Section 3.1.1 describes the structure and dynamics of AGNs in detail. Section 3.1.2 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 3.1.3 shows how the model shown in Figure 3.1 can be embedded into a larger programmable neural network that constructs and manipulates data structures according to learned programmatic procedures.

3.1.1 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 states, settling momentarily at each state before rapidly transitioning to its successor [78, 131]. 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 3.1, 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 3.1.2.1). 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 3.5.

Let r denote a particular region in a neural architecture, such as mem or ctx in Figure 3.1. 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 [97, 191], as discussed in Section 3.1.3.

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. Section 3.1.2 returns to the neural circuit shown in Figure 3.1 to explain how these dynamics contribute to compositional working memory. Here the subscript r refers to a generic region. First, synaptic input is aggregated from gated recurrent connectivity and external inputs:

$$\mathbf{s}_r(t) = \underbrace{g_r^S(t)\ \omega_r\ \mathbf{v}_r(t)}_{\text{saturation}} + \underbrace{g_r^A(t)\ A_r\ \mathbf{v}_r(t)}_{\text{convergence}} + \underbrace{g_r^H(t)\ H_r\ \mathbf{v}_r(t)}_{\text{transition}} + \mathbf{I}_r(t) \tag{3.1}$$

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 3.1.2, or from outside the model for initialization purposes, as shown in Section 3.1.1.1.

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

$$\mathbf{x}_r(t) = \begin{cases} \mathbf{c}_r(t), & \text{if } g_r^C(t) = 1 \\ \mathbf{1}, & \text{otherwise} \end{cases} \tag{3.2}$$

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 3.1.
- $\mathbf{1}$ 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:

$$\mathbf{v}_r(t+1) = \sigma_r\Big(\mathbf{x}_r(t) \odot \mathbf{s}_r(t)\Big) \tag{3.3}$$

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 3.2, the hyperbolic tangent activation function is used for the *mem* AGN shown in Figure 3.1.
- $\odot$ 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

Table 3.1: 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			

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 3.1. Stages 1 and 2 take one timestep each, while Stages 3 and 4 each occur over several timesteps.

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 3.2, 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 $\mathbf{c}^{(2)}$.

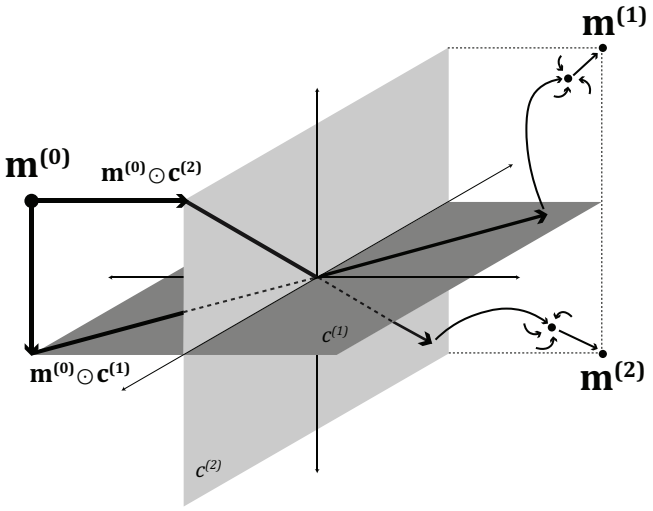


Figure 3.2: Visual depiction of context-dependent attractor transitions in the activation space of an AGN with three neurons. Each axis represents the activation level of one neuron (from -1 to +1). Two transitions from state $\mathbf{m}^{(0)}$ (left side) are shown, each with a distinct context pattern ($\mathbf{c}^{(1)}$ 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).

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:

$$\Delta A_r = \frac{1}{\rho_r^2 N_r} \sigma_r^{-1}(\mathbf{v}) \mathbf{v}^\top \tag{3.4}$$

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:

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

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:

$$\Delta H_r = \frac{1}{\lambda \rho_r^2 N_r} \sigma_r^{-1}(\mathbf{c} \odot \mathbf{v})(\mathbf{c} \odot \mathbf{u})^\top \tag{3.5}$$

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 $\mathbf{c}$. This chapter considers both traditional Hebbian learning (Equations 3.4 and 3.5) and the fast store-erase learning rule [97]:

$$\Delta A_r = \frac{1}{\rho_r^2 N_r}\left(\sigma_r^{-1}(\mathbf{v}) - \left(A_r\mathbf{v}\right)\right)\mathbf{v}^\top \tag{3.6}$$

$$\Delta H_r = \frac{1}{\lambda\rho_r^2 N_r}\left(\underbrace{\sigma_r^{-1}(\mathbf{c}\odot\mathbf{v}) - \left(\mathbf{c}\odot H_r\left(\mathbf{c}\odot\mathbf{u}\right)\right)}_{\text{masked target delta}}\right)\underbrace{\left(\mathbf{c}\odot\mathbf{u}^\top\right)}_{\text{source}} \tag{3.7}$$

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 3.7 (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 3.2, where empirical results are presented that compare memory storage and retrieval in AGNs learned with either traditional Hebbian or store-erase learning.

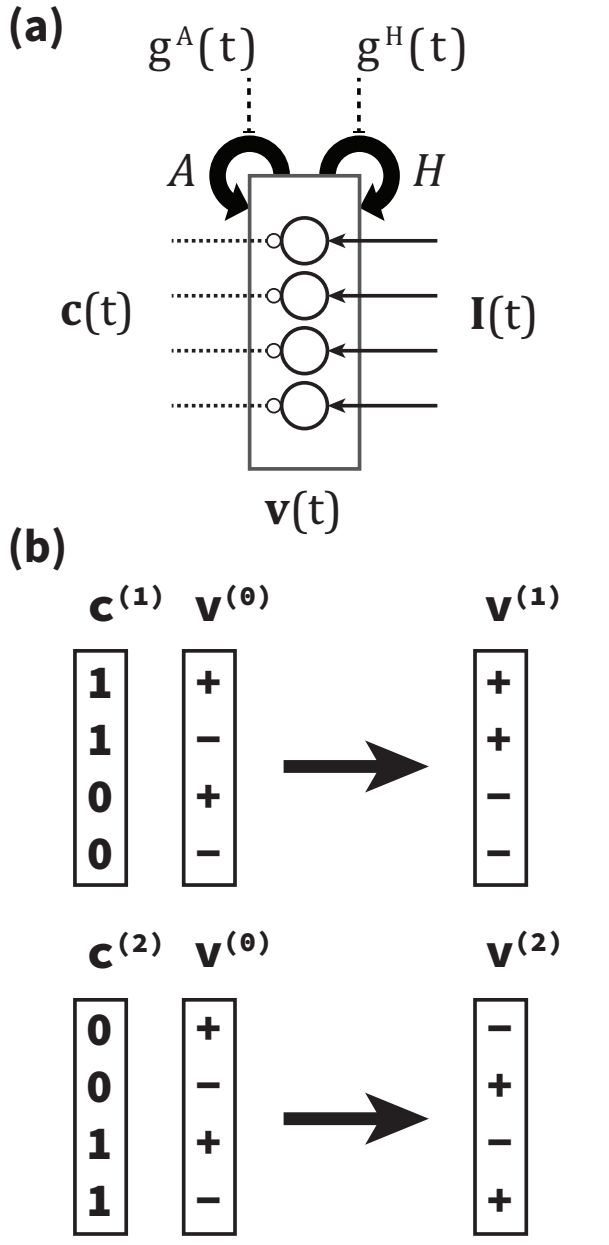
3.1.1.1 Contextual Gating Supports Functional Branching

Contextual gating signals in AGNs make it possible to store multiple transitions from an attractor state using a single hetero-associative weight matrix. This branching of attractor transitions is referred to 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.

The following simple toy example clarifies how AGNs work, and illustrates how contextual gating signals support functional branching. 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 3.3a, learns two attractor transitions (Figure 3.3b) that make up a simple attractor graph (Figure 3.3c). 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 3.4: $\mathbf{v}^{(0)} = [+1, -1, +1, -1]^\top$, $\mathbf{v}^{(1)} = [+1, +1, -1, -1]^\top$, and $\mathbf{v}^{(2)} = [-1, +1, -1, +1]^\top$. For

Figure 3.3: (next page) 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).


(a)
$g^A(t)$
$g^H(t)$
A
H
$\mathbf{c}(t)$
$\mathbf{I}(t)$
$\mathbf{v}(t)$
(b)
$\mathbf{c}^{(1)}$ $\mathbf{v}^{(0)}$ $\mathbf{v}^{(1)}$
1 1 0 0
+ − + −
+ + − −
$\mathbf{c}^{(2)}$ $\mathbf{v}^{(0)}$ $\mathbf{v}^{(2)}$
0 0 1 1
+ − + −
− + − +
(c)
$\mathbf{c}^{(1)}$
$\mathbf{c}^{(2)}$
$\mathbf{v}^{(0)}$
$\mathbf{v}^{(1)}$
$\mathbf{v}^{(2)}$

simplicity, the normalization term $\frac{1}{\rho^2 N}$ is omitted 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 3.5: $\mathbf{v}^{(0)}$ 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, the renormalization term $\frac{1}{\lambda\rho^2 N}$ is omitted for simplicity.

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

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 3.2 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

Table 3.2: 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			$\mathbf{1}$	$A\mathbf{v}(2)$	$\sigma_r(A\mathbf{v}(2)) = \begin{bmatrix} +1 \\ +1 \\ -1 \\ -1 \end{bmatrix}$

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 3.2 for the transition from $\mathbf{v}^{(0)}$ to $\mathbf{v}^{(1)}$ using context signal $\mathbf{c}^{(1)}$ (blank cells indicate zero values).

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.

3.1.2 Compositional Memory

The AGN model outlined in the previous section describes the dynamics of the *mem* region of the model shown in Figure 3.1. This section describes the interactions between the *mem*, *ctx*, and *lex* regions of this model, and explains 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 3.2). 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 3.1) have dense connectivity matrices that are learned using one-step associative learning.

Activity in the *mem* region evolves according to Equations 3.1 - 3.3. 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 3.1 and 3.3 that do not include contextual gating:

$$\mathbf{s}_r(t) = g^S_r \omega_r \mathbf{v}_r(t) + \mathbf{I}_r(t) + \sum_q g_{r,q}(t) W_{r,q}(t) \mathbf{v}_q(t) \tag{3.8}$$

$$\mathbf{v}_r(t+1) = \sigma_r\left(\mathbf{s}_r(t)\right) \tag{3.9}$$

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 3.1, 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 3.1.1, both traditional Hebbian learning (Equation 3.10) and the store-erase rule (Equation 3.11) are considered:

$$\Delta W_{r,q} = \frac{1}{\rho_q^2 N_q} \sigma_r^{-1}(\mathbf{v}) \mathbf{u}^\top \tag{3.10}$$

$$\Delta W_{r,q} = \frac{1}{\rho_q^2 N_q} \left(\sigma_r^{-1}(\mathbf{v}) - W_{r,q}\mathbf{u} \right) \mathbf{u}^\top \tag{3.11}$$

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.

3.1.2.1 Representing Compositional Data Structures

The graph-organized memory represented by the *mem* region attractor graph is suitable for encoding compositional data structures. This section focuses 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 sub-graphs 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

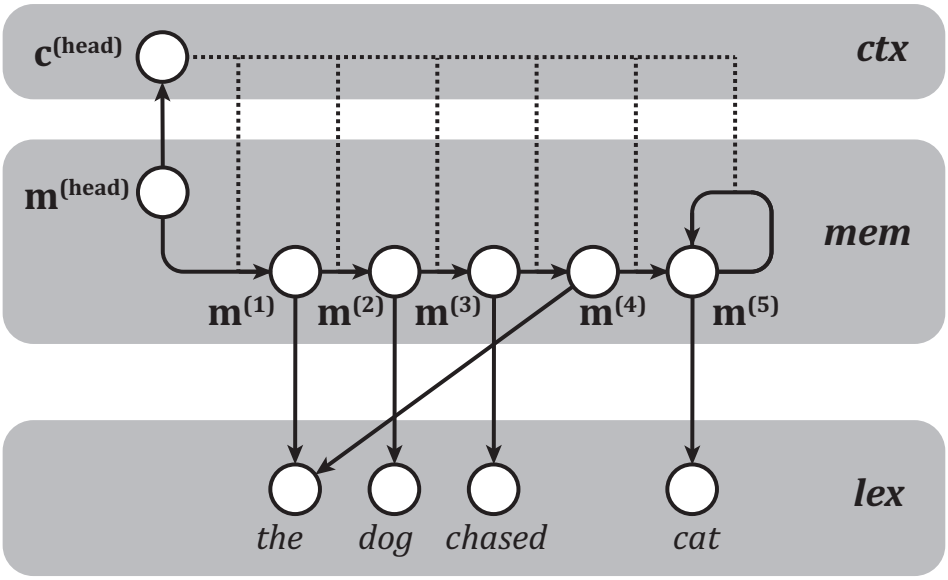
activity state can be compared with the value state. If they do not match, the lookup was not successful.

A linked list is encoded as a trajectory through the attractor graph, as shown in Figure 3.4a. 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.

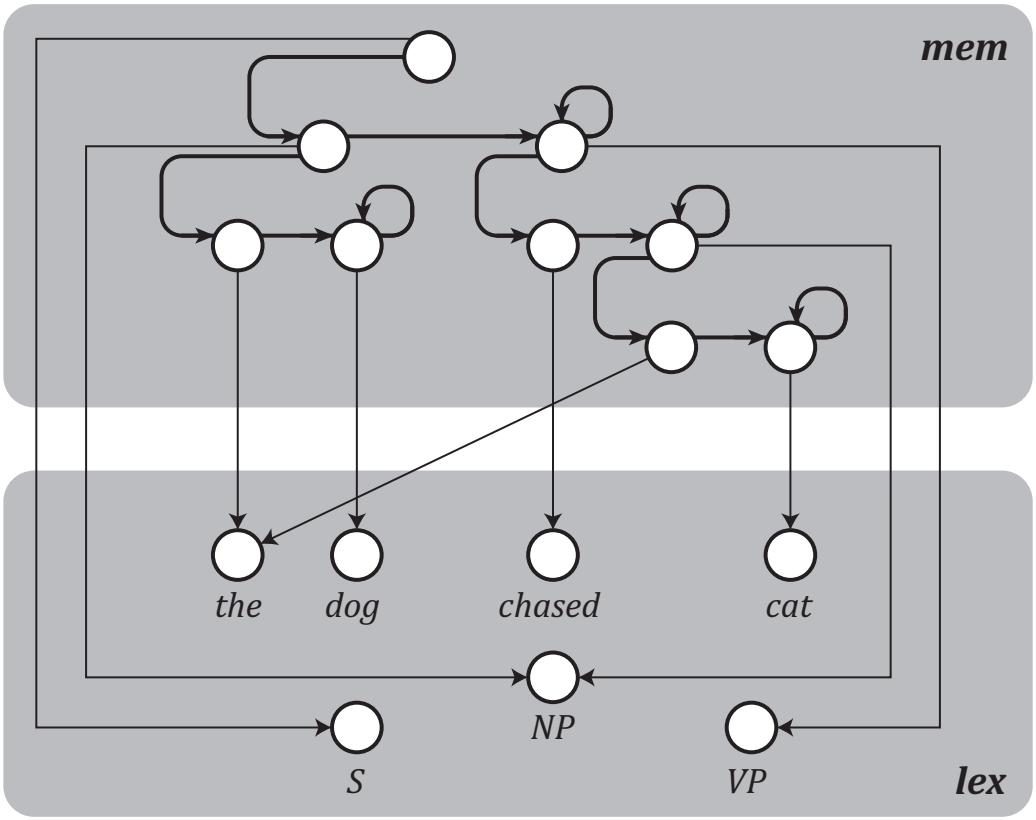
Because transitions in a linked list are contextualized by a list-specific context pattern, a memory state may be contained in several lists. In this case, the shared memory state has distinct transitions to list-specific successors. This means that memory states are *reusable components* that may be used in multiple compositional data structures. This approach is similar to [29] 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

Figure 3.4: (next page) 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).

(a)


c(head)
ctx
m(head)
mem
m(1)
m(2)
m(3)
m(4)
m(5)
lex
the
dog
chased
cat

(b)


mem
the
dog
chased
cat
NP
S
VP
lex

transitions for distinct sequences are learned in distinct but overlapping sets of connection weights.

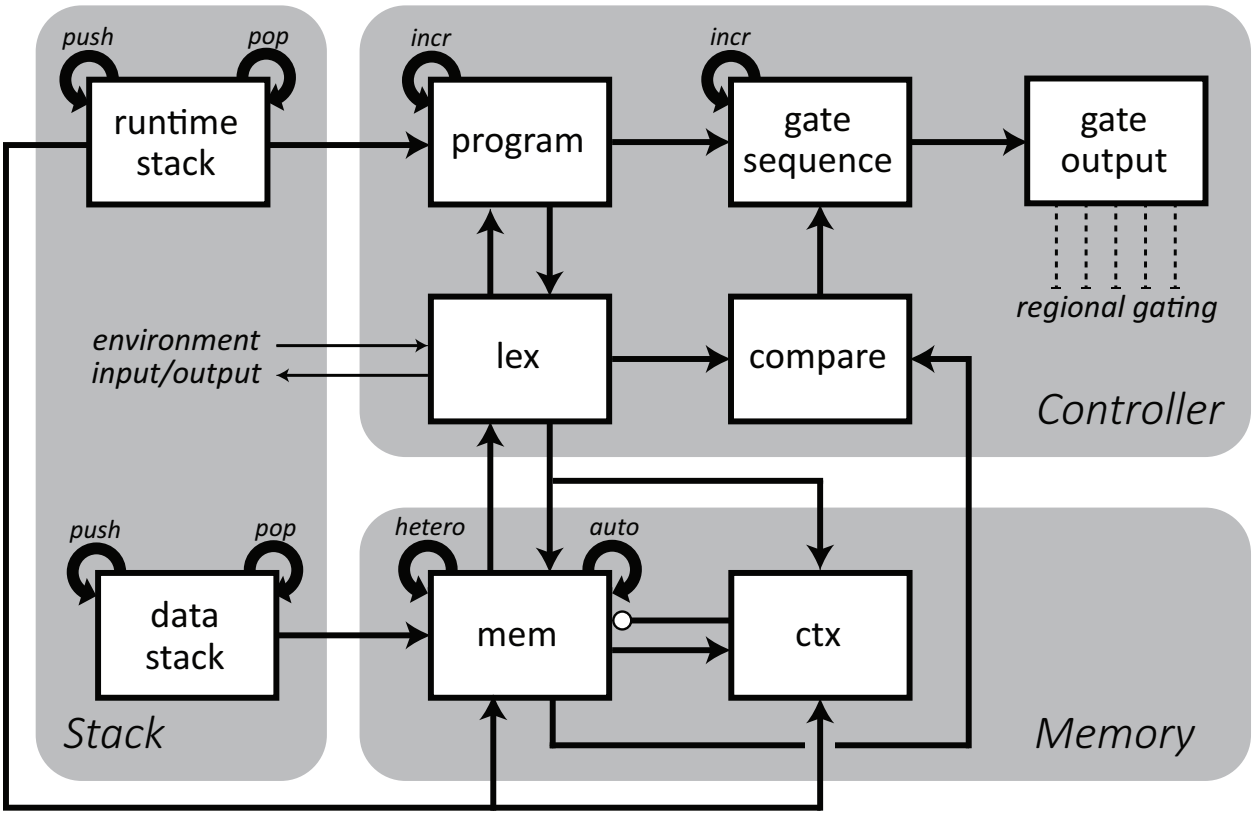
Trees can be encoded in attractor graphs by recursive composition of either associative arrays or linked lists, each with distinct advantages. Tree nodes represented by associative arrays have direct parent-child relations that are labeled by *ctx* patterns, which must be provided during tree traversal. 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 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 3.4b.

3.1.3 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 3.5 shows a programmable neural network with additional regions that provide this control. This section first describes the functionality of these control mechanisms, including the representation and execution of learned programs that make use of compositional memory. Then follows a description of a planning task that the model learns to perform, which involves constructing and modifying complex hierarchical data structures in memory.

The complete model depicted in Figure 3.5 is a programmable neural network with program-independent circuitry that is based on the Neural Virtual Machine (NVM) [97], but has several key differences. Most notably, the sequential tape-like memory of the NVM is replaced by attractor

Figure 3.5: (next page) 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.


push
pop
runtime stack
incr
program
incr
gate sequence
gate output
regional gating
environment
input/output
lex
compare
Controller
push
pop
data stack
hetero
auto
mem
ctx
Stack
Memory

graph memory (*mem* and *ctx*), permitting storage of compositional data structures via direct context-dependent associations between memory states. In addition, the model presented here 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.

The model presented here includes three major subnetworks, shown as large gray rectangles in Figure 3.5. The memory subnetwork contains the *mem* and *ctx* regions, which implement compositional memory with attractor graphs, as described in Sections 3.1.1 and 3.1.2. 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 3.1.2. 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.

3.1.3.1 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 3.1. 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 3.1.3.3).

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 heteroassociative matrices (*push* and *pop* loops in Figure 3.5).

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).

3.1.3.2 Online Learning

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

$$W_{r,q}(t+1) = W_{r,q}(t) + g^{\ell}_{r,q}(t)\Delta W_{r,q}(t) \tag{3.12}$$

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:

$$\Delta W_{r,q}(t) = \frac{1}{\rho_q^2 N_q}\left(\sigma_r^{-1}(\mathbf{v}_r(t)) - W_{r,q}(t)\mathbf{v}_q(t)\right)\mathbf{v}_q(t)^{\top} \tag{3.13}$$

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:

$$\Delta A_r(t) = \frac{1}{\rho_r^2 N_r}\left(\sigma_r^{-1}(\mathbf{v}_r(t)) - A_r(t)\mathbf{v}_r(t)\right)\mathbf{v}_r(t)^{\top} \tag{3.14}$$

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. This is addressed with an eligibility trace $\epsilon_r(t)$ that stores a target activity pattern for subsequent learning:

$$\epsilon_r(t+1) = \begin{cases} \mathbf{c}_r(t) \odot \mathbf{s}_r(t), & \text{if } g^{\epsilon}_r(t) = 1 \\ \epsilon_r(t), & \text{otherwise} \end{cases} \tag{3.15}$$

where $\epsilon_r(t)$ is the eligibility trace of region r at time t, $g_r^{\epsilon}(t)$ 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 3.1 and 3.2). When $g_r^{\epsilon}(t) = 1$, 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:

$$\Delta H_r(t) = \frac{1}{\lambda_r \rho_r^2 N_r} \left(\epsilon_r(t) - \Big(\mathbf{c}_r(t) \odot H_r(t) \mathbf{v}_r(t) \Big) \right) \mathbf{v}_r(t)^\top \tag{3.16}$$

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 3.1).

3.1.3.3 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* ($\mathbf{v}_{cmp}^{true}$). Unlike other connectivity updates, this overwrites the corresponding weight matrix rather than incrementally updating it:

$$\Delta W_{cmp,q}(t) = \frac{1}{\rho_q^2 N_q} \sigma_{cmp}^{-1} \Big(\mathbf{v}_{cmp}^{true} \Big) \mathbf{v}_q(t)^\top - W_{cmp,q}(t) \tag{3.17}$$

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 3.13, except that it completely overrides the existing weights and associates the input pattern with a fixed pattern ($\mathbf{v}_{cmp}^{true}$).

The *cmp* region has an input bias toward a *false* activity pattern ($\mathbf{v}_{cmp}^{false}$) 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 $\mathbf{v}_{cmp}^{true}$; otherwise it is $\mathbf{v}_{cmp}^{false}$ The following equation describes the synaptic input term for the cmp region:

$$\mathbf{s}_{cmp}(t) = \underbrace{g_{cmp}^{S}(t)\ \omega_{cmp}\ \mathbf{v}_{cmp}(t)}_{\text{saturation}} - \underbrace{\Big(1 - g_{cmp}^{S}(t)\Big)\ \theta\ \omega_{cmp}\mathbf{v}_{cmp}^{false}}_{\text{bias to false pattern}} + \underbrace{\sum_{q}\Big(g_{cmp,q}(t)\ W_{cmp,q}(t)\ \mathbf{v}_{q}(t)\Big)}_{\text{inter-regional input}} \tag{3.18}$$

where $\mathbf{v}_{cmp}^{false}$ 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 3.8 (with $r = cmp$). The synaptic input $\mathbf{s}_{cmp}(t)$ is transformed to neural activation $\mathbf{v}_{cmp}(t)$ by Equation 3.9 (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 ($\mathbf{v}_{cmp}^{true}$). Specifically, a value of $\theta = 0.95$ means that an input pattern must have a cosine similarity exceeding 0.95 in order to activate $\mathbf{v}_{cmp}^{true}$.

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.

3.1.3.4 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. A gated noise term is included in the *mem* and *ctx* regions. When a noise gate is opened, a random pattern of activation is established in the corresponding region:

$$\mathbf{v}_{mem}(t+1) = \sigma_{mem}\Big(\mathbf{x}_{mem}(t) \odot \mathbf{s}_{mem}(t) + \underbrace{g^N_{mem}\omega_{mem}\mathbf{n}_{mem}(t)}_{\text{gated noise}}\Big) \tag{3.19}$$

$$\mathbf{v}_{ctx}(t+1) = \sigma_{ctx}\Big(\mathbf{s}_{ctx}(t) + \underbrace{g^N_{ctx}\omega_{ctx}\mathbf{n}_{ctx}(t)}_{\text{gated noise}}\Big) \tag{3.20}$$

$$\mathbf{n}_{mem}(t) \sim Bernoulli(0.5) \qquad \mathbf{n}_{ctx}(t) \sim Bernoulli(\lambda)$$

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 3.3 (for *mem*) and Equation 3.9 (for *ctx*). Random vectors are generated by a Bernoulli process with probabilities 0.5 and λ (context density, defined in Section 3.1.1). 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.

3.1.3.5 I/O

Environmental inputs to the model are provided to the *lex* region via the external input variable $\mathbf{I}_{lex}$ (Equation 3.8). 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.

3.1.3.6 Planning Task

Planning is a high-level executive task that involves reasoning about actions and organizing them to achieve goals [56, 67]. Behavioral plans are often hierarchically structured and require compositional reasoning [31, 47]. To further test the model's ability to construct and maintain compositional data structures in memory, it was trained to perform an automated planning task using hierarchical task networks (HTNs) [56, 67]. 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 sub-actions (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.

A version of HTN planning was implemented 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 3.1.2, and actions are only decomposed if a rule is successfully retrieved.

For testing purposes, a planning domain was designed to simulate a simple repair task involving a mechanical assembly unit (see Appendix A.1 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).

3.2 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, experiments were designed 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, the model was evaluated on its ability to manipulate compositional data structures using the HTN planning task described in Section 3.1.3.6. 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. Runtime complexity was evaluated 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. For each test, the model's constructed HTN was validated, and the number of timesteps taken to complete the task was measured. The specific knowledge-bases, environments, and top-level sequences used are listed in Appendix A.1.

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 3.2.1 - 3.2.5 evaluated the model shown in Figure 3.1 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 3.4 - 3.7). All other pathways were learned with non-gated versions of these rules (Equations 3.10 and 3.11. 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 3.2.6 evaluated the full programmable network shown in Figure 3.5. In this experiment, regions were

sized according to the number of patterns to be learned for the planning task described in Section 3.1.3.6. Regions not present in Figure 3.1 used the sign/signum activation function ($\rho = 1$).

3.2.1 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. Thus, the first experiment reported here was designed 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 3.1.1). 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 3.6. 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, the number of learned attractors was limited to 64 to avoid errors in attractor convergence.

3.2.2 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 3.1.1.1). The number of transitions from an attractor is referred to as the attractor's *branching factor*. The experiment reported here evaluated learning of attractor graphs with various branching factors to determine how many transitions can be learned from a single source attractor. Specifically, AGNs were trained 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

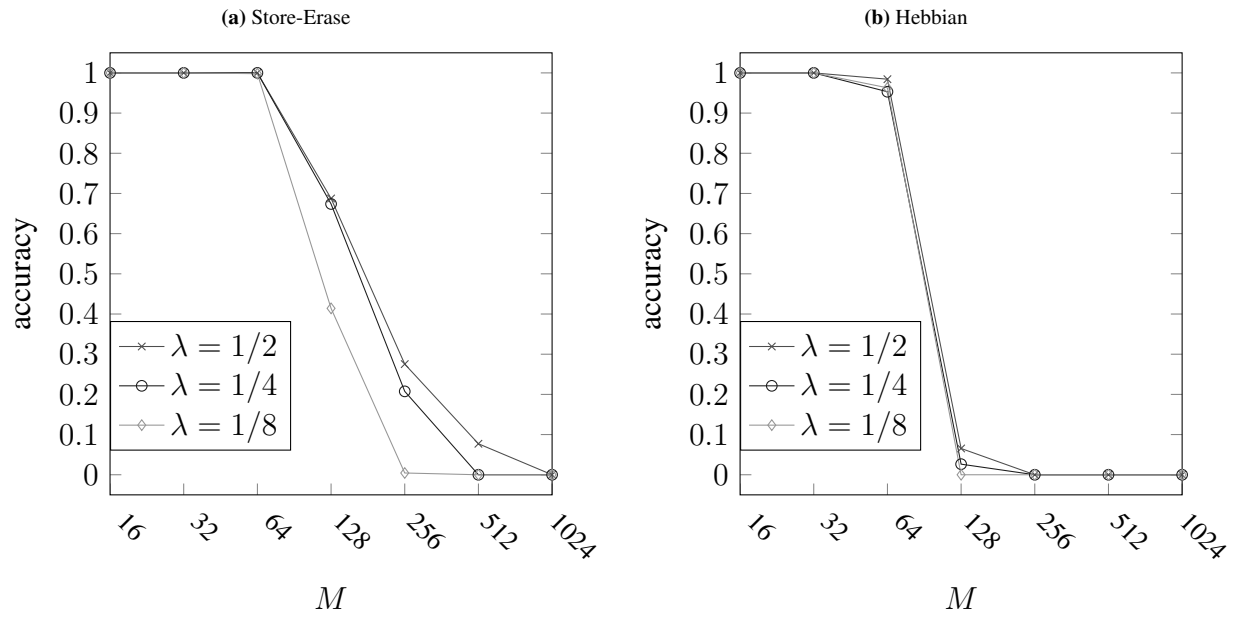


Figure 3.6: Accuracy of attractor convergence (pattern recovery/completion). Each plot shows the recall accuracy (y-axis) of an AGN with $N = 1024$ neurons trained with either the fast store-erase learning rule (left) or traditional Hebbian learning (right). The network was trained with sets of memory 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.

reliably learn $M = 64$ attractors. Thus, attractor graphs were generated 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 3.7. 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 $\lambda = \frac{1}{8}$.

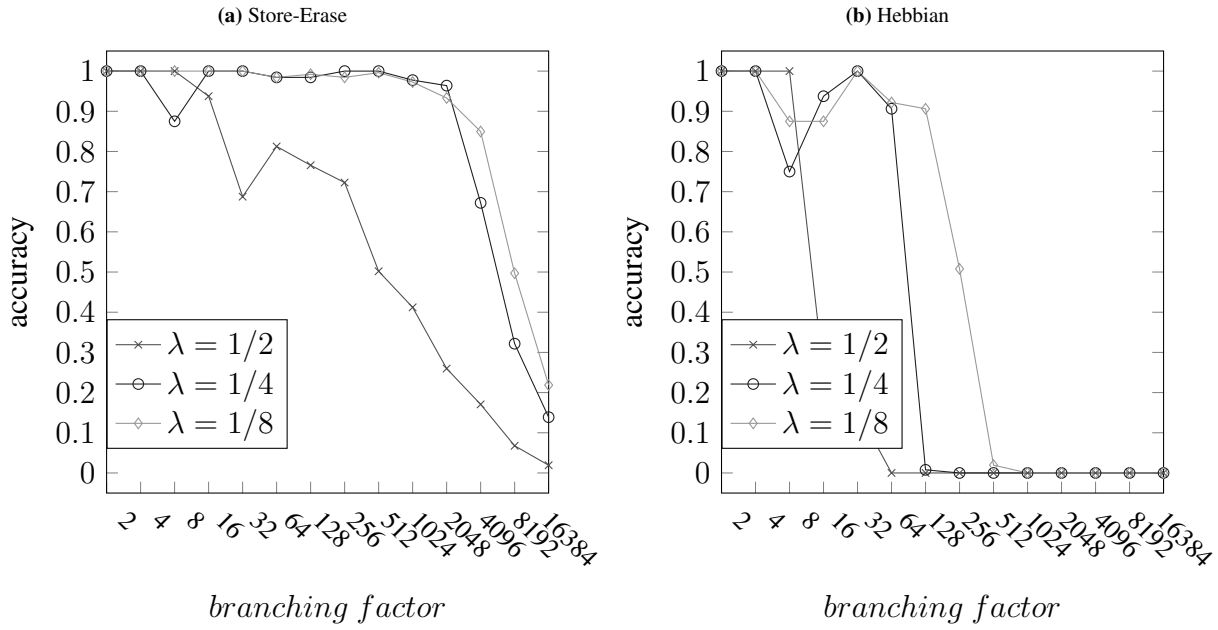


Figure 3.7: Accuracy of attractor graphs with various branching factors. Each plot shows the transition accuracy for an AGN with $N = 1024$ neurons trained with either the fast store-erase learning rule (left) or traditional Hebbian learning (right). The network was trained with attractor graphs containing $M = 64$ 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.

3.2.3 Random Graphs

The above results establish that AGNs can learn attractor graphs with high branching factors. The following experiment was designed to determine how many transitions can be stored when the transitions do not share a source node. To do so, an AGN was trained 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 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 3.8. 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, the number of learned attractor transitions was limited to 1024.

3.2.4 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. The following experiment 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 3.1.2). Each attractor served as the head of a unique list containing E elements (varied

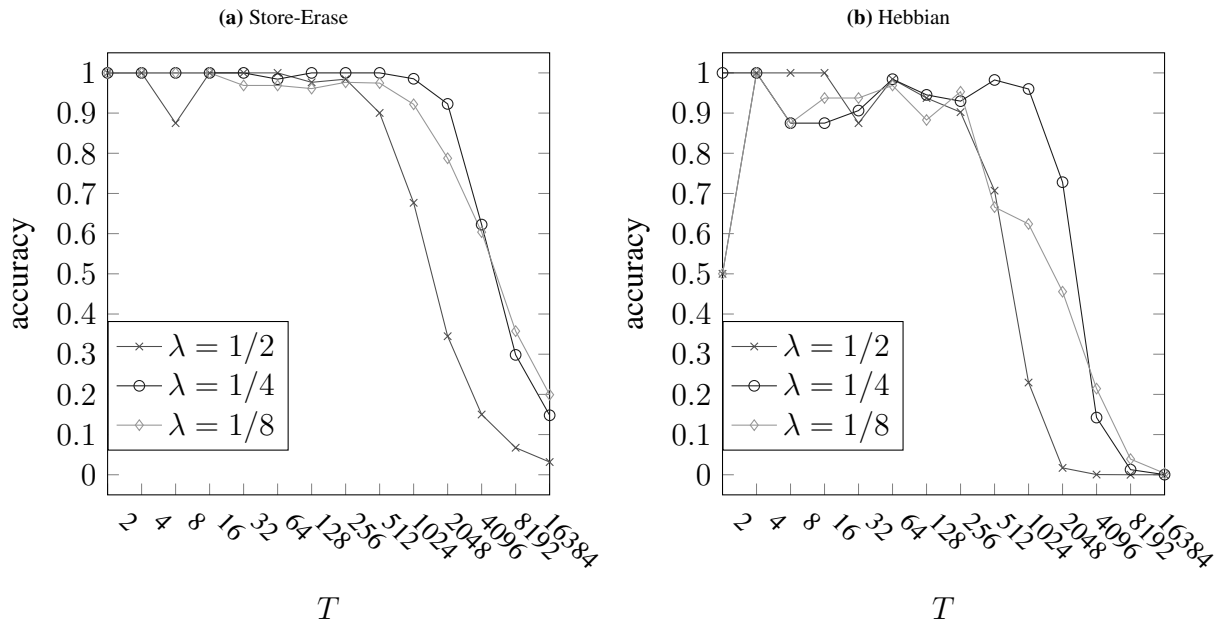


Figure 3.8: Accuracy of randomly generated attractor graphs. Each plot shows the transition accuracy for an AGN with $N = 1024$ neurons trained with either the fast store-erase learning rule (left) or traditional Hebbian learning (right). The network was trained with attractor graphs containing $M = 64$ 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.

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 3.1), and were retrieved at the beginning of each traversal at testing time.

Results are shown in Figure 3.9. 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 3.8. 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.

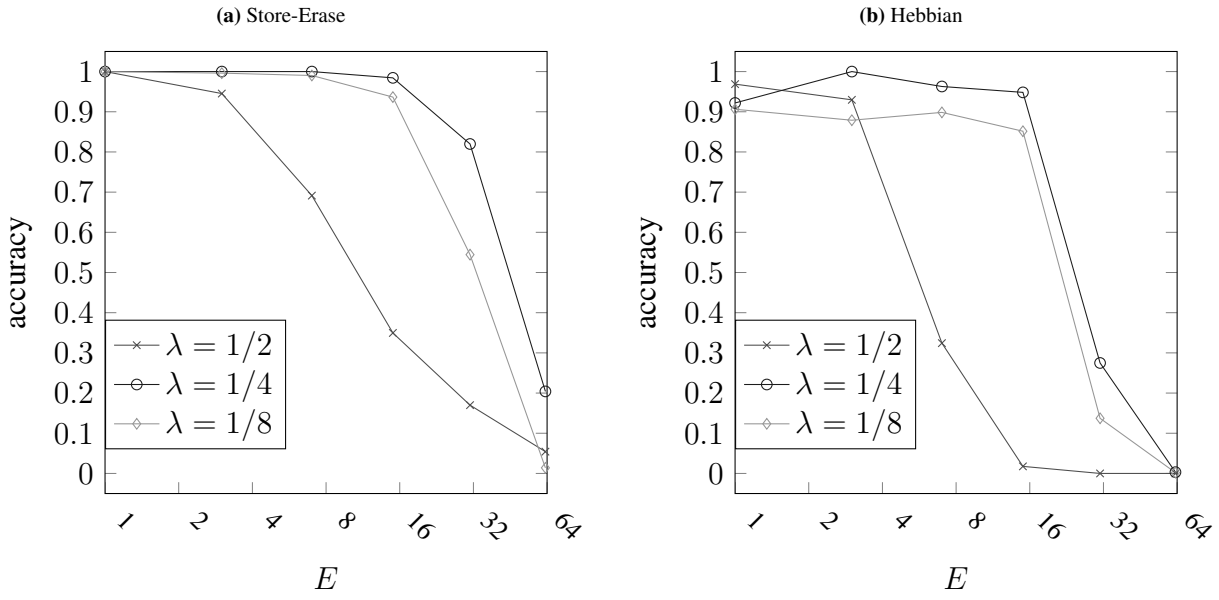


Figure 3.9: Accuracy of attractor graphs encoding multiple linked lists. Each plot shows the transition accuracy for an AGN with $N = 1024$ neurons trained with either the fast store-erase learning rule (left) or traditional Hebbian learning (right). The network was trained with attractor graphs containing $M = 64$ 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.

3.2.5 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, the model was evaluated on 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 3.10. 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 3.2.1, 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

[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/*

traditional Hebbian learning.

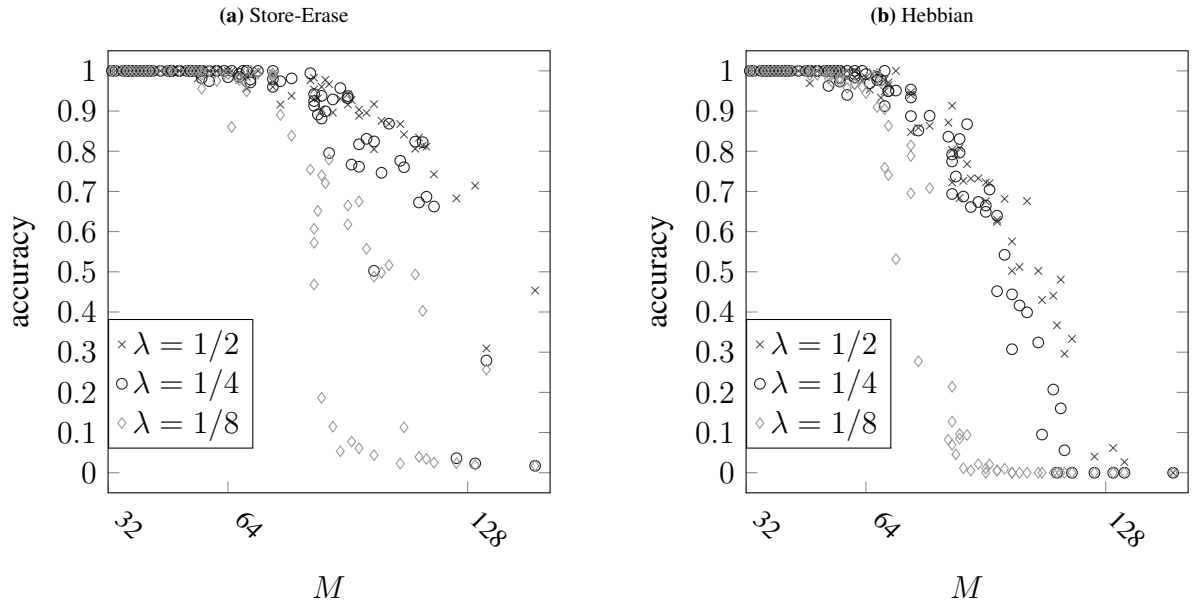


Figure 3.10: Accuracy of attractor graphs encoding parse trees. Each plot shows the accuracy for an AGN with $N = 1024$ neurons trained with either the fast store-erase learning rule (left) or traditional Hebbian learning (right). The network was trained with attractor graphs representing sentence parse 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.

The store-erase learning rule contains an anti-Hebbian component that erases previously learned associations. To evaluate this unique contribution, a second experiment was performed 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 3.11 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).

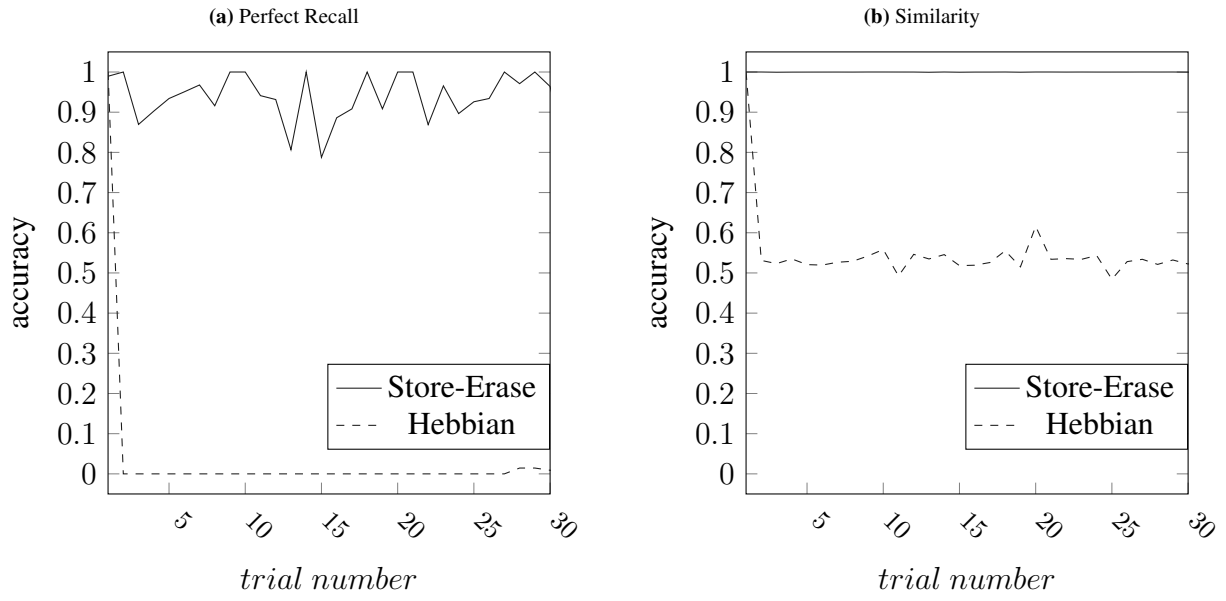


Figure 3.11: Learning parse trees without weight resets. The plot shows the performance an AGN with $N = 1024$ neurons trained with either the fast store-erase learning rule (solid lines) or traditional Hebbian learning (dashed lines), reported as the average similarity for *lex* activity patterns representing 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.

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 3.11 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.

3.2.6 Planning Task

To evaluate autonomous construction, access, and manipulation of compositional data structures, the model was tested using the HTN planning task outlined in Section 3.1.3.6. 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 [97]. During testing, the model was provided with a sequence of inputs encoding a knowledge-base, environmental 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). Constructed HTNs were checked to ensure that they 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 3.1.3.2).

The model successfully performed the task in all cases, and produced the sequence of outputs corresponding to the correct HTN tree. Figure 3.12 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.

3.3 Discussion

This chapter presented a recurrent neural network model that represents compositional data structures as systems of itinerant attractors called *attractor graphs*. This 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 transitions are selected during model execution by multiplicative contextual gating signals that control memory retrieval and iteration through learned data structures. This is referred to as *functional branching*, as the branches in attractor sequences are determined by patterns of activity and are not stored in distinct

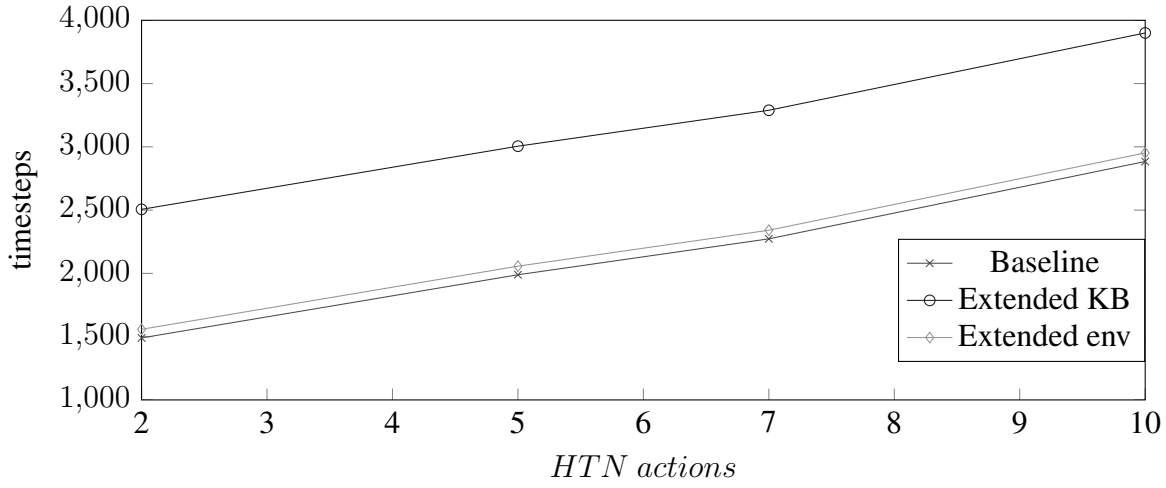


Figure 3.12: HTN planning runtime. The full programmable neural network was pre-trained to perform the HTN planning task, and evaluated on inputs representing a simple repair task domain (Section 2.3.6). Each line shows the number of timesteps taken to parse the domain knowledge-base and environment 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.

connectivity matrices.

Empirical results demonstrate that attractor graph networks 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, the 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. Two forms of one-step associative learning were evaluated: traditional Hebbian learning and the fast store-erase learning rule [97]. 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.

Results also show 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 [97]. 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. These limitations are addressed in Chapter 4, which presents a programmable neural network that implements a LISP interpreter and uses attractor graphs to store programmatic expressions.

Another significant limitation of the model presented here is that it does not identify opportunities to reuse existing memory structures, and does not have a stable long-term memory. As mentioned at the beginning of this chapter, 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. The results presented here show 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.

Attractor graph networks differ 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, AGNs do 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 [19, 121, 132, 170, 185], 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, as in recursive auto-associative memory (RAAM) [151] or the Semantic Pointer Architecture (SPA) [54], 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 [28].

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 [88, 154, 209], or that have architecturally separated representations of each hierarchical level, as in Dynamic Field Theory [53]. In contrast, attractor graphs in the *mem* region of the HTN planning model can represent arbitrarily nested hierarchical structures without distinct regions for each level of the hierarchy.

4

High-Level Neural Programming

As previously mentioned, limitations in the cognitive abilities of artificial neural networks are often addressed in hybrid models that combine neural networks with symbolic algorithms, leveraging the unique benefits of both methods. This indicates that neural networks lack the cognitive control provided by symbolic programming, which is puzzling given that human nervous systems can reliably perform a wide array of high-level cognitive tasks. This *computational explanatory gap* between cognitive and neurocomputational algorithms hinders development of human-level neurocognitive models.

The previous chapter presented a neural model of compositional working memory based on *attractor graphs*, as well as a programmable neural network that performs basic hierarchical planning. While this is a significant step toward bridging the computational explanatory gap, this network, like the NVM (Chapter 2), is limited by its use of low-level assembly-like programming that makes it difficult to express the high-level programs that are common in symbolic AI. In addition, it features segregated regions for representing programs and data, making it difficult to implement cognitive procedures that involve reasoning about behavior (e.g., planning, imitation,

The research presented in this chapter was previously published by Elsevier in the journal *Neural Networks*: https://doi.org/10.1016/j.neunet.2021.11.009

metacognition, etc). These procedures are more readily implemented in high-level languages that treat programs as "first-class citizens" that can be programmatically manipulated, such as LISP or Scheme.

This chapter presents *NeuroLISP*, an attractor neural network that can represent and execute programs written in the LISP programming language. NeuroLISP implements the core functionality of a LISP interpreter using only neural computations, and demonstrates how high-level symbolic structures can be reliably constructed and manipulated by sub-symbolic neural processes. As such, this model contributes to bridging the computational explanatory gap, and may inform studies on the neural basis of cognition and consciousness [161]. In addition, NeuroLISP serves as a purely neural replacement for the top-down control provided by symbolic algorithms in hybrid models, and has the potential to carry over the unique advantages of neural computation to high-level cognition, such as adaptive learning, improved generalization abilities, fault tolerance, and seamless integration with low-level neural models of sensory and motor processing.

To my knowledge, this is the first effort to implement a high-level functional programming language in a fixed neural architecture with distributed representations. NeuroLISP is based on the same core principles as the NVM; namely, itinerant attractor dynamics, fast associative learning, and top-down gating. However, it implements several features of high-level symbolic programming that are absent in the NVM and other programmable neural networks, such as native support for compositional data structures, scoped variable binding, and the ability to construct, manipulate, and execute programmatic expressions (i.e., programs can be treated as data). These features facilitate implementation of high-level cognitive processes by improving both the static and dynamic components of working memory.

Empirical results are presented that demonstrate the breadth of NeuroLISP's capabilities.

After verifying the correctness of the implemented interpreter with a suite of handwritten tests, the network's memory capacity was evaluated with basic programs involving list storage and variable binding. Results show that the network's memory capacity scales linearly with the size of its memory regions. Next, NeuroLISP was trained with a small library of multiway tree processing algorithms, including depth-first traversal and substitution, demonstrating its ability to learn procedures that manipulate complex data structures. Then, NeuroLISP was evaluated using programs with greater relevance to artificial intelligence. Specifically, tests were conducted with a library of sequence manipulation functions that solves the PCFG SET task, a benchmark for compositionality in machine learning models [82], and a first-order unification algorithm that performs symbolic pattern matching, a key component of automated reasoning. With sufficiently sized neural regions, NeuroLISP achieved perfect performance on test cases with significant memory and processing demands. Finally, runtime and memory usage was evaluated to show that the model can be simulated efficiently, and that it scales well on parallel computing hardware. Overall, results indicate that NeuroLISP is an effective neurocognitive controller that can replace the symbolic components that provide robust top-down control in hybrid models, and serves as a proof of concept for further development of high-level symbolic programming in neural networks.

4.1 Methods

LISP is a family of high-level programming languages with an extensive history of use in artificial intelligence [127, 136]. Today, active communities of developers exist for several dialects of LISP, including Common Lisp [178], Clojure [75], and Racket [58]. LISP is celebrated for the simplicity and consistency of its syntax and underlying data structures: the contents of memory

are made up of "s-expressions" (symbolic expressions), each of which is either an atomic symbol or a pair containing two s-expressions (referred to as a "cons cell"). This recursive definition permits expression of compositional structures such as lists and trees. Notably, s-expressions are used to represent both programs and the data they manipulate, which facilitates programmatic modification and generation of programs (i.e., *programs as data*). LISP also includes operators that allow programs to influence their own evaluation and switch between treatment of s-expressions as programs or data: the "quote" operation prevents evaluation of a sub-expression in the program, and instead returns it directly as data, while the "eval" operation explicitly induces evaluation of an s-expression that was returned as data from evaluation of a program sub-expression. Altogether, the ability to interchange programs and data makes LISP a valuable language for modeling high-level cognitive functions that include reasoning about behavior, such as planning, imitation, and metacognition, which are difficult for neural networks to learn. More generally, high-level symbolic programming provides a number of useful tools for cognitive modeling, such as scoped variable binding and compositional data structures.

NeuroLISP[1] is a purely neural model that emulates an interpreter for a dialect of LISP that includes the core functionality of Common Lisp, and serves as a proof of concept for further development of high-level symbolic programming in neural networks. The operators supported in NeuroLISP are listed in Table 4.1 and described in more detail in Table 4.2. NeuroLISP represents discrete symbols as distributed patterns of neural activation that are organized into complex data structures by learned associations in neural pathways. The high-level workflow of NeuroLISP is shown in Figure 4.1. The model is constructed by a one-time user-configurable procedure that involves learning the underlying "firmware" of the LISP interpreter using a one-step associative

[1] `https://github.com/vicariousgreg/neurolisp`

Table 4.1: Operators supported in NeuroLISP

Lists (Cons Cells)	`cons`, `car`, `cdr`, `cadr`, `list`
Hash Maps (Assoc. Arrays)	`makehash`, `checkhash`, `gethash`, `sethash`, `remhash`
I/O	`read`, `print`
Function Definition	`defun`, `lambda`, `label`
Variable Binding	`let`, `setq`
Conditional Statements	`cond`, `if`
Logical Statements	`eq`, `atom`, `listp`, `not`, `and`, `or`
Evaluation	`eval`, `quote`
Control	`progn`, `dolist`, `error`, `halt`

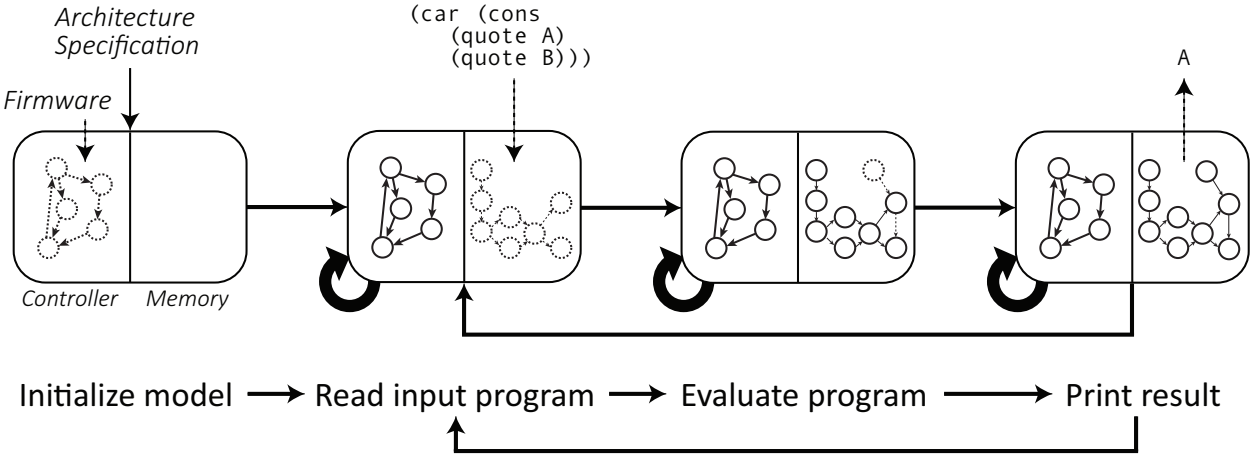


Figure 4.1: NeuroLISP workflow. Graphs depict learned distributed activity states and transitions in neural memory. First, the model is constructed and initialized by a one-time procedure (left) that constructs the neural components and "flashes" the interpreter firmware (dashed graph in left Controller half of model). Then, the model begins execution in a read-eval-print-loop that begins by parsing a sequence of inputs representing a program to be executed (center left). The program depicted here (top center) constructs a cons cell containing two symbols, `(A B)`, and retrieves the `car` (first) element. The program is fed into the model as a sequence of activation states over time, each representing one symbol in the program. During parsing, the model modifies its memory to create a representation of the program in neural memory (dashed graph in right Memory half of model). Once the program is parsed and learned in memory, it is evaluated, which may involve construction of new memories (dashed circle and arrows, center right). Upon completion, the result (`A`) is printed as output of the model (right) via a sequence of activation states representing the symbols in the output stream. Finally, the model returns to the beginning of the loop to parse the next program. The previously learned programs remain in memory as new programs are learned.

Table 4.2: Full specification of the NeuroLISP language operators

`(cons x y)`	creates a cons cell containing two values
`(car x)`	returns the first value in a cons cell
`(cdr x)`	returns the second value in a cons cell
`(cadr x)`	equivalent to `(car (cdr x))`
`(list x ...)`	creates a list containing the supplied elements (zero or more arguments), represented by a chain of cons cells, terminated by the NIL symbol (empty list)
`(makehash)`	creates a hash map (associative array)
`(checkhash key map)`	checks whether a key is contained in a map (returns `true` or `false`)
`(gethash key map)`	returns the value associated with a key in a map (undefined if key not contained in map)
`(sethash key val map)`	associates a key and value in a map
`(remhash key map)`	removes the key/value pair for a key in a map
`(read)`	reads an input expression and stores it in memory. Parentheses-delineated sequences are recursively parsed as lists, and the quote symbol (`'`) is parsed as an encapsulating quote operation (e.g., `'x` becomes `(quote x)`)
`(print x)`	prints an expression as output. Nested expressions made of cons cells are printed recursively, while functions and hash maps are printed as `#FUNCTION` and `#HASH`
`(defun name (args ...) body)`	defines a function with a name, zero or more arguments, and a body expression
`(lambda (args ...) body)`	creates an anonymous function with zero or more arguments and a body expression
`(label name (args ...) body)`	like lambda, except the function closure contains a binding from the given name to the anonymous function, permitting recursion
`(let ((var val) ...) body)`	binds a series of one or more variable/value pairs, and executes a body expression with the bindings in scope. After completion, the bindings fall out of scope.
`(setq var val ...)`	binds a series of one or more variable/value pairs in the current environment namespace. If bindings for any variable exist in the current scope, they are updated. Otherwise, new bindings are created in the default environment.

`(cond ((clause body) ...)`	conditionally evaluates expressions based on test clauses. Each test clause is evaluated in sequence until one returns `true` or the end of the list is reached. If a clause returns true, its corresponding body expression is evaluated
`(if clause true-body false-body)`	if clause evaluation returns true, the true-body is evaluated. Otherwise, the false-body is evaluated
`(eq x y)`	returns true if `x` and `y` are identical. Does not check for structural equivalence in non-atomic data structures (i.e., different cons cells with the same contents are not considered equal)
`(atom x)`	returns true if `x` is an atomic symbol
`(listp x)`	returns true if `x` is a cons cell
`(not x)`	returns true if `x` is `false` or `NIL`
`(and x y ...)`	returns true if none of the arguments evaluate to `false` or `NIL`. Evaluation is short-circuited if an argument evaluates to `false` or `NIL`
`(or x y ...)`	returns true if all of the arguments evaluate to `false` or `NIL`. Evaluation is short-circuited if any argument evaluates to something other than `false` or `NIL`
`(eval expr)`	evaluates the return value of evaluating an expression
`(quote expr)`	returns an expression without evaluating it
`(progn expr ...)`	evaluates a sequence of expressions and returns the return value of the last expression
`(dolist (var list ret-val) body)`	iterates through a list, binding each element to a variable, and evaluating a body expression with the binding in scope. Upon completion, returns either `NIL` or evaluates an optional `ret-val` expression and returns the result
`(error msg)`	prints an error with an optional message, and halts the interpreter
`(halt)`	halts the interpreter

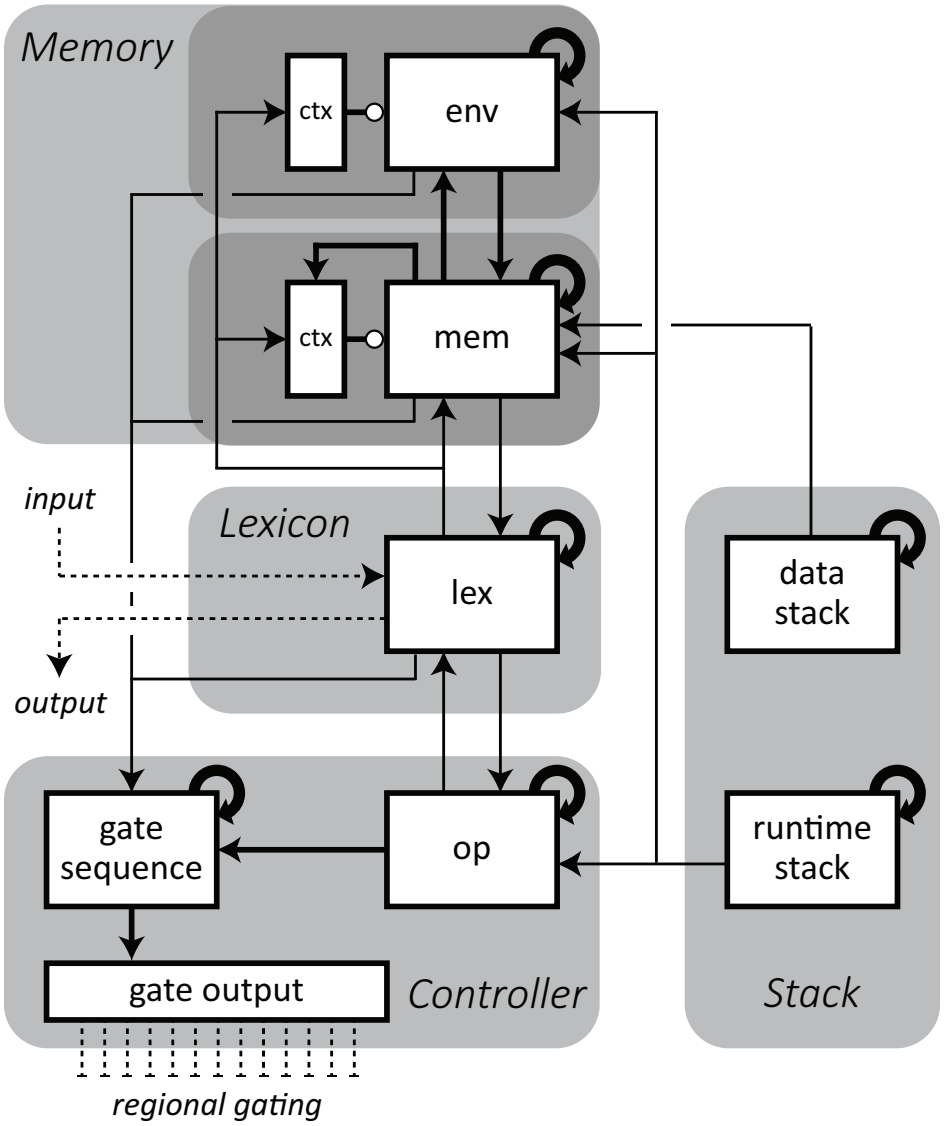
learning rule. Subsequently, the model executes a cycle of activity that 1) reads a sequence of input activity patterns specifying a program to be executed, 2) evaluates the program according to the implemented language by modifying its memory, and 3) prints the result as a sequence of output activity patterns. Programs are read in as temporally-extended sequences of neural inputs, and stored in memory as attractor graphs (Chapter 3) in a recurrent neural region that represents a shared program/data memory space. During evaluation, new memories are constructed based on the interpreted programmatic expressions, and the final result is printed via sequential activation of neural patterns that represent a stream of output symbols. The neurocomputational procedures involved in parsing, evaluation, and printing are discussed in Section 4.1.3.

The following first outlines the mechanisms that govern model execution and the various types of dynamics that they support (Section 4.1.1). Then, Section 4.1.2 describes the fundamental data structures of the virtual interpreter, their representation as systems of attractors, and the basic operations that are performed on them via algorithmic control of top-down gating. This is followed by an explanation of the virtual interpreter, including expression evaluation, comparison operations, input/output, scoped variable bindings, function definitions, and function applications (Section 4.1.3). Finally, experimental methods are outlined in Section 4.1.4, along with empirical results that demonstrate that NeuroLISP properly implements the LISP programming language and can successfully execute high-level programs (Section 4.2).

4.1.1 Neural Architecture

NeuroLISP is a multi-region recurrent neural network with gated inter-connections that implements a virtual LISP interpreter. The architecture of the model (shown in Figure 4.2 and

Figure 4.2: (next page) NeuroLISP architecture, inspired by the Neural Virtual Machine [97] and the stack machine architecture of traditional LISP machines [106]. The model is made up of several neural regions (boxes) with recurrent and inter-regional connectivity (looped and straight arrows with solid lines). Connections are controlled by neurons in the *gate output* region of the Controller sub-network (bottom left), which determine the components of the model that are active at each timestep (regional gating, dashed lines). Each gate ($g(t)$ with subscripts in Equations 4.1 - 4.6) is assigned to a unique neuron in the *gate output* region, and its activation level is used to determine whether the gate is open or closed at each timestep. Activation of the *gate output* region is guided by a cascade of regions with recurrent dynamics (*gate sequence* and *op*) that implement the core functionality of a virtual LISP interpreter. Together, the Controller regions translate learned LISP programs to temporally-extended sequences of regional gating that specify pathways for information processing over time, much like the control unit of a conventional computer architecture. The lexicon region (*lex*, center) serves as a bridge between model components, and its activity patterns represent discrete symbols that may correspond to interpreter functions (i.e., LISP operators) or arbitrary symbols that can be read from or written to an external environment (dashed input/output lines, center left). Data structures, including LISP programs, are represented by systems of attractors in the core memory region (*mem*, center), which store symbolic contents via the pathway from *mem* to *lex*. The remaining components support interpretive functions: the environment region (*env*, top) stores a tree structure containing namespaces of variable bindings that are modified and accessed during program execution, and the Stack sub-network regions (*runtime stack* and *data stack*, right) store stack sequences made up of pointers to various activation states in the model, making them accessible without persistent maintenance.


Memory
ctx
env
ctx
mem
input
Lexicon
lex
data
stack
output
gate
sequence
op
runtime
stack
gate output
Controller
Stack
regional gating

described in more detail in Appendix A.2) is inspired by the Neural Virtual Machine [97] and the stack machine architectures used for early LISP machines [106]. A Controller sub-network (bottom left) controls the flow of information processing through the model over time by translating patterns of activation into temporally-extended gating of model components. These activation patterns represent learned programs that are evaluated by the underlying virtual interpreter, and can be modified during program evaluation (i.e., programs can be treated as data). Discrete symbols are represented by patterns of activation in a lexicon region (*lex*, center of Figure 4.2) that can be exchanged as input and output with an external environment, allowing the model to be programmed via environmental interactions. Unlike contemporary deep neural networks, NeuroLISP uses a one-step local learning rule that permits rapid modification of associative networks within and between regions. This learning rule is used for both one-time initialization of interpreter functions and online learning during program execution (e.g., when modifying variable bindings, creating new data structures, and modifying runtime/data stacks).

Regions in NeuroLISP represent symbolic information as distributed patterns of activation, and function according to a shared set of rules for activation dynamics and learning. Neurons in a region r receive inputs from a variety of sources, each with a unique gate that determines when it is active during model execution:

$$\mathbf{s}_r(t) = \underbrace{\sum_{q,\ell} g_{r,q[\ell]}(t)\; W_{r,q[\ell]}(t)\; \mathbf{v}_q(t)}_{\text{weighted connectivity}} + \underbrace{g_r^{bias}(t)\; \mathbf{b}_r}_{\text{bias}} + \underbrace{g_r^{noise}(t)\; \mathbf{n}_r(t)}_{\text{noise}} + \underbrace{g_r^{read}(t)\; \mathbf{I}_r(t)}_{\text{external inputs}} + \underbrace{g_r^{saturate}(t)\; \sigma_r^{-1}(\mathbf{v}_r(t))}_{\text{maintenance}} \tag{4.1}$$

where $\mathbf{s}_r(t)$ is a vector of cumulative synaptic input to region r at time t that is aggregated from several sources:

- weighted inputs from connected regions (solid lines with arrow heads in Figure 4.2). $W_{r,q[\ell]}$

is a weight matrix for the connection from region q to region r that is active when $g_{r,q[\ell]}(t) = 1$, ℓ is a label that distinguishes between weight matrices that share source and target regions, and $\mathbf{v}_q(t)$ is a vector of neural activity in source region q at time t. When $r = q$, the connection is recurrent (looped arrows in Figure 4.2).

- bias vector. $\mathbf{b}_r$ is a bias vector for region r that is active when $g_r^{bias}(t) = 1$. The bias term is used by the *gate sequence* region during comparison operations (Section 4.1.3.2).

- random noise. When $g_r^{noise}(t) = 1$, a vector of random inputs $\mathbf{n}_r(t)$ generates a random activity pattern in region r at time t. The random vector $\mathbf{n}_r(t)$ is produced by a Bernoulli process with probability λ_r. For regions with recurrent dynamics, $\lambda_r = 0.5$ to maintain balance between positive and negative activation levels. For context regions (labeled *ctx* in Figure 4.2), λ_r is a variable parameter (for details on the implications of this parameter on contextualized attractor dynamics, see Chapter 3).

- external inputs. When $g_r^{read}(t) = 1$, region r "reads" an input pattern $\mathbf{I}_r(t)$ from the external environment. The dashed line entering the lex region in Figure 4.2 indicates external inputs. The adjacent dashed line labeled $output$ indicates gated outputs that are "printed" to the environment when $g_r^{write}(t) = 1$. Unlike other gates in the model, these gates signal to the external environment to interact with the activity in the *lex* region. In principle, input and output processes could be implemented by gated pathways with additional sensory and motor networks that control continuous behavior in a simulated or real environment. Instead, the model presented here focuses on cognitive control of such processes, and it omits the low-level networks involved with sensorimotor processing.

- activity maintenance. When $g_r^{saturate}(t) = 1$, activity $\mathbf{v}_r(t)$ in region r is cycled back into the region's inputs to maintain it over time. σ_r^{-1} is the inverse of the region's activation function. For simplicity, it is assumed that $g_r^{saturate}(t) = 1$ whenever all of the above gates are closed (i.e., a region maintains its activation pattern whenever it is not receiving synaptic input).

A region may also receive gated inputs that contextualize its dynamics via multiplicative modulation:

$$\mathbf{x}_r(t) = \prod_q \begin{cases} \mathbf{v}_q(t) > 0, & \text{if } g_{r,q}^{context}(t) = 1 \\ \mathbf{1}, & \text{otherwise} \end{cases} \tag{4.2}$$

where $\mathbf{x}_r(t)$ is a vector of cumulative multiplicative inputs to region r at time t, $\mathbf{1}$ is a vector all ones, and $\prod$ indicates the Hadamard product of a set of vectors. $\mathbf{x}_r(t)$ is aggregated from the activity state $\mathbf{v}_q(t)$ of each region q that provides multiplicative inputs when the corresponding gate $g_{r,q}^{context}(t) = 1$. These connections are depicted by solid lines with circular heads in Figure 4.2. When none of these gates are active (or if region r has no contextual inputs), $\mathbf{x}_r(t) = \mathbf{1}$. Note that in NeuroLISP, there is a single dedicated context region for each recurrent region that receives contextual input (*mem* and *env*), but the mathematical model presented here does not impose such constraints.

These two types of inputs are combined and passed into the neural activation function:

$$\mathbf{v}_r(t+1) = \sigma_r\Big(\mathbf{x}_r(t) \odot \mathbf{s}_r(t)\Big) \tag{4.3}$$

where $\mathbf{v}_r(t)$ is a vector of neural activity in region r at time t, σ_r is the activation function of neurons in region r, and $\odot$ is the Hadamard product. The synaptic inputs $\mathbf{s}_r(t)$ are gated by multiplicative inputs $\mathbf{x}_r(t)$ before being passed into the activation function. When $\mathbf{x}_r(t)$ contains zeroes, the corresponding neurons receive no net input. Note that recurrent regions with contextual dynamics must use a sign-preserving bipolar activation function (e.g., sign/signum or the hyper-

bolic tangent). This ensures that deactivated neurons function differently from neurons receiving strong negative input.

Learning in the model occurs in two stages, each controlled by a different type of gate. The first stage involves updating a regional eligibility trace to store the current pattern of activation as a target for subsequent learning:

$$\epsilon_r(t+1) = \begin{cases} \sigma_r^{-1}\Big(\mathbf{v}_r(t+1)\Big), & \text{if } g_r^{\epsilon}(t) = 1 \\ \epsilon_r(t), & \text{otherwise} \end{cases} \tag{4.4}$$

where $\epsilon_r(t)$ is the eligibility trace for region r at time t, $\mathbf{v}_r(t+1)$ is the most recently computed activity pattern in region r for timestep $t+1$ (Equation 4.3), and σ_r^{-1} is the inverse of the activation function for region r. When $g_r^{\epsilon}(t) = 1$, $\epsilon_r(t)$ is updated such that $\sigma_r\left(\epsilon_r(t+1)\right) = \mathbf{v}(t+1)$. $\epsilon_r(t)$ is referred to as an eligibility trace, a term borrowed from reinforcement learning, because it temporarily stores an activity state for use in the second stage of learning, in which a pathway-specific weight matrix is updated with the store-erase learning rule [97]:

$$\Delta W_{r,q[\ell]}(t) = \underbrace{\frac{1}{||\mathbf{v}_q(t)||}}_{\text{norm}} \left(\underbrace{\epsilon_r(t) - \Big(\mathbf{x}_r(t) \odot W_{r,q[\ell]}(t)\ \mathbf{v}_q(t)\Big)}_{\text{target delta}} \right) \underbrace{\mathbf{v}_q(t)^{\top}}_{\text{source}} \tag{4.5}$$

$$W_{r,q[\ell]}(t+1) = W_{r,q[\ell]}(t) + g_{r,q[\ell]}^{learn}(t)\ \Delta W_{r,q[\ell]}(t) \tag{4.6}$$

where $W_{r,q[\ell]}(t)$ is the weight matrix for connection ℓ from region q to region r at time t. Weight updates are distributed across the weight matrix and are normalized according to the magnitude of the source pattern. When $g_{r,q[\ell]}^{learn}(t) = 1$, weights are updated such that the current inputs will produce the eligibility trace in the future:

$$\mathbf{x}_r(t) \odot W_{r,q[\ell]}(t+1)\ \mathbf{v}_q(t) = \epsilon_r(t)$$

This equality is only guaranteed to hold for the most recently learned association, which may deteriorate as additional associations are learned. This was investigated empirically in Chapter 3,

in which practical memory capacities were established for networks that learn using contextually-gated store-erase learning.

The mathematical model outlined above affords a diverse set of dynamics that depend on top-down control of regional gating over time (e.g., $g_{r,q[\ell]}(t)$, $g_r^{noise}(t)$, etc). In NeuroLISP, these gates are controlled by neurons in the *gate output* region (one neuron per gate) based on learned sequences of activation patterns in the Controller sub-network (Figure 4.2). This allows the model to be "programmed" with new computational procedures that specify pathways through which activation flows, much like opcodes in the instruction set of a conventional computer architecture. These procedures make up the core functionality of the implemented language interpreter that are established using one-step learning during model construction, and remain fixed during model execution. Among these procedures are the core functions for constructing and accessing compositional data structures, which are implemented as systems of attractors linked by contextualized transitions (Section 4.1.2).

Connection gating is used for both inter-regional and recurrent connectivity. Inter-regional gating allows the model to control the spread of activation between neural regions, initializing a target region to a state that is specified by learned associations via weighted inputs from a source region. Recurrent dynamics within a region fall into one of two categories: attractor convergence and sequential transitions. By interleaving these dynamics, a region can be made to iterate through a sequence of attractor states, settling at each attractor before advancing to the next state in the sequence (i.e., itinerant attractor dynamics, [78, 131]). Inter-regional and recurrent dynamics can be combined into complex behavior that is orchestrated by the gate controller according to learned "programs". For example, inter-regional gating may be used to initialize a region to the start of an attractor sequence that can be traversed with subsequent recurrent gating.

Attractor itinerancy is typically limited in that each state has a single successor state in the sequence. This is overcome by the addition of "contextual" multiplicative gating (Equations 4.2 and 4.3), which permits context-dependent recurrent transitions that depend non-linearly on inputs from another region. These inputs differ from weighted inter-regional inputs in that they do not drive the target region toward a particular pattern directly; instead, they contextualize its recurrent dynamics, selecting among multiple learned associations to govern each transition. Thus, when a region executes a transition in this regime, the consequent state depends on both the initial state of that region and the pattern of activation that is used to contextualize the transition. This "functional branching" makes it possible to learn directed graphs of attractors and transitions through which a region may traverse. Chapter 3 shows that attractor graphs can efficiently represent compositional data structures such as linked lists, associative arrays, and trees. Such structures can be traversed via temporally-extended top-down control of regional and contextual gating.

Previous programmable attractor networks have relied primarily on itinerant attractor sequences without contextual gating, which restricts the space of possible programming languages that may be implemented. For example, the Neural Virtual Machine implements an assembly-like language, with programs represented as linear sequences of instructions [97]. This makes it difficult to encode complex programs that are much more easily expressed in higher-level languages as abstract syntax trees, which can be represented directly in neural memory as attractor graphs. Section 4.1.2 shows how the "cons cells" of the LISP programming language can be represented by simple attractor graphs and composed into nested expressions to represent complex programs (stored in the *mem* region). These expressions can then be recursively evaluated by other regions with simpler non-contextualized dynamics (Controller regions) that implement an assembly-like language suitable for defining LISP interpreter functions (e.g., evaluation, variable lookups, in-

put/output, etc), as described in Section 4.1.3.

4.1.2 Compositional Data Structures

Compositional data structures are implemented in NeuroLISP as systems of attractors (distributed representations) with gated transitions, called attractor graphs. The details of the dynamics underlying attractor graphs can be found in Chapter 3. This section describes how they are used to implement cons cells and associative arrays (maps), two fundamental data structures that serve as building blocks for NeuroLISP's memory system (Figure 4.3). These data structures can be constructed and accessed in neural memory via computationally-efficient gating operations with constant time and linear memory requirements.

4.1.2.1 Cons Cells

Cons cells are ordered pairs of elements that may be atomic symbols or other cons cells. Atomic symbols are represented as attractor states in the memory (*mem*) region that are associated with a corresponding pattern of activation in the lexicon (*lex*) region (Figure 4.3a). Cons cells are also represented as *mem* attractor states (Figure 4.3b), but they differ from atomic symbols in three ways. First, they are associated with a reserved *lex* pattern that identifies the memory state as a cons cell. Second, each cons cell serves as the source state for a unique sequence of transitions from the cons cell through the two elements of its ordered pair. Third, each cons cell *mem* state is associated with a unique state in the memory context region (*ctx* region adjacent to *mem*) that contextualizes the transitions linking the cons cell with its elements. This organization allows memory states to be contained as elements in several cons cells without duplication, as the transitions linking the

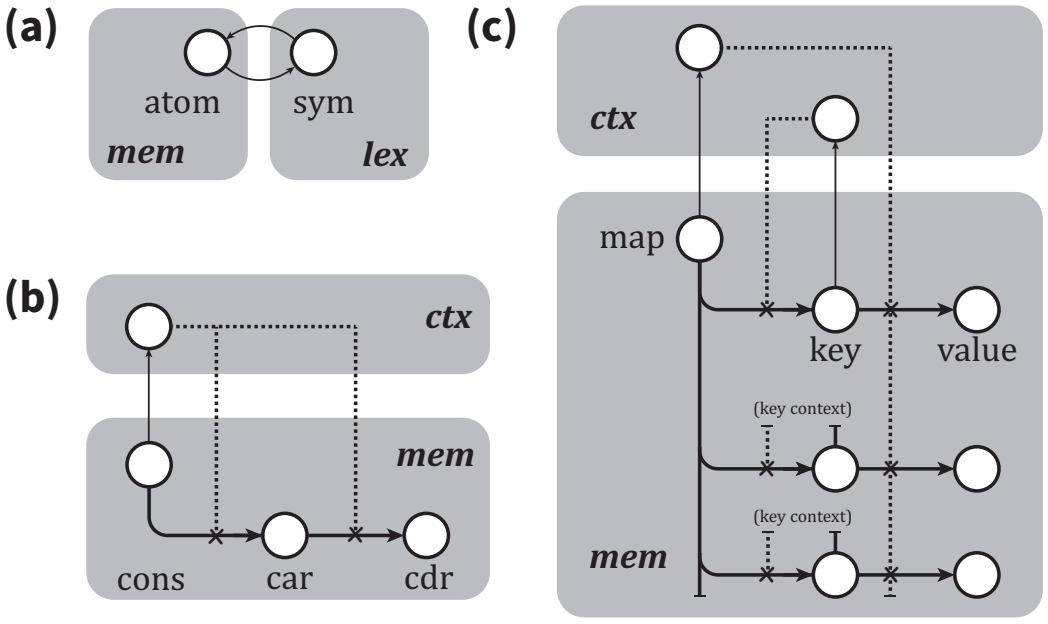


Figure 4.3: Graphical depiction of attractor graph representations of fundamental data structures (see Chapter 3). Each gray rectangle represents the activity space of a region (*mem*, *lex*, or *ctx*), and each circle represents a unique distributed neural activity pattern in that region. Solid lines indicate learned associations between states, either between or within regions. Dashed lines indicate contextual dependencies for recurrent *mem* transitions, which can only be executed when the corresponding *ctx* pattern is present. (a) Atomic symbols are represented by pairs of states in the *lex* and *mem* region. The *mem* state allows the symbol to participate in compositional structures, while the *lex* state allows the symbol to be read from or written to the environment, and interpreted as a variable name, function name, or LISP operator. (b) Cons cells are represented by a unique activity state (labeled "cons") that serves as the head of a trajectory through the elements contained in the cell (labeled "car" and "cdr"). These transitions are contextualized by a unique *ctx* state (circle within *ctx* rectangle). (c) Associative arrays (maps) are represented similarly, except that there are multiple trajectories from the head state (labeled "map") that run through each key/value pair in the map. Each transition to a key is contextualized by a unique *ctx* state associated with the key (lower right circle in *ctx* rectangle). This permits verification that a key is contained in the map prior to value lookups (see Appendix A.4). Each key state is associated with the corresponding value in the map by a unique map context state (top right circle in *ctx* rectangle). This allows the same memory state to serve as a key in multiple maps, each with a unique corresponding value. For clarity, only the context states for a single key/value pair are shown, and the remaining pairs are abbreviated (bottom of *mem* rectangle).

elements of a cons cell are contextualized by a unique multiplicative pattern (see Appendix A.3).

The car and cdr operations retrieve the first and second elements of a cons cell, respectively. These operations can be performed by iterating through the `mem` attractor sequence, starting with the cons cell attractor, and stopping at the desired element. This sequence is contextualized by the unique `ctx` state associated with the cons cell, which must be retrieved prior to iteration. To construct a cons cell, a `mem` attractor sequence must be constructed using several gates, including the noise (Equation 4.1), eligibility trace (Equation 4.4), and plasticity gates (Equation 4.6). This sequence links together a newly created cons attractor with the two elements that will be contained as car and cdr elements. The details of these operations are included in Appendix A.3.

4.1.2.2 Associative Arrays (Maps)

Associative arrays, or maps, are collections of key/value pairs. The fundamental operations of maps include addition, modification, and removal of key/value pairs, checking whether a key/value pair exists for a given key, and retrieving that value if it exists. Maps are generally implemented as hash tables in conventional computers, which use a hashing function to transform keys into unique offsets for indexing an array in linear memory. In lieu of such a hashing function, neural attractor transitions are used to uniquely associate keys with values, similarly to the above implementation of cons cells. Several unique features of maps, however, make their underlying implementation more involved than cons cells.

The organization of maps in neural memory is shown in Figure 4.3c. Each key/value pair in a map corresponds to a pair of attractor states linked by a contextualized transition. Unlike cons cells, the first element of a pair (the key) is provided during operations performed on the map (e.g.,

lookups). To retrieve the value for a given key, the context state for the map is used to execute a transition from the key's memory state to the value's memory state, much like the cdr element of a cons cell is retrieved from the corresponding car element. However, unlike with cons cells, the corresponding transition from the map to the key state is not contextualized by the map's context state, as a map may contain multiple key/value pairs. Instead, a context state associated with the key memory is used for the transition, making it possible to check whether a key is contained in a map (see Appendix A.4 for details).

4.1.3 Virtual Interpreter

NeuroLISP implements a virtual interpreter that evaluates programmatic expressions stored as nested cons cells in neural memory (Section 4.1.2.1). Interpreter functions are orchestrated by sequential activation in the Controller sub-network, which controls model functionality by opening and closing gates on model components over time (Equations 4.1 - 4.6). Much like a conventional computer, the Controller specifies pathways through the architecture for information processing based on learned instructions. This includes manipulation of memory, runtime/data stacks, input/output pathways, and environmental variable bindings, but also the functionality of the Controller itself. For example, conditional statements require the Controller to execute different procedures based on the result of operations performed on the contents of memory (e.g., comparisons, logical operations). Interpreter functionality in the Controller (i.e., interpreter firmware) is learned with fast associative learning (Equations 4.4 - 4.6) during a one-time initialization procedure.

The design of the Controller and Stack sub-networks is inspired by the Neural Virtual Machine, which implements an assembly-like language and represents programs using temporal se-

quences of distributed neural activity patterns [97]. Sequences in the *op* region represent programs in a low-level assembly-like language that implement interpreter functions for the higher-level LISP language. These low-level programs are referred to as op-sequences to distinguish them from LISP programs. The relationship between these two levels is discussed in more detail in Appendix A.5.

4.1.3.1 Evaluation

In LISP, *eval* is a core interpreter function that recursively evaluates a LISP expression and returns the result. The *eval* function contains conditional logic that determines how an expression is to be evaluated based on its contents and structure, and invokes other necessary interpreter functions. For example, an atomic expression is interpreted as a variable name to be looked up in the environment, while a list is interpreted as an application of a function or built-in operator.

The *eval* function is implemented in NeuroLISP as a central op-sequence of the Controller that branches off into one of several other op-sequences based on the currently active pattern of activity in *mem*, which represents the LISP expression to be evaluated. This begins a cascade of op-sequence calls that implements the expression's operator via top-down control of gated neural computations, and may include recursive evaluation of sub-expressions. During recursive evaluation, the Controller uses the stack regions to temporarily store `op`, `mem`, and `env` activity states by learning associations in the corresponding pathways (see Figure 4.2). For example, evaluation of a cons expression (e.g., `(cons 'a 'b)`) begins with recursive evaluation of the sub-expressions for the car and cdr elements, which are stored temporarily on the data stack and retrieved during construction of the cons cell attractor sequence (Section 4.1.2.1). Further details on recursive eval-

uation are included in Appendix A.5.

Compound expressions are stored in memory as lists represented by chains of cons cells. The first element of the list represents either a built-in operation (Table 4.1) or a sub-expression that can be evaluated to retrieve a function (e.g., a function name or lambda expression). To distinguish between these cases, all patterns in the *lex* region representing symbols for built-in operators are learned as attractors in a recurrent *lex* matrix. These patterns are recognizable via comparison (Section 4.1.3.2): if the pattern remains stable following recurrent dynamics, it represents a built-in operator. Each built-in operator has a corresponding *op* sequence that implements its operation and can be retrieved via the pathway from *lex* to *op*. When the first element of a compound expression is not a built-in operation, it must evaluate to a function, and may either be a variable naming a function that can be retrieved from the environment (Section 4.1.3.4), or a cons cell representing a lambda function (Section 4.1.3.5). In either of these cases, the operator is recursively evaluated, and the parent expression is interpreted as a call to the returned function, with the remaining elements of the list interpreted as expressions for the values of the function's arguments. When evaluating non-compound expressions (i.e., individual symbols), built-in symbols are simply returned, and other symbols are interpreted as variable names, and looked up in the environment.

Two special LISP operators provide programmatic control of evaluation: `quote` and `eval`. The `quote` operator instructs the interpreter to skip evaluation of sub-expressions and return them directly as data. Conversely, the `eval` operator instructs the interpreter to evaluate the value that was returned from evaluation of the sub-expression. These two operators allow seamless interchange of programs and data (i.e., *programs as data*), and make it straightforward to implement programs that generate other programs. The `quote` operator is implemented in NeuroLISP by an op-sequence that simply retrieves the memory state representing the expression's

argument (e.g., evaluating `(quote x)` returns the memory state representing the symbol `x`). The op-sequence implementing the `eval` operator recursively evaluates the argument sub-expression, then performs a second round of recursive evaluation on the resulting memory state, and returns the final result.

4.1.3.2 Conditional Evaluation

Conditional evaluation involves comparisons that are initiated by branching instructions in the *op* region. A comparison is performed on two activity states that occur in the same region (one of *mem*, *lex*, or *env*) at different timesteps. The result of the comparison causes the *gate sequence* region to initiate one of two operations for advancing the *op* region, analogous to the jump operations that occur in conventional computer architectures. If the compared states are within a threshold of similarity, *op* is advanced to a new sequence designated by the branch operation's operand. Otherwise, *op* is simply advanced to the next instruction in the current op-sequence.

Comparisons are performed in two stages. For illustrative purposes, consider a comparison performed between two *mem* states to determine whether a key is contained in an associative array / map (Section 4.1.2.2). First, the key memory state is retrieved, and an association is learned in the pathway from *mem* to *gate sequence* that links the key memory state with a designated *gate sequence* state corresponding to the jump gate procedure. Next, the key is used to execute a transition from the map state in *mem*, which yields the key state if the key is contained in the map, or a random state otherwise. The comparison is performed to determine which of these two cases occurred. At this point, two gates are opened in the *gate sequence* region: one that controls the pathway from *mem*, and another that activates a bias input that pushes *gate sequence* toward a

designated sequence that corresponds to a false comparison. If the current *mem* state matches the memorized state (i.e., the key memory state), then the net input to *gate sequence* will match the jump sequence state. Otherwise, the net input will be dominated by the bias term. The resulting *gate sequence* activity performs the appropriate operation for advancing *op* according to the result of the comparison: jump to the operand if the result was true, or advance to the next instruction in the current op-sequence if false. The details of the comparison operation are discussed in Appendix A.6.

4.1.3.3 Input/Output

As mentioned in Section 4.1.1, symbols represented by *lex* activity patterns can be read from or printed to the environment via control of special input/output gates. Each of these patterns is also reciprocally associated with a unique activity pattern in *mem*, which allows the symbol to serve as a component of compositional structures (Figure 4.3a). The bi-directional associations between *mem* and *lex* representations of a symbol make it possible for the model to recognize that a symbol has never been seen before, and to construct a new *mem* state to represent it. This is done by reading the symbol to *lex*, memorizing it, executing an inter-regional transition from *lex* to *mem* and back to *lex*, and comparing the resulting *lex* activity pattern to the memorized pattern (Section 4.1.3.2). If they match, the symbol has a *mem* representation already. If not, one is created, and the bi-directional associations are created. Thus, NeuroLISP's lexicon is automatically expanded as it encounters new input symbols.

A read operation parses a sequence of symbolic inputs into a data structure in memory. Two special symbols representing open and closed parentheses indicate delimiters of nested expres-

sions, which can be parsed recursively. When the open symbol is encountered, the interpreter enters a loop, recursively parsing each symbol until a close symbol is encountered. Each parsed memory, besides the close symbol, is placed in a chain of cons cells representing a list. An additional quotation symbol streamlines expression quotation; when it is encountered, the result of parsing the next symbol(s) is appended to a list containing the `quote` symbol. A read operation is completed when the expression is closed (or immediately after reading a non-delimiting symbol), and the resulting memory structure is returned to the caller. Conversely, a write operation performs a pre-order traversal of a memory structure, printing an open parenthesis upon entry to a cons cell, a close parenthesis upon exit of a cons cell, and the corresponding symbol when a leaf node is reached.

NeuroLISP begins operation in a read-eval-print-loop, in which an expression to be evaluated is read in as input, constructing a program in memory to be evaluated. The memory state that is returned from evaluation is then printed, and the loop repeats. Thus, once the model is initialized with interpreter functions, it can be programmed via environmental interactions that prompt the model to control its own plasticity, rather than by direct manipulation of the weight matrices in the model.

4.1.3.4 Environment Management

In high-level programming languages, variable bindings are maintained and updated in an *environment* that is accessible during program evaluation. Environments are typically composed of distinct *namespaces* that manage different bindings for the same variable name that are relevant to different execution contexts. For example, if a function *f(x)* calls another function *g(x)*, two distinct

bindings are maintained for x that may contain different values. When a lookup is performed, the evaluator must retrieve the correct binding based on the current execution context. This can be done *dynamically*, in which the most recent binding is retrieved, or *lexically*, in which the correct binding is determined based on the location of the expression being executed within the program. Lexical scoping is more complex than dynamic scoping because variable lookups are relative to the code being executed, and several bindings for a given variable must be maintained separately. The organization of environments is described in detail in Appendix A.7. NeuroLISP supports both dynamic and lexical scoping because the procedures for environment organization and access are determined by learned interpreter functions. Lexical scoping was used for the experiments described below.

4.1.3.5 Function Definitions and Applications

In lexically-scoped languages, function definition involves creation of a *closure* that binds together the body of the function, its argument list, and a namespace containing variable bindings that were accessible at the time of definition. In NeuroLISP, closures are stored as cons cells in the *mem* region (Section 4.1.2.1) that have special learned associations with namespaces represented in the *env* region (Section 4.1.3.4), and are associated with a reserved "function" symbol in *lex*. When a function is called, the corresponding namespace is retrieved, and a new namespace is constructed from it to store the bindings of the function's arguments. This namespace is then used for variable lookups during execution of the function body, and it contains both the argument bindings and the extant bindings from when the function was first defined (see Appendix A.8 for details). When dynamic scoping is used, the closure namespace is ignored, and the new namespace is branched

off of the caller's namespace.

4.1.4 Experimental Methods

Experiments were performed on NeuroLISP using several different programs: 1) To verify its correctness, NeuroLISP was first tested with a suite of 37 simple handwritten test cases that evaluate the various functions of the implemented language. 2) To determine the relationship between region sizing and memory capacity, the network's memory capacity was evaluated with basic programs involving list storage and variable binding. 3) To demonstrate that it can successfully execute basic LISP programs that manipulate complex data structures, NeuroLISP was tested with a small library of multiway tree processing functions. 4) To demonstrate that NeuroLISP is capable of compositional processing, it was tested with a library of sequence manipulation functions that solves the PCFG SET task described in Hupkes et al. [82]. 5) Finally, to show that NeuroLISP can perform high-level procedures that are relevant to traditional symbolic AI, it was tested with a first-order unification algorithm [173], a key component of automated reasoning, type checking, and logic programming. Empirical results from these experiments are presented in Sections 4.2.1 - 4.2.5.

To distinguish between bugs in the interpreter firmware and neurocomputational errors (i.e., corruption of learned neural associations), a non-neural emulator for the NeuroLISP architecture was implemented that faithfully reproduces the flow of information that occurs through regions in the architecture, but uses explicit symbols and lookup tables in lieu of activation patterns and weight matrices. This allowed determination of the number of associations in various pathways that would be learned during correct execution of a program, which is referred to as the program's

complexity, without the possibility of interference from neurocomputational errors. Because the memory capacity of simple attractor networks is relative to the number of neurons in the network [4, 97], experiments were designed to determine the relationship between the size of Neuro-LISP regions and the complexity of programs that it can successfully execute. Specifically, these experiments examined the number of generated memory states (attractors in the *mem* region) and the number of variable bindings (associations between *env* and *mem* states), relative to the size of the *mem*, *lex*, and *env* regions. In addition, these experiments examined the impact of the context density parameter $\lambda_{env-ctx}$, which determines how many *env* neurons participate in each variable binding.

Each experiment included several trials in which the NeuroLISP architecture was instantiated with a particular set of model parameters, initialized with the interpreter firmware, and executed with a particular set of inputs encoding a test program. The output of the model was compared to the correct reference output for the trial inputs to determine if the trial was successful or not. Experiments were performed in blocks with shared model parameters and inputs with a shared property that were not necessarily identical (e.g., testing retention of different lists of the same length). Each block contained 20 trials, and results are reported as the percentage of trials in each block that produced correct outputs (each datapoint in the plots in Section 4.2.1 - 4.2.5 corresponds to one block of 20 trials). Each model parameter was systematically varied one at a time in order to determine its impact on model performance (e.g., how does the size of the *mem* region impact list retention?). The remaining parameters were set in order to avoid degradation of performance (e.g., a large *mem* region size was used when testing the impact of *env* region sizing on variable binding). The details of model parameters used during experiments can be found in Appendix A.9.

The computational experiments presented here address the following questions:

- **Correctness**: Does the NeuroLISP firmware correctly implement the language interpreter? Can NeuroLISP successfully execute high-level programs, including multiway tree processing functions, the string manipulation functions of the PCFG SET task, and a first-order unification procedure used in automated reasoning?

- **Memory capacity**: How do the sizes of the *mem* and *lex* regions impact program/data memory storage capacity?

- **Binding capacity**: How does the size of the *env* region and its context density parameter $\lambda_{env-ctx}$ affect a) the number of variable bindings that can be stored in a single namespace, and b) the number of namespaces that can store separate bindings for the same variable name?

It was hypothesized that correct performance would require sizing the *mem*, *lex*, and *env* regions according to the complexity of the executed program. Based on results reported in Chapter 3, the following results were predicted:

- **Region sizing**: a linear relationship between a) *mem* region size and program/data memory storage capacity capacity, and b) *env* region size and the number of bindings stored for the same variable name in different namespaces.

- **Context density**: a larger *env* context density $\lambda_{env-ctx}$ would facilitate storage of bindings with the same variable name in different namespaces, but interfere with storage of several variables in one namespace. Thus, a balanced context density would lead to the best performance on complex programs involving multiple bindings across multiple namespaces.

Finally, the scalability of the model was investigated in parallel computing environments with additional experiments that evaluated runtime performance and memory usage. Experiments were performed on a 3.5GHz 10-core Xeon E5-2687W v3 with two NVIDIA RTX 3060 GPUs. The model was implemented in the Python programming language using the `numpy` library for multi-dimensional arrays and the `pyCUDA` library for GPU processing. Model computations that involved matrix operations (e.g., input activation and learning) were performed on GPU(s), while those that did not (e.g., neural activation functions) were performed on the CPU. The results of these experiments are presented in Section 4.2.6. Experiments were designed to investigate several approaches to improving both the runtime and memory efficiency of model simulation without affecting the accuracy of its symbolic behavior, as described below.

A significant advantage of the gated region-and-pathway paradigm is that only a subset of the model (corresponding to active regions and connections) must be computed during each timestep, greatly reducing the computational cost of model execution. This also means that scaling up the number of neurons in a specific region only incurs performance penalties for computations that involve that region and its associated connectivity matrices. For example, the size of the `env` region does not affect operations that do not involve variable bindings or namespaces, such as parsing inputs and constructing programs in memory. This is illustrated by separate measurements of the runtime for program parsing and execution.

Because each timestep of simulation only involves a subset of model computations, the theoretical performance benefits from distributing kernels among multiple compute devices is limited. A more effective strategy would be to split up individual regions, divvying up neurons to separate devices for distributed computation of individual kernels. This possibility is not pursued here, but it is feasible due to the locality of associative learning. Instead, weight matrices were

distributed among GPUs to take advantage of their available memory. When the size of the model exceeds available GPU memory, matrices must be shuttled between the host and GPU as they are needed during kernel execution, incurring a significant runtime performance penalty called memory thrashing. Thus, the use of multiple GPUs therefore provides additional memory and allows larger models to be simulated without intractable increases in runtime. In addition, the use of half-precision floating points for weight matrices reduces their memory footprint by 50%, allowing further increases in model size.

When a region is contextually gated, a subset of its neurons do not participate in computations, as their output is guaranteed to be zero and their corresponding weights will not be affected by learning. Thus, a naive implementation of matrix operations involves unnecessary computations when contextual gating is active, as compute threads are allocated to deactivated neurons. The percentage of wasted computations depends on the context density parameter, which is typically less than $\frac{1}{2}$ (i.e., more than half of the computations are wasted). Thus, efficient versions of the matrix operation kernels were implemented that perform preprocessing to determine which neurons are active and assign them to compute threads accordingly.

Overall, it was hypothesized that runtime would be significantly improved by the use of efficient kernels that avoid unnecessary computations for deactivated neurons. In addition, runtime was hypothesized to scale roughly with model size until GPU memory is exceeded, at which point runtime would rapidly increase due to memory thrashing. Thus, the use of half-precision weights and distribution of the model across GPUs would increase the maximum model size possible without intractable increases in runtime.

4.2 Results

4.2.1 Interpreter Test Suite

NeuroLISP was first tested with 37 simple handwritten test cases, listed in Table 4.3, that exercised the various components of the implemented language to verify the correctness of the virtual interpreter firmware. These tests include constructing and navigating s-expressions and associative arrays, reading and printing s-expressions, logical and conditional operations, function definitions, and variable bindings in nested namespaces. For each test, NeuroLISP was constructed with *mem*, *lex*, and *env* region sizes of 2048, 2048, and 1024, respectively, and an *env* context density parameter $\lambda_{env} = \frac{1}{4}$. With these parameters, the model successfully passed all tests.

4.2.2 Memory and Variable Binding Capacity

To examine the relationship between program/data memory capacity and the size of the *mem* and *lex* regions, simple tests were performed involving storage and retrieval of lists containing between 10 and 100 symbols randomly drawn from a set of 10 possible symbols. During each trial, a list of symbols was read in as input using the `read` operation, stored in memory, then printed back as output. First, the size of the *mem* region was systematically varied from 300 to 1500 neurons while keeping the *lex* region size constant at 2048 neurons. Then, the opposite was done, testing variations of the *lex* region size from 300 to 1500 while keeping the *mem* region size constant at 2048. In both cases, $\lambda_{env-ctx}$ was fixed at $\frac{1}{4}$. The results are shown in Figure 4.4, where each data point indicates the percentage of successful trials (y-axis) involving lists of a specific length (x-axis). Each line indicates results for instantiations of the model with the same

Table 4.3: Basic test suite for the NeuroLISP interpreter. The left column contains test programs, and the right column contains the corresponding expected outputs. Note that because NeuroLISP executes a read-eval-print-loop, the return value of each expression is printed alongside any outputs provided by explicit print commands. Lists (cons cells) are printed as parenthesis-enclosed expressions, while function closures and hash maps (associative arrays) are printed as #FUNCTION and #HASH, respectively.

`(cons (quote A) (cons (quote B) NIL))`	`(A B)`
`(list (quote A) (quote B))`	`(A B)`
`(quote (A B))`	`(A B)`
`(car (cons (quote A) NIL))`	`A`
`(car (cdr (cdr (list` `  (quote A) (quote B) (quote C)))))`	`C`
`(car (cdr` `  (car (cdr (quote (A (B C) D))))))`	`C`
`(cadr (quote (A (B C) D)))`	`(B C)`
`(eq 'x 'x)`	`true`
`(eq 'x 'y)`	`false`
`(eq 'x (list 'x))`	`false`
`(atom 'x)`	`true`
`(atom (list 'x))`	`false`
`(listp 'x)`	`false`
`(listp (list 'x))`	`true`
`(print (read)) A`	`A A`
`(print (list (read) (read) (read))) A B C`	`(A B C) (A B C)`
`(progn (print 'foo) (print 'bar) 'baz)`	`foo bar baz`
`(dolist (x '(A B C) x) (print x))`	`A B C C`
`(eval (quote (print (quote x))))`	`x x`
`(eval (cons 'print` `  (cdr (list 'foo '(quote x)))))`	`x x`
`(if true 'foo 'bar) (if false 'foo 'bar)`	`foo bar`
`(if (or false (and true true)) 'foo 'bar)`	`foo`

`(cond (false 'a)`
`  ((or false false) 'b)`
`  ((and true false) 'c)`
`  ((not true) 'd)`
`  ((eq 'x 'y) 'e)`
`  (true 'f))`
	`f`

`((lambda (x y) (list x y)) 'foo 'bar)`	`(foo bar)`
`((lambda (f x y) (f x y))` `  (lambda (x y) (list x y)) 'foo 'bar)`	`(foo bar)`

`((label f (lambda (x)`
`  (if x`
`  (progn (print (car x)) (f (cdr x))))))`
`  (list 'foo 'bar))`
	`foo bar NIL`

`(defun f (x y) (list x y))`
`(defun g (x y) (f x y))`
`(g 'foo 'bar)`
	`#FUNCTION`
	`#FUNCTION`
	`(foo bar)`

`(let ((x 'foo))`
`  (progn (print x)`
`  (let ((x 'bar)) (print x)) x))`
	
	`foo bar foo`

`(let ((x 'foo) (y 'bar)) (list x y))`	`(foo bar)`

`(let ((x 'foo) (y 'bar))`
`  (progn (defun f (x) (print (list x y)))`
`  (f 'baz) x))`
	
	`(baz bar) foo`

`(progn (setq x 'foo) x)`	`foo`
`(let ((x 'foo)) (progn (setq x 'bar) x))`	`bar`
`(let ((x 'foo))` `  (progn (let ((x 'bar)) (setq x 'baz)) x))`	 `foo`

`(let ((x 'foo))`
`  (progn (defun f (x) (setq x 'bar))`
`  (f 'baz) x))`
	
	`foo`

`(defun f () (setq x 'foo)) (f) x`	`#FUNCTION foo foo`

`(let ((hash (makehash))) (progn`
`  (sethash 'key1 'val1 hash)`
`  (sethash 'key2 'val2 hash)`
`  (print (and`
`  (checkhash 'key1 hash)`
`  (checkhash 'key2 hash)`
`  (not (checkhash 'key3 hash))))`
`  (print (list`
`  (gethash 'key1 hash)`
`  (gethash 'key2 hash)))`
`  (remhash 'key1 hash)`
`  (print (checkhash 'key1 hash))`
`  (sethash 'key1 'foo hash)`
`  (print (checkhash 'key1 hash))`
`  (print (gethash 'key1 hash))`
`  hash))`
	
	`true (val1 val2)`
	`false true foo`
	`#HASH`

`(progn`
`  (((lambda (le)`
`  ((lambda (g) (g g))`
`  (lambda (h)`
`  (le (lambda (x) ((h h) x))))))`
`  (lambda (f)`
`  (lambda (x) (cond`
`  (x (progn`
`  (print x)`
`  (f (cdr x))`
`  (print (car x)))`
`  (true x))))))`
`  '(a b c))`
`  'complete)`
	
	`(a b c) (b c) (c)`
	`c b a complete`

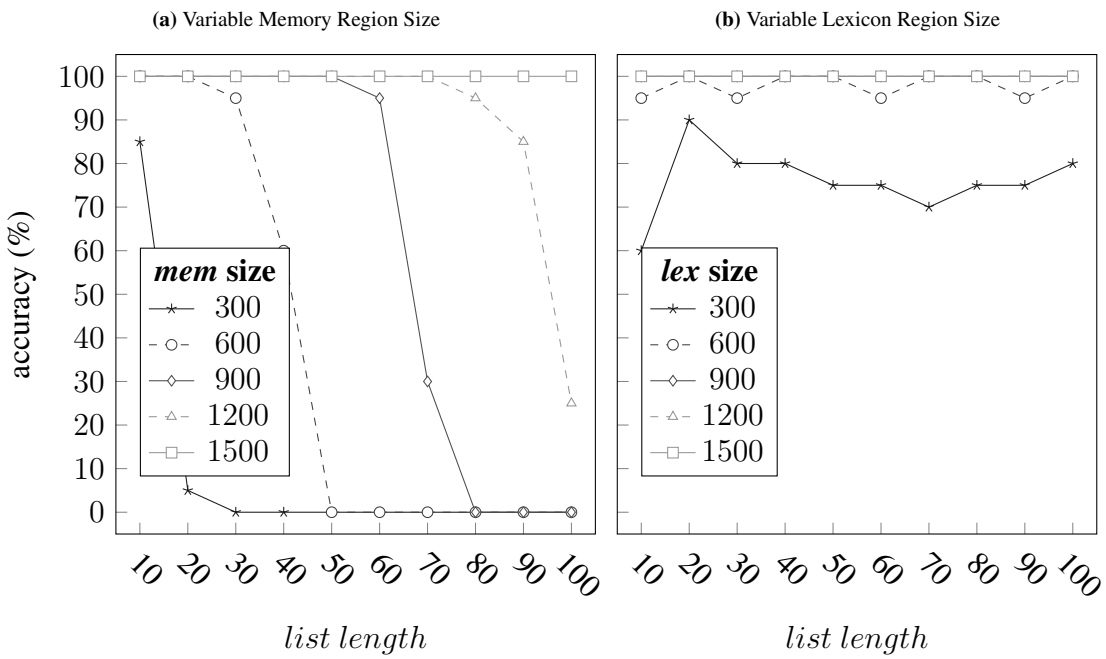


Figure 4.4: Results for list storage and retrieval with varying *mem* and *lex* region sizes. During each trial, NeuroLISP read in a randomly generated list of symbols of a specified length (x-axis) and stored it in memory, then traversed and printed its contents. A trial was considered successful if the printed list matched the input list. Each datapoint indicates the percentage of successful trials (y-axis) out of 20 for a specified list length (x-axis). (a) With the *lex* region size fixed at 2048 neurons, varying the size of the *mem* region reveals a roughly linear relationship with storage capacity (i.e., successful storage and retrieval of longer lists requires correspondingly larger *mem* region sizes). (b) With the *mem* region size fixed at 2048 neurons, the *lex* region size does not show a linear relationship with storage capacity, but a sufficient *lex* size is necessary for reliable storage. Note that because perfect accuracy is achieved for *lex* sizes of 900 and above, some lines are stacked and are not visible in the plot.

parameters. Figure 4.4a shows that, as predicted, the memory capacity of the model scales roughly linearly with the size of the *mem* region, with larger lists requiring a larger *mem* size to be reliably stored and retrieved. This can be seen by noting the gaps between the lines, and the points at which each line diverges from 100% accuracy, indicating the maximum storage capacity for a given *mem* region size (e.g., 600 neurons suffices for a list of 20 elements, 900 neurons for 50 elements, 1200 neurons for 70 elements, and 1500 neurons for 100 elements). This linear relationship is not found in Figure 4.4b, which nevertheless shows that a sufficiently large *lex* region is necessary to learn the

associations between *mem* states and the corresponding *lex* patterns representing stored symbols.

Variable binding was tested in two ways, referred to as *breadth* and *depth* testing. In breadth testing, NeuroLISP was tested with a program that created many bindings with different variable names in the same namespace (Figure 4.5a). In contrast, depth testing involved a recursive function that creates many bindings with the same variable name in different namespaces (depth refers to depth of recursion; see Figure 4.5b). These two situations were tested separately because they involve different neurocomputational demands: depth requires learning many attractors with the same context masking pattern, and breadth involves learning many attractors with the same activity pattern, but with different context masks. It was hypothesized that $\lambda_{env-ctx}$ would affect these two situations differently. Specifically, a higher $\lambda_{env-ctx}$ would improve depth performance by allowing a greater percentage of the namespace pattern to participate in attractor dynamics, but would reduce breadth performance by increasing the overlap and interference between learned attractors for different variables in the same namespace. This tradeoff was explored in Chapter 3, in which it was shown that a moderate $\lambda_{env-ctx}$ balanced performance for auto-associative and hetero-associative learning. For both breadth and depth testing, the experiment was repeated for $\lambda_{env-ctx}$ values of $\frac{1}{8}$, $\frac{1}{4}$, and $\frac{1}{2}$.

The results for breadth testing are shown in Figure 4.6. The size of the *lex* and *mem* regions was fixed at 2048 neurons and 5000 neurons, respectively, while the size of the *env* region was varied from 100 to 600 neurons. For each test, a unique program was generated to create a specific number of variable bindings in the same namespace (using the *setq* operation) before accessing them to print the corresponding values (Figure 4.5a). Each variable had a unique name, and was bound to a random symbol drawn from a set of 10 possible symbols. As predicted, a higher $\lambda_{env-ctx}$ greatly reduced performance, presumably by increasing interference between learned attractors

(a) Sample Breadth Testing Program

```
(setq v0 'B)
(setq v1 'G)
(setq v2 'F)
(setq v3 'A)
v0
v1
v2
v3
```

(b) Depth Testing Program

```
(progn
  (defun f (x)
    (if x
      (progn
        (f (cdr x))
        (print (car x)))))
  (f (read))
  'NIL)
```

Figure 4.5: Programs for variable binding capacity testing. (a) Breadth testing involves binding several variables with different names in the same namespace. The sample program shown here binds four variables to random symbols using the `setq` operation, then retrieves them sequentially. NeuroLISP prints the result of evaluating each expression, providing an output sequence to verify correct evaluation. (b) Depth testing involves binding several variables of the same name in different namespaces. The program shown here includes a recursive function that prints a list in reverse. The input list is read in using the `read` operation, and the printed output is compared with the input list to verify correct evaluation.

due to greater overlap.

The results for depth testing are shown in Figure 4.7. The size of the *lex* and *mem* regions was fixed at 2048 neurons each, while the size of the *env* region was varied from 1000 to 5000 neurons. The test program passed a random list of symbols (drawn from a set of 10 possible symbols) into a recursive function that printed the list in reverse order (Figure 4.5b). Importantly, the last symbol in the list was printed by the deepest recursive function call, and variable retrievals were only performed after all bindings were created. The results match the predicted linear relationship between *env* region size and the number of bindings that can be successfully stored and retrieved. This can be seen by noting the gaps between the lines in Figure 4.7a, and the points at which each line diverges from 100% accuracy, indicating the maximum binding capacity for a given *env* region size (e.g., 1000 neurons suffices for 10 bindings, 2000 neurons for 20 bindings, 3000 for 50, 4000 for 60, and 5000 for 80). In addition, the results corroborate the prediction of higher performance

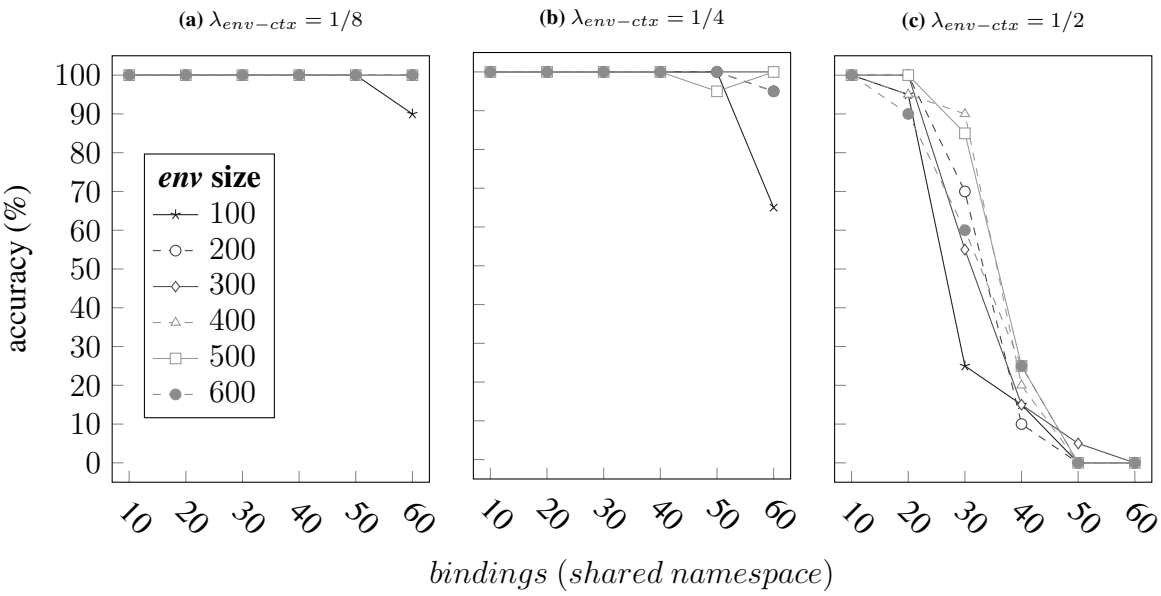


Figure 4.6: Results for *breadth* testing of variable binding with varying *env* region size and $\lambda_{env-ctx}$. Each test involved binding several variables with unique names in the same namespace using the *setq* operation, then retrieving and printing their values (Figure 4.5a). The x-axis indicates the number of stored variable bindings, and the y-axis indicates the percentage of successful trials (20 per datapoint). (a) Better performance is achieved with a low $\lambda_{env-ctx}$ of $\frac{1}{8}$, which minimizes the interference between associative learning of distinct variable bindings in the same namespace. (b) Performance deteriorates with a higher $\lambda_{env-ctx}$ of $\frac{1}{4}$. (c) With a high $\lambda_{env-ctx}$ of $\frac{1}{2}$, the model struggles to store large numbers of bindings in the same namespace, even with larger *env* region sizes. Note that some lines in the plots are stacked and are not visible.

with a higher $\lambda_{env-ctx}$.

4.2.3 Multiway Tree Processing

Given the rough guidelines for sizing the regions of the model established above, NeuroLISP was next tested with a small library of multiway tree processing functions to demonstrate that the model can successfully execute basic LISP programs that manipulate complex data structures[2]. A multiway tree is represented as either an atom (leaf node), or a list containing an atom (node

[2]The multiway tree data structure representation is thanks to Werner Hett's "Ninety-Nine Prolog Problems", and the processing functions are inspired by binary tree processing functions found in the Appendix to Paul Graham's "ANSI Common Lisp" textbook.

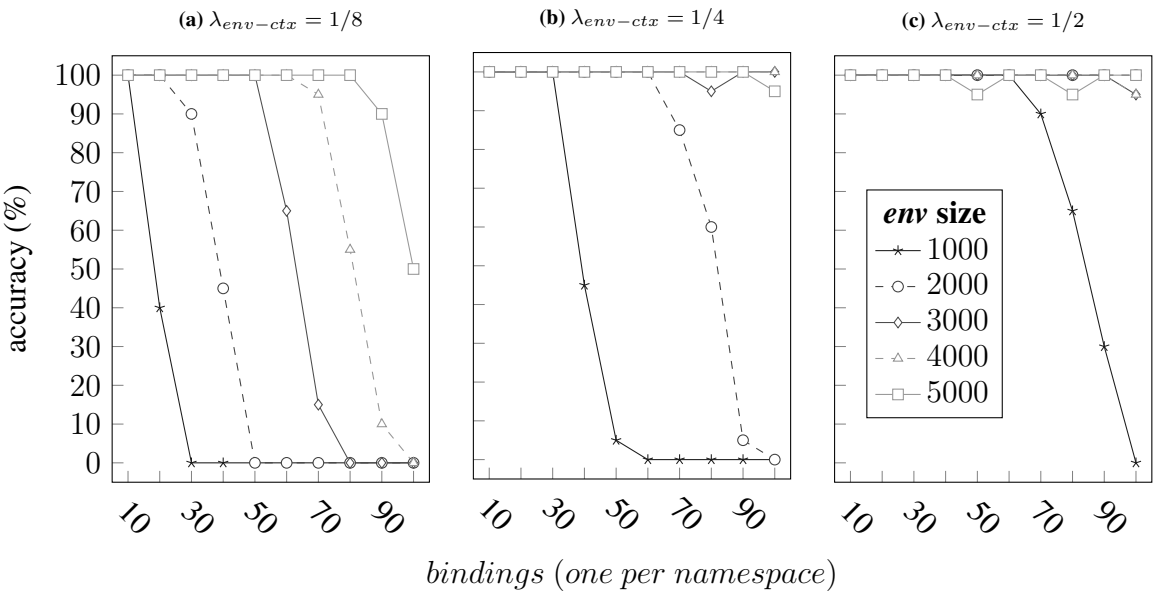


Figure 4.7: Results for *depth* testing of variable binding with varying *env* region size and $\lambda_{env-ctx}$. Each test involved execution of a recursive function with a single variable, and required storing several bindings with the same variable name in different namespaces (Figure 4.5b). Because the function uses head recursion, the model was required to store all bindings before retrieving and printing them. The x-axis indicates recursive depth (number of namespaces), and the y-axis indicates the percentage of successful trials (20 per datapoint). (a) With a low $\lambda_{env-ctx}$ of $\frac{1}{8}$, a linear relationship can be seen that resembles that of Figure 4.4a: more *env* neurons are required to store more bindings. (b) Unlike with breadth testing (Figure 4.6), a higher $\lambda_{env-ctx}$ of $\frac{1}{4}$ improves binding capacity across namespaces by increasing the number of neurons that participate in *env* region attractor dynamics and improving the discriminability of masked namespace patterns. (c) Increasing $\lambda_{env-ctx}$ to $\frac{1}{2}$ further improves performance. Note that some lines in the plots are stacked and are not visible.

label) and one or more multiway trees (children). The implemented functions are listed in Table 4.4. The full implementation and test cases for the `is-tree?` function are listed in Figure 4.8 (see Appendix A.10 for remaining functions). Each function was tested with several test cases. For each test, NeuroLISP was constructed with *mem*, *lex*, and *env* region sizes of 6000, 2048, and 1024, respectively, and an *env* context density parameter $\lambda_{env} = \frac{1}{4}$. With these parameters, the model successfully passed all tests.

Table 4.4: Multiway tree processing functions implemented in NeuroLISP.

`(expr-equal? x y)`	Recursively determines whether two s-expressions are equivalent.
`(tree? expr)`	Determines whether an expression is a valid multiway tree. A multiway tree is either an atom, or a list containing an atom and 1 or more multiway trees.
`(copy-tree tree)`	Creates a deep copy of a tree.
`(tree-member elm tree)`	Determines whether an atomic element is contained in a tree as a node label.
`(tree-prefix tree)`	Returns a list containing the node labels of a tree in prefix traversal order. The implementation is memory efficient, and only allocates the memory necessary for the final list.
`(tree-subst new old tree)`	Returns a tree that is equivalent to the input `tree`, except any subtrees matching `old` are replaced with `new`. The implementation is memory efficient: memory is only allocated for ancestors of replaced subtrees.
`(tree-sublis subs tree)`	Performs substitutions on the input `tree` using a list of `old`/`new` pairs (`subs`). As with tree-subst, the implementation minimizes memory allocation.

(a) Implementation of `is-tree?` Function

```
(defun is-tree? (expr)
  (or (atom expr)
      (and (listp expr)
           (atom (car expr))
           (cdr expr)
           (is-forest? (cdr expr)))))
(defun is-forest? (expr)
  (or (not expr)
      (and (is-tree? (car expr))
           (is-forest? (cdr expr)))))
```

(b) Test Cases for `is-tree?` Function

```
(is-tree? 'a)
(is-tree? '(a b))
(is-tree? '(a (b c)))
(is-tree? '(b d e))
(is-tree? '(a (f g) c (b d e)))
(is-tree? '(x y z))
(is-tree? '(a))

(is-tree? '(a (b c) (d) e))
(is-tree? '((a b c) (d e)))
```

Figure 4.8: (a) Implementation of the `is-tree?` function, which tests if an expression represents a valid multiway tree. (b) Test cases for `is-tree?`. The first seven test cases evaluate valid trees, while the last two evaluate invalid trees.

4.2.4 PCFG SET Compositionality Task

The PCFG SET task is a sequence processing task designed to evaluate compositional learning in machine learning models [82]. The task involves input sequences that specify nested operations performed on strings of symbols. For example:

`append swap F G H , repeat I J` ⟶ `H G F I J I J`

where the left side of the arrow indicates the input sequence, and the right side indicates the expected output sequence. This task is particularly challenging because it requires learning several operations that can be arbitrarily composed into complex expressions. Notably, Hupkes et al. [82] report empirical results demonstrating that several state-of-the-art artificial neural networks struggle to learn the task, including recurrent, convolution-based, and transformer neural networks. These models learn the task in a data-driven fashion, and are trained using large datasets of generated input/output pairs. In contrast, NeuroLISP was trained with one-step learning on LISP functions that implement each operation of the task (Figure 4.9a), and show that it can successfully compose these functions to solve input sequences encoding nested operations like the example listed above.

The NeuroLISP emulator was used to determine the complexity of PCFG SET test cases, and drew a sample covering a range of memory demands. Because tests with high memory demands require very large models, tests were filtered to include only those requiring between 250 and 350 memory states, up to 128 namespaces, and up to 64 runtime/data stack states. From the remaining tests, two samples were created. The first included 20 tests from each bin of required memory states (i.e., 20 tests requiring 250-259 memory states, 20 requiring 260-269 states, etc). The second included tests with varying numbers of variable bindings from 20-120 (i.e., 20 tests

(a) PCFG SET Functions

```
(defun append (x y)
    (if x
        (cons (car x)
            (append (cdr x) y))
        y))
(defun prepend (x y) (append y x))
(defun remove_first (x y) y)
(defun remove_second (x y) x)

(defun last (x) (dolist (e x e)))
(defun copy (x) x)

(defun reverse (pre)
    (let ((post NIL))
        (dolist (x pre post)
            (setq post
                (cons x post)))))

(defun shift (x)
    (append (cdr x) (list (car x))))

(defun swap-helper (first mid)
    (if (cdr mid)
        (cons (car mid)
            (swap-helper first
                (cdr mid)))
        (list first)))
(defun swap_first_last (x)
    (cons (last x)
        (swap-helper (car x) (cdr x))))

(defun repeat (x) (append x x))
(defun echo (x)
    (append x (list (last x))))
```

(b) Unification Functions

```
(defun var? (x)
    (and
        (listp x)
        (eq (car x) 'var)))

(defun match-var (var pat subs)
    (cond
        ((and (var? pat)
            (eq var (cadr pat))) subs)
        ((checkhash var subs)
            (unify (gethash var subs)
                pat subs))
        (true (sethash var pat subs))))

(defun unify (pat1 pat2 subs)
    (cond
        ((not subs) subs)
        ((var? pat1)
            (match-var (cadr pat1)
                pat2 subs))
        ((var? pat2)
            (match-var (cadr pat2)
                pat1 subs))
        ((atom pat1)
            (if (eq pat1 pat2) subs NIL))
        ((atom pat2) NIL)
        (true
            (unify (cdr pat1) (cdr pat2)
                (unify (car pat1)
                    (car pat2) subs)))))
```

Figure 4.9: LISP functions implementing the PCFG SET sequence manipulation operations (a) and first-order unification algorithm (b). During testing, these expressions were broken into sequences of symbols, each of which was translated into the corresponding neural activation pattern in the *lex* region, and fed into NeuroLISP one at a time (Section 4.1.3.3). These functions were invoked by additional test code (e.g., an expression encoding a PCFG SET or unification test case) that was also fed into NeuroLISP as sequential activation patterns, stored in neural memory, and executed by the virtual interpreter. Finally, the results were printed as a sequence of neural activation patterns, translated back to symbols, and compared with the ground truth of the corresponding test case to determine if the test was successful.

requiring 20-29 bindings, 20 requiring 30-39 bindings, etc). Although NeuroLISP is capable of implementing a parsing procedure, for simplicity, each test input sequence was preprocessed into a LISP expression by converting it to prefix form and adding quote operations for element sequences (e.g., "`append swap F G H , repeat I J`" becomes `(append (swap '(F G H)) (repeat '(I J)))`).

The results for the first PCFG SET tests are shown in Figure 4.10. The size of the *lex* and *env* regions was fixed at 2048 and 1024, respectively, and the $\lambda_{env-ctx}$ was set to $\frac{1}{4}$. The *mem* region size was varied from 3000 to 5500. As expected, tests requiring greater numbers of *mem* states required a larger *mem* region size. With a sufficiently sized *mem* region, the model was able to successfully pass all of the tests. Figure 4.11 shows the results of the second set of PCFG SET tests with varying numbers of variable bindings. Here, the *lex* and *mem* region sizes are fixed at 2048 and 5500, respectively, while the *env* region size was varied from 100 to 600, and $\lambda_{env-ctx}$ was tested at $\frac{1}{8}$, $\frac{1}{4}$, and $\frac{1}{2}$. The best results were achieved with a moderate $\lambda_{env-ctx}$ of $\frac{1}{4}$ (middle plot), which permitted a surprisingly large number of bindings with a small *env* region size: perfect performance was achieved for up to 120 bindings with only 500 neurons. These results support the hypothesis that a moderate $\lambda_{env-ctx}$ balances between depth and breadth requirements for variable binding, providing a reasonable capacity for many variables bound across many namespaces.

4.2.5 First-Order Unification

First-order unification is a symbolic matching process that is an integral component of automated reasoning systems such as theorem provers [173]. Two expressions containing unbound variables can be unified if there exists a set of substitutions for the variables that makes the ex-

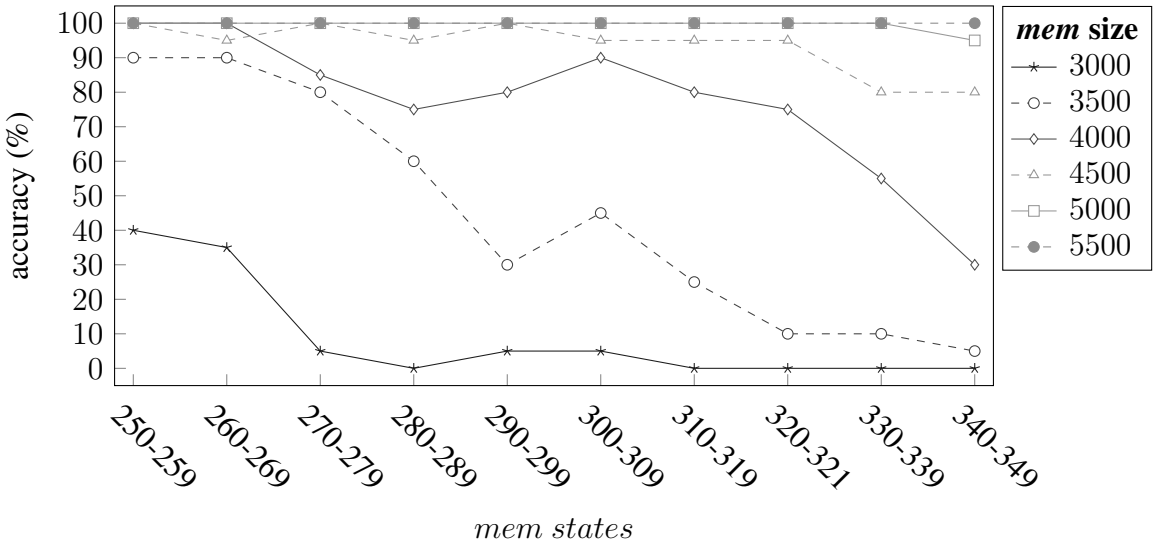


Figure 4.10: Results for PCFG SET testing with varying *mem* region size. Each test involved reading in an s-expression indicating a composition of symbolic sequence manipulations, executing the indicated functions, and printing the resulting sequence. We labeled and binned each test according to the number of memory states (*mem* attractors) it required based on an emulator for the NeuroLISP architecture, and sampled 20 tests per bin (i.e., 20 tests requiring 250-259 memory states, etc). The x-axis indicates the bin, and the y-axis indicates the percentage of successful trials for tests in each bin. As expected, model performance is contingent upon adequate *mem* region sizing; with a sufficient size, the model achieved perfect performance.

pressions equivalent. For example, unifying expressions P and Q below yields the listed set of substitutions:

P: `(f (var x) (g b))`

Q: `(f a (g (var y)))`

Substitutions: {`x` $\longrightarrow$ `a, y` $\longrightarrow$ `b`}

where `(var x)` indicates a variable named *x*, and the substitution set contains mappings from variables to values that unify the expressions. Previous work has shown that neural networks can be incorporated as components in automated reasoning systems [18, 84, 168], and that expression-specific neural networks with local representations can perform unification with error-correction learning [105]. This section shows that NeuroLISP can learn to perform first-order unification on

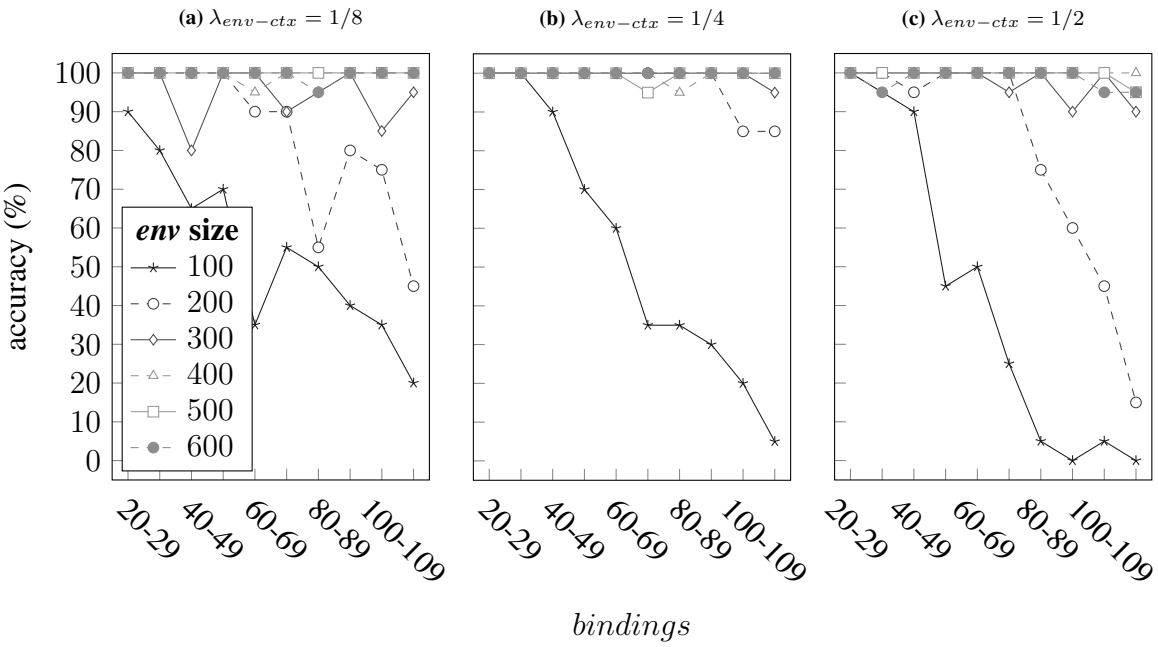


Figure 4.11: Results for PCFG SET testing with varying *env* region size and *env* context density. As in Figure 4.10, each test involved reading in an s-expression indicating a composition of symbolic sequence manipulations, executing the indicated functions, and printing the resulting sequence. We labeled and binned each test according to the number of variable bindings it required based on an emulator for the NeuroLISP architecture, and sampled 20 tests per bin (i.e., 20 tests requiring 20-29 variable bindings, etc). The x-axis indicates the bin, and the y-axis indicates the percentage of successful trials for tests in each bin. The best performance is achieved with a moderate context density of $\frac{1}{4}$ that balances the differing demands of storing several variable bindings within (Figure 4.6) and across (Figure 4.7) namespaces.

arbitrary expressions using a fixed architecture and distributed representations. NeuroLISP was trained with a unification algorithm (Figure 4.9b) based on that presented in Russell and Norvig [173], and show that it works on test cases with randomly generated nested expressions.

Unification test cases were produced by randomly generating trees and converting them to s-expressions (see Appendix A.11 for details). The complexity of these expressions was experimentally varied by varying the number of nodes in the randomly generated trees from 6 nodes to 14 nodes. Although the stochastic process introduced variations in the size of the final trees due to variable substitutions, the number of starting nodes provides a rough estimate of the complexity

of a test case. Twenty test cases were generated per initial tree size (expression complexity), and tested the model as above with a) varying *mem* region sizes and b) varying *env* region sizes and $\lambda_{env-ctx}$. In 20% of the test cases, one input expression was mutated to induce a mismatch during unification, and the model was expected to indicate that the expressions could not be unified.

Figure 4.12 shows the results of unification testing with variable *mem* region sizes. The *lex* and *env* region sizes were fixed at 2048 and 1024 neurons, respectively, and $\lambda_{env-ctx}$ was set to $\frac{1}{4}$. The *mem* region size was varied from 3000 to 4500 neurons. Perfect results were achieved with 4500 neurons. Figure 4.13 shows the results with variable *env* region sizes (100-600 neurons) and *env* context densities ($\frac{1}{8}$, $\frac{1}{4}$, and $\frac{1}{2}$). Here the *lex* and *mem* region sizes were fixed at 2048 and 5500 neurons. As with the PCFG SET tests, a relatively small *env* size was sufficient for accurate performance on relatively complex test cases. However, the best results were achieved with a smaller $\lambda_{env-ctx}$ of $\frac{1}{8}$. This may be caused by differences in the PCFG SET and unification programs: the string manipulation functions of the PCFG SET required deeper recursion, and therefore suffered more from smaller $\lambda_{env-ctx}$. These results highlight the trade-off involved with $\lambda_{env-ctx}$: the optimal parameter value depends on the demands of the programs that the model is running.

4.2.6 Runtime Performance and Scalability

While the above experiments specifically address model performance in terms of symbolic behavior, experiments in this section evaluate how region sizing affects runtime and memory usage, and investigate approaches to improving the efficiency of its implementation. The model was tested with varying `mem` and `env` region sizing using a program that includes a simple recursive

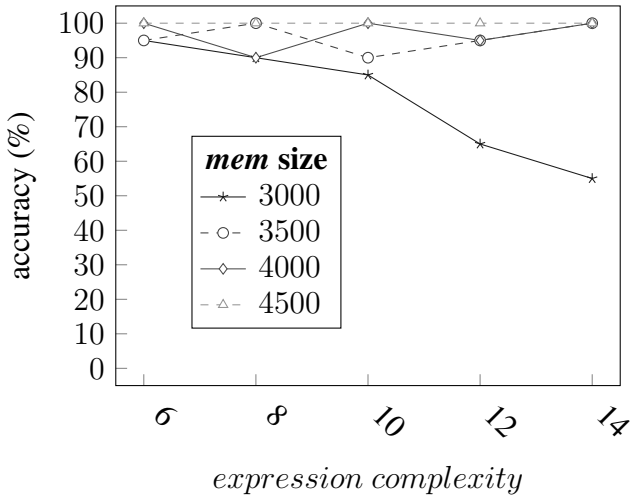


Figure 4.12: Results for first-order unification testing with varying *mem* region size. Each test involved reading in two s-expressions representing patterns with variables, performing unification on the patterns, and printing the resulting substitutions if the unification was successful. The expressions for each test case were randomly generated using an initial complexity parameter (x-axis; see Appendix A.11). The y-axis indicates the percentage of successful trials for each complexity parameter setting (20 trials per datapoint). Successful execution required a sufficiently sized *mem* region to meet the memory demands of unifying complex expressions.

function (Figure 4.14). This program is first parsed in its entirety, requiring no modifications to variable bindings or namespaces, and therefore no computations that involve the `env` region and its connectivity. After parsing, the program is executed, which involves both memory access and the utilization of variable bindings and namespaces. These two runtimes are reported separately to show that scaling a region only affects the runtime performance of relevant model computations.

Tests were performed using `mem` region sizes varying from 10,000 to 60,000 (`env` size fixed at 10,000) and `env` region sizes varying from 10,000 to 70,000 (`mem` size fixed at 10,000). The `lex` region size was fixed at 2048, and the context density parameters for the `mem` and `env` regions was set to $\frac{1}{4}$. These tests were repeated using different implementation configurations:

- One or two GPUs. For two GPUs, the weight matrices for the model were distributed between GPUs using a greedy algorithm, in which the next largest matrix is assigned to

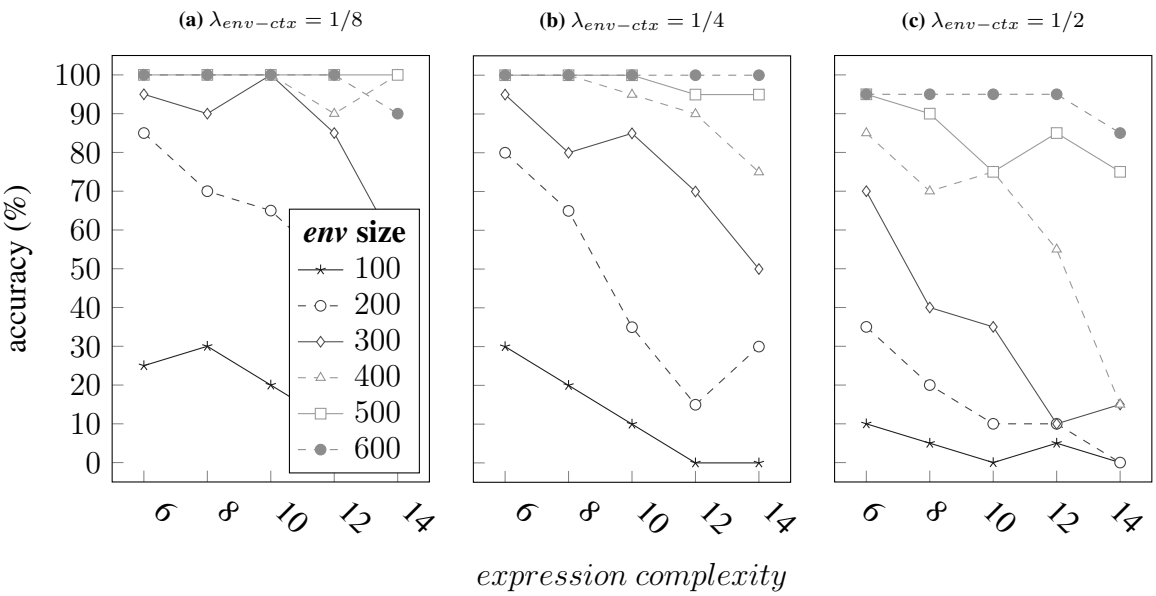


Figure 4.13: Results for first-order unification testing with varying *env* region size and *env* context density. As in Figure 4.12, each test involves performing unification on randomly generated expressions and printing substitutions if unification succeeds. The x-axis indicates the complexity of the test expressions (see Appendix A.11), and the y-axis indicates the percentage of successful trials per complexity parameter setting (20 trials per datapoint). Unlike with PCFG SET testing, slightly better performance is achieved with a low context density of $\frac{1}{8}$. Because low context densities favor larger numbers of bindings within a namespace (Figure 4.6), this may be caused by differences between the PCFG SET and unification algorithms: the former requires deeper recursion with fewer variables per namespace than the latter.

```
(progn
    (print 'executing)
    (defun f (x)
        (if x (f (cdr x))))
    (f '(a b c d e f g h i j))
    'complete)
```

Figure 4.14: Program used to evaluate runtime and memory performance. NeuroLISP first parses the entire program and stores it in memory, which requires no modifications to variable bindings or namespaces, and therefore no computations that involve the `env` region. Then, it evaluates the program, executing a recursive function that creates several namespaces and variable bindings. The timing of the printed outputs ("executing" and "complete") indicates the runtimes of parsing and execution, respectively.

the GPU with the most available memory. Matrix operations for a connection were executed on the GPU holding its weight matrix, and connections were computed one at a a time.

- Single (four byte) or half (two byte) floating-point precision for connection weights. Half-precision weights did not noticeably impact the symbolic behavior of the model, but cut memory usage in half. Thus, larger models could be executed without memory saturation.

- Simple (slow) or efficient (fast) kernels for contextually-gated connection computations. Fast kernels use preprocessing to assign only active neurons and synapses to compute threads, while slow kernels use a naive implementation that does not skip deactivated neurons and synapses.

Raw runtime and memory usage is reported in Figure 4.15. Program parsing and execution are reported separately in the first and second rows. Runtime scales with the size of the model until GPU memory is saturated, at which point runtime spikes significantly due to memory thrashing. This can be seen in the parsing runtime plot (top left); runtime spikes at different points depending on the number of available GPUs and the floating-point precision used, as these affect the maximum available memory and the total memory used by the model. As expected, parsing runtime was not affected by `env` region sizing (top right), as the corresponding neurons and weights are not needed during parsing. Thus, although memory for `env` region connections exceeds the maximum available GPU memory, it is not accessed during parsing, and does not affect runtime. During execution, however, runtime scales with both `mem` and `env` region sizing (middle row). The third row indicates memory usage, which only differs with floating-point precision. The total available GPU memory is indicated by the dotted horizontal lines (12000 megabytes (MB) for one GPU and 24000MB for two GPUs). The points where memory usage crosses these lines correspond to the

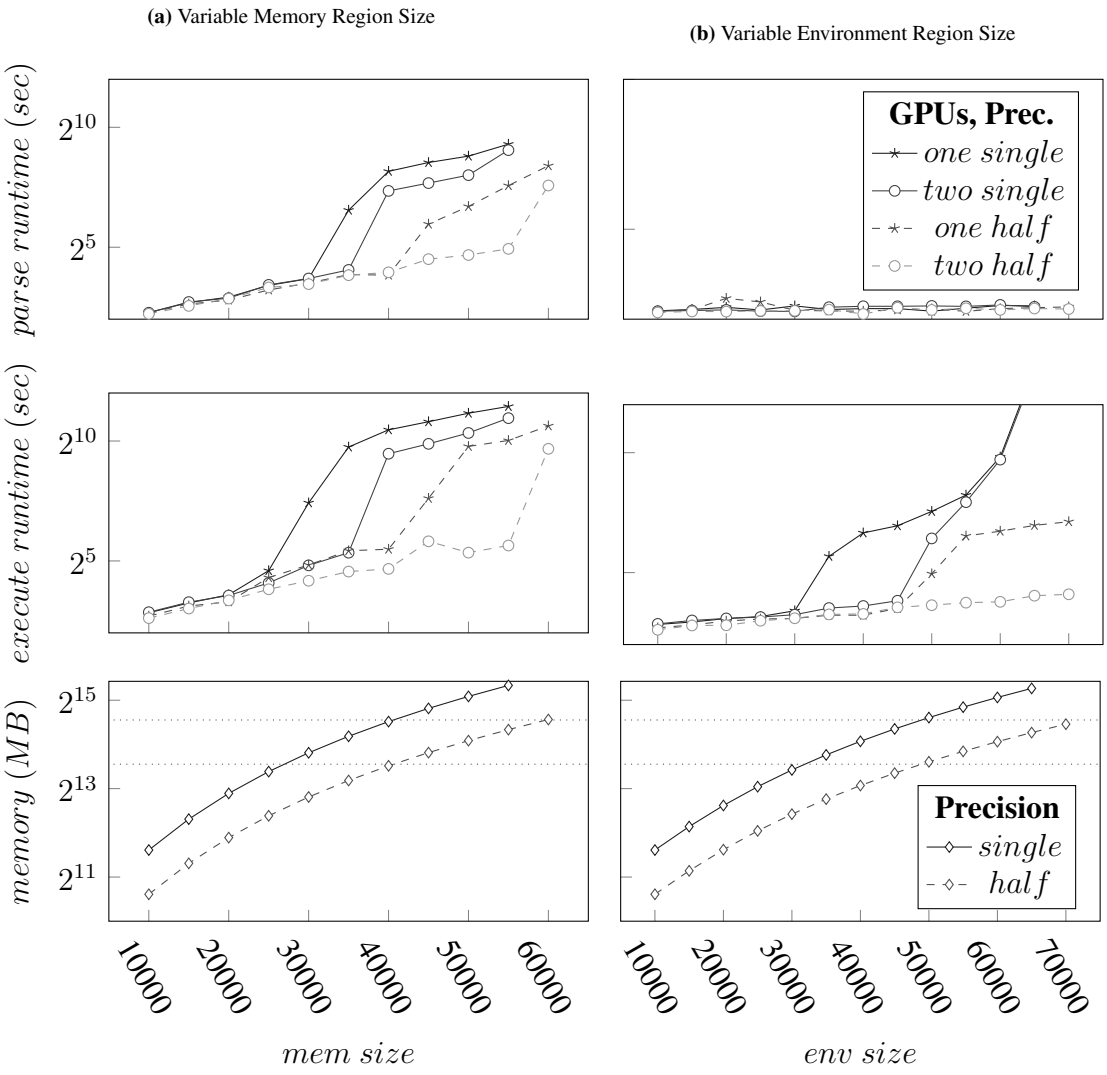


Figure 4.15: Model runtime and memory usage (y-axes) with increasing `mem` and `env` region sizing (x-axes). Each line indicates performance for a particular combination of GPU count and floating-point precision for connection weights. Solid lines indicate single-precision weights (four bytes per weight) and dashed lines indicate half-precision weights (two bytes per weight). Each precision configuration is tested with both a single and dual GPU setup. The first row shows runtime for program parsing, which does not involve `env` region computations. Thus, as the `env` region is scaled, parsing runtime does not change, as the corresponding weights are not needed in GPU memory. The second row indicates runtime for program execution, which involves both `mem` and `env` region computations. Runtime scales relative to model size until GPU memory is saturated, at which point runtime increases significantly due to memory thrashing (see Figure 4.16). Total available GPU memory is indicated by dotted horizontal lines in the third row plots (12000MB for one GPU or 24000MB for two GPUs).

spikes in runtime in the top two rows; runtime spikes at 12000MB with single GPU configurations and 24000MB with dual GPU configurations.

Figure 4.16 shows total runtime (parsing and execution combined) relative to memory usage, reported as seconds per MB. Relative runtime slightly decreases as model size increases, likely

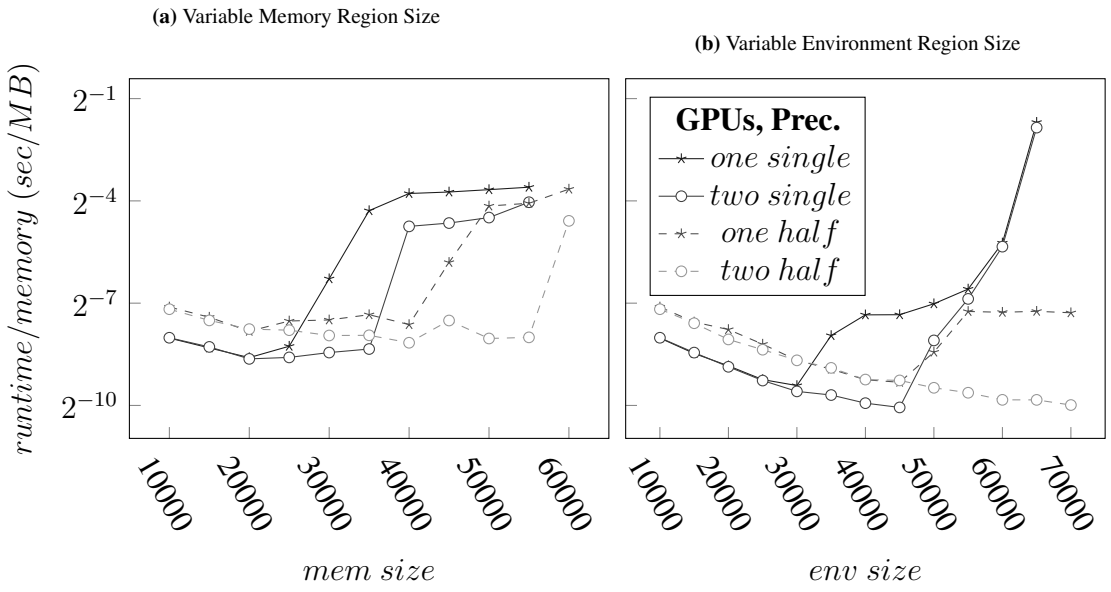


Figure 4.16: Model runtime relative to memory usage (y-axis) with increasing `mem` and `env` region sizing (x-axes). As in Figure 4.15, each line indicates performance for a particular combination of GPU count and floating-point precision for connection weights. Here the total runtime, including both program parsing and execution, is reported relative to memory usage (seconds per MB). Relative runtime decrease slightly as the overhead of CPU computations and CUDA kernel dispatch is washed out by increasingly large sizes for connection weight matrices. Once GPU memory is saturated (see third row of Figure 4.15), performance degrades significantly, and each weight incurs a more significant penalty on runtime.

because the overhead of CPU computations and CUDA kernel dispatch is washed out by increasingly large kernel execution times. The rate of decrease is more significant with `env` region scaling (right plot) because `env` region computations are much less common than `mem` region computations. Once memory is saturated, relative runtime increases sharply in both plots. Although relative performance appears to level off, single-precision performance for high `env` region sizing (right

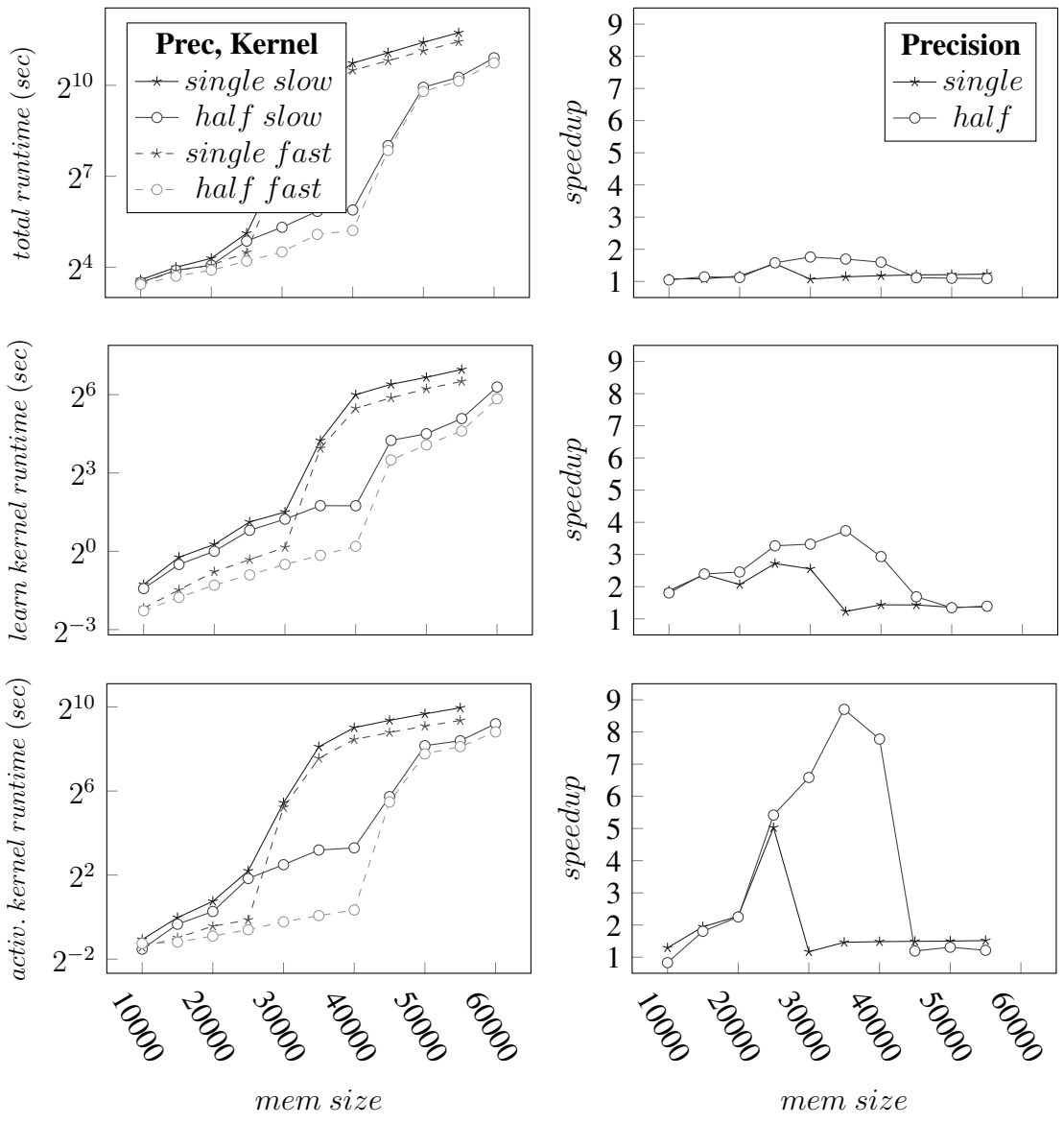
side) indicates that the plateau does not persist; once the model becomes large enough, the relative performance quickly reaches intractable levels, and the GPUs spend most of their time performing memory transfers.

Finally, Figure 4.17 show the performance benefits gained by the use of fast kernels for contextually-gated connection computations. Results are reported for single-GPU configurations with either single or half precision weights and either slow or fast kernels. The first row shows the total runtime (top left) and the corresponding speedup gained from using fast kernels over slow kernels (top right). Because not all computations involve contextual gating, the overall performance gain is relatively low, and peaks near 2x for moderately sized models. Thus, results are also reported for hetero-associative recurrent computations of the `mem` region, which involves contextual gating. The second row shows runtime (middle left) and speedup (middle right) for learning in this connection, which involves both reads and writes for weights. Speedup peaks near 4x with half-precision weights, which corresponds to the fraction of neurons activated by contextual gating. The third row shows runtime (bottom left) and speedup (bottom right) for input activation in this connection, which only involves reads for weights. Here the peak speedup reaches just over 8x for half-precision weights. These results indicate that memory access is a major bottleneck for performance, and that contextual gating provides significant performance increases. This suggests that more widespread use of contextual gating in the model would provide greater increases in performance.

Figure 4.17: (next page) Performance benefits from using efficient matrix computation kernels for contextually-gated connection computations. As in Figures 4.15 and 4.16, each line indicates performance for a particular combination of GPU count and floating-point precision for connection weights. Solid lines indicate simple kernels that naively allocate deactivated neurons and weights to compute threads, while dashed lines indicate efficient kernels that perform preprocessing to assign only active neurons and weights (see Section 2.4 for details). Note that the context density parameter for these tests was set to $\frac{1}{4}$, and contextually-gated connection computations therefore involve only $\frac{1}{4}$ of the neurons in the target region, and $\frac{1}{16}$ of the total incoming weights. The first row indicates the overall runtime (top left) along with the relative speedup gained by using efficient kernels (top right). Because only a subset of connection computations involve contextual gating, the overall speedup remains relatively low, peaking near 2x for moderately sized models. The second row indicates the cumulative runtime for learning in the hetero-associative recurrent connection of the `mem` region (middle left) and the corresponding speedup (middle right). Speedup peaks near 4x for moderately sized models. Finally, the third row indicates cumulative runtime for activation in the hetero-associative recurrent connection of the `mem` region (bottom left) and the corresponding speedup (bottom right). Unlike learning, which involves both reading and writing of connection weights, activation only involves reads, and is therefore less intensive. As a result, speedup peaks just above 8x for moderately sized models.

(a) Runtime with Fast and Slow Kernels

(b) Speedup from Fast Kernels



4.3 Discussion

This chapter presented NeuroLISP, a multi-region recurrent neural network that implements a virtual interpreter for a dialect of the LISP programming language. The network's architecture is composed of program-independent circuitry that learns both interpreter functions and LISP programs using one-step associative learning, and is capable of flexible reprogramming without architectural changes. To my knowledge, this is the first effort to implement a high-level functional programming language in a fixed neural architecture with distributed representations. NeuroLISP is most closely related to previous programmable attractor neural networks like the Neural Virtual Machine [97] and GALIS [191], which include program-independent circuitry, distributed representations, and local one-step associative learning. However, NeuroLISP includes several novel features that improve its computational capabilities. Most significantly, NeuroLISP implements an interpreter for a high-level programming language (LISP) that supports nested expressions rather than an assembly-like language with sequential instructions. This makes it easy to express complex algorithms like those used in traditional symbolic AI. The network's shared program/data memory region stores compositional data structures as attractor graphs (Chatper 3) that can be constructed, manipulated, and accessed via top-down gating during program execution. Notably, this includes programs themselves; programs can be treated as data, and generated in memory by other programs or by sequential inputs that represent code. Finally, NeuroLISP supports function definitions and variable binding with scoping rules that are determined by the learned interpreter firmware, facilitating program modularity and reuse.

Compared with other programmable neural networks, NeuroLISP is highly flexible and extensible due to its program-independent architecture and fully distributed representations. New

programs and interpreter functions can be learned with fast associative learning without adding new neurons/connections or retraining the model on previously learned behaviors. This is in contrast to approaches that involve compiling programs into specialized neural circuits with local representations [33, 134], storing programs in segregated sub-populations of a RAM-like memory matrix [30, 156], or performing neural program induction with iterative gradient descent learning [70, 71, 211]. NeuroLISP also features purely neural mechanisms for procedures that are sometimes performed by non-neural components in hybrid systems, such as call-stack management [156], storage and manipulation of structured memories [32, 179], and coordinating the flow of information through and between neural circuits [5]. All of these mechanisms in NeuroLISP are controlled by learned activity in the Controller regions of the model, and can therefore be reprogrammed in various ways to modify the virtual interpreter without architectural changes.

NeuroLISP also differs from prior models in that its working memory is based on learned attractor dynamics rather than persistent activity patterns, and it achieves temporal locality without specialized RAM-like memory matrices. Models based on neural attention are typically provided with simultaneous access to a sequence of input patterns (temporal non-locality) [24, 201], and/or selectively read and write to a large array of neurons that maintain activity patterns in segregated neural "addresses" [71, 156, 175], neither of which is considered biologically plausible. In contrast, NeuroLISP processes inputs one at a time, stores them in memory using fast associative learning, and retrieves them through top-down control of attractor dynamics. This means that NeuroLISP's working memory does not require copying activity patterns between neural regions, or complex mechanisms for memory allocation, garbage collection, or indexing schemes for data structures (e.g., usage vectors, temporal link matrices; Graves et al. [71]). Instead, new memories are created by randomly generating activity patterns, establishing them as attractors using

one-step auto-associative learning, and linking them directly to other memories using one-step hetero-associative learning (i.e., attractor graphs). The organization of data structures in memory is based on learned algorithmic behaviors contained in the interpreter firmware. Thus, NeuroLISP features improvements to both the static and dynamic aspects of working memory: its memory region natively supports attractor graphs, and its Controller regions learn specific procedures for organizing them into compositional data structures.

The computational experiments reported here demonstrate the correctness of the NeuroLISP interpreter and show that it can learn to successfully execute several non-trivial programs, including functions that operate on complex derived data structures (multiway trees). The results for the PCFG SET task show that NeuroLISP readily learns string manipulation operations that can be composed into nested expressions, a significant challenge for state-of-the-art neural networks, including recurrent, convolution-based, and transformer networks [82]. Results also show that NeuroLISP can successfully implement first-order unification, a high-level symbolic AI task that is an integral component of automated reasoning systems [173]. To my knowledge, unification with neural networks has only previously been attempted using expression-specific architectures and local representations [105], whereas NeuroLISP learns to unify arbitrary expressions using a fixed architecture and distributed representations. In accordance with prior work on the memory capacity of attractor neural networks [4, 97], the results show that the model's storage capacity for data structures and variable bindings is linearly dependent upon the size of its memory regions.

Performance testing with parallel processors indicates that simulation runtime scales with memory usage until penalties are introduced by memory thrashing, which occurs when available device memory is exceeded. Because NeuroLISP's architecture is modular, its components can be effectively distributed to different compute devices. In addition, the use of half floating-

point precision over single-precision reduces memory usage without affecting symbolic behavior, allowing larger region sizes to be simulated on the same hardware. Finally, contextually-gated connections can be computed more efficiently by allocating only active neurons and synapses to compute threads, and improves runtime efficiency in a way that is similar to architectural modularity. This suggests that more widespread use of contextual gating might lead to further performance gains by reducing the computational load of each simulation timestep.

NeuroLISP is not meant to be a veridical model of the human brain, but several aspects of its architecture and dynamics are inspired by neuroanatomy. Its "region-and-pathway" architecture [191] is inspired by the organization of the cerebral cortex, and includes several recurrent regions with heterogenous functions. Interactions among these regions are controlled by top-down gating signals that are reciprocally dependent upon regional dynamics. This resembles the functional dynamics of the basal ganglia, which are guided in part by cortical activity and provide top-down control of cortical interactions via modulation of the thalamus [52, 142]. In particular, the *gate sequence* and *gate output* regions of NeuroLISP's Controller subnetwork might represent striatal and pallidal circuitry, while the remaining regions represent various subregions of the prefrontal cortex. Although NeuroLISP does not include sensory or motor circuitry, it could be readily extended to include regions representing sensory and motor cortices, along with corresponding subcortical regions such as the tectum, cerebellum, and motor thalamus.

4.3.1 Limitations and Future Work

NeuroLISP is limited by its short-term memory capacity and lack of long-term memory. If memory becomes overloaded, learned programs are prone to corruption as memory is updated

during program execution and subsequent learning of new programs. This phenomenon, known as *catastrophic forgetting*, is a pervasive issue in artificial neural networks that affects both short-term and long-term memory [100, 143, 163]. Although NeuroLISP is subject to catastrophic forgetting, its memory retention may be improved by enrichment of synaptic structure and behavioral strategies. For example, memories may be refreshed via rehearsal [7] and subject to representational drift to reduce interference between memories [172]. On a structural level, synapses may be augmented with history-dependent transitions in plasticity, as in cascade models [65], or multiple interacting weights with heterogeneous time constants and learning rates [22, 104].

Another limitation of NeuroLISP is that it currently deals exclusively with symbolic processing, and lacks circuitry for low-level perception and action. Prior work has shown that sensory-motor circuits are readily incorporated into programmable neural networks [46, 191], which provide the top-down control typically afforded by non-neural symbolic algorithms in hybrid models. Future work might therefore involve the addition of sensory and motor networks that allow the model to run on robotic hardware that interacts directly with realtime multi-modal environments. The flexibility of the model makes such an extension straightforward, as it only requires additional gating neurons for new regions and pathways, as well as new learned interpreter functions for sensory attention and motor control.

Finally, NeuroLISP currently only learns to execute human-authored programs, and does not learn directly from input/output examples (i.e., program induction). However, its shared program/data memory space makes it possible to learn algorithms for inducing and synthesizing programs directly in neural memory, and more generally suggests future work on alternative learning paradigms such as neuroevolution [182, 187], reinforcement learning [98, 99], and imitation learning [34, 95, 116, 139]. These approaches may benefit from the addition of sensory and motor

circuitry, and contribute to stabilizing memory and resolving catastrophic forgetting.

Despite its limitations, NeuroLISP provides a promising framework for future research. Its implementation of a high-level programming language with compositional expressions makes it much easier to encode complex behaviors that are difficult to express in low-level assembly languages. Thus, NeuroLISP can replace the non-neural components of hybrid models to create purely neural systems that integrate high-level cognitive reasoning with the low-level processing that traditional neural networks excel at. Such hybrid models include neural-guided search algorithms as well as hybrid robotic learning systems that use symbolic reasoning to guide neural sensory-motor processing [95, 160]. Conversion to purely neural modeling facilitates future work on leveraging neural learning to adapt and refine cognitive algorithms based on experience.

4.3.2 Conclusions

In conclusion, this chapter demonstrates that high-level programming constructs can be incorporated into neural models to significantly advance their cognitive abilities. NeuroLISP presents a proof of concept that neural networks can implement cognitive algorithms that are typically implemented using symbolic programming techniques, including compositional logical reasoning. This model is therefore an effective neurocognitive controller that can replace the non-neural components of hybrid models, promoting seamless integration of top-down cognitive control with the strengths of contemporary machine learning. NeuroLISP's working memory system is based on biologically-inspired principles such as temporally-local control of dynamical attractors (distributed representations), fast associative learning, and top-down gating, and is therefore also relevant to interdisciplinary researchers in neuroscience and cognitive science as well

as artificial intelligence. Future work should address the control of sensory-motor dynamics in cognitive-robotic systems, as well as experience-based adaptation and refinement of cognitive procedures using methods such as reinforcement learning, program synthesis, and imitation learning. The last of these is addressed in the following chapter.

5

Hypothetico-Deductive Causal Imitation Learning

The previous two chapters presented a principled framework for implementing high-level cognitive algorithms with attractor neural networks. Chapter 3 showed how compositional data structures can be stored as *attractor graphs*, systems of interconnected dynamical attractors in recurrent neural networks. This compositional working memory was incorporated into a programmable neural network that performs basic hierarchical planning. Chapter 4 presented *NeuroLISP*, a purely neural interpreter for a subset of Common LISP that is capable of learning algorithms for compositional sequence manipulation and symbolic pattern matching. These abilities are key ingredients for causal imitation learning, which is addressed in this chapter.

As previously mentioned, imitation learning is a fundamental human ability that emerges early in life but remains a natural and intuitive method for acquiring new skills throughout the lifespan [90, 129]. Human-level imitation learning involves not only replicating observable motor behavior, but also inferring the underlying goals and intentions of the demonstrator. This allows learners to generalize demonstrated skills to novel environments by abstracting away details that are circumstantial to the demonstration environment.

Programming robots to carry out complex tasks in a human-like fashion is difficult and typically requires laborious programming by an experienced roboticist. Robotic imitation learn-

ing provides a solution to this problem, and makes robotic programming easy and accessible to non-experts [26, 83, 155, 176]. However, most work in robotic imitation learning focuses on reproducing overt motor activity, which affords only limited generalization [155]. Developing more human-like imitation in robots requires algorithms for reasoning about observed actions to construct a deeper understanding of the demonstrator's goals and intentions that can be adapted to novel environments. This approach also provides a common framework for reasoning about human and robot behavior, which facilitates an understanding of roles and perspectives that promotes seamless human-robot collaboration [197].

As discussed in Chapter 2, CERIL is a previously developed robotic imitation learning system that uses abductive inference to construct causal interpretations of demonstrated motor behavior and generalizes them for imitation in novel environments [95]. While effective and provably correct, CERIL's algorithms are implemented with traditional non-neural symbolic programming and have a limited degree of cognitive plausibility. To infer intentions, CERIL uses a bottom-up dynamic programming algorithm that exhaustively enumerates plausible causal explanations, which places unrealistic demands on working memory. It also requires multiple passes through a demonstration, whereas human imitators reason about demonstrated behavior as it occurs to construct partial explanations before a demonstration is complete. Finally, it is unclear how this approach might be implemented using neural networks to leverage the unique advantages of neural computation, such as its capacity for learning and generalization, and provide insight into the neurobiological foundations of human imitation learning.

To explore whether it is viable to develop purely neural controllers for social robotic systems that behave in a human-like manner, this chapter presents NeuroCERIL, a programmable neural network that learns human-like algorithms for causal inference during imitation learning (left side

of Figure 2.1). NeuroCERIL is evaluated using CERIL as a target system, as it has been demonstrated to be an effective cognitive controller for bimanual robots. To address CERIL's limitations in cognitive plausibility, NeuroCERIL implements a novel causal inference algorithm based on the hypothetico-deductive approach, an influential model of diagnostic and scientific reasoning [114, 115, 122, 158, 181]. Hypothetico-deductive reasoning involves a combination of bottom-up abductive inference and top-down predictive verification, which obviates the need for exhaustive search by focusing cognitive processing on relevant causal knowledge. NeuroCERIL's cognitive processes are therefore much more human-like than CERIL's, and they are supported by neuro-computational mechanisms that more closely resemble those used by people during cause-effect reasoning.

Empirical results show that NeuroCERIL is potentially an effective neurocognitive controller for robotic imitation learning systems, as it is able to reproduce CERIL's performance on a battery of demonstrations of procedural maintenance tasks. Examination of NeuroCERIL's runtime and memory usage during causal inference shows that they scale roughly linearly with the length of the demonstration. Further, many of its memories have very short lifetimes, and are only accessed during a narrow window of processing. Thus, like human working memory, many of its short-term memories are rapidly abandoned, and only a small fraction of its memories need to be maintained through the duration of a demonstration.

5.1 Methods

NeuroCERIL[1] is a brain-inspired cognitive model that learns procedural skills from demonstrations using cause-effect reasoning. The model's architecture is an extension of NeuroLISP,

[1] `https://github.com/vicariousgreg/neuroceril`

a programmable neural network that can store and evaluate programs written in a subset of the Common LISP programming language (Chapter 4). NeuroCERIL is programmed with a novel causal inference algorithm based on hypothetico-deductive reasoning, which combines bottom-up abductive inference with top-down deductive prediction and verification. This approach allows NeuroCERIL to anticipate future behavior and focuses cognitive computations on plausible explanations for observed behavior.

Although NeuroCERIL is implemented using attractor neural networks, its distributed neural computations represent algorithmic procedures performed on symbolic data structures. It is therefore convenient to begin by describing its behavior in terms of symbolic information processing, beginning with the robotic imitation learning domain in which NeuroCERIL operates (Section 5.1.1), and the algorithms and data structures that it uses to implement hypothetico-deductive causal inference (Section 5.1.2). Then follows a presentation of the neurocognitive architecture that learns to represent and evaluate these algorithms and data structures using only neural computations (Section 5.1.3). Finally, Section 5.1.4 describes the empirical experiments conducted to validate NeuroCERIL, including a battery of test demonstrations that was used to validate CERIL. Results show that NeuroCERIL performs comparably to CERIL, but that its iterative hypothetico-deductive approach is memory efficient and scales well to long demonstrations.

5.1.1 Robotic Imitation Learning Domain

NeuroCERIL operates in the robotic imitation learning domain designed for CERIL, which involves a bimanual robot (Baxter) learning procedural maintenance tasks [95]. As previously mentioned, a teacher demonstrates these tasks using SMILE, a simulated 3D environment that

allows users to interact with virtual objects such as blocks, drawers, switches, and screw valves [80]. SMILE also includes a simulation of the Baxter robot, shown in Figure 2.2 with a variety of simulated objects. SMILE greatly simplifies the low-level sensory processing involved in recognizing and segmenting actions and objects, allowing focus on the higher level cognitive processing that occurs during imitation. When a user is finished recording a demonstration, SMILE produces a transcript containing the sequence of recorded actions, along with a record of changes that occur in the environment, such as changes in object state or location.

Actions are encoded as discrete events with free parameters that refer to objects or locations in the environment. For example, grasping a `red-block` with the `left-gripper` is encoded as:

```
grasp<red-block, left-gripper>
```

The identifier `left-gripper` refers to the demonstrator's left hand, and `red-block` refers to an object in the environment, which is encoded as a collection of named properties:

```
{id:red-block, type:block, color:red, location:loc}
```

Once the block is grasped, its `location` property is updated to `left-gripper` to indicate that it is currently located in the demonstrator's left hand. This change is represented as a record containing the object identifier, property name, and the new property value:

```
(red-block location left-gripper)
```

Once the block is moved and placed, this property is updated again to reflect its new location. Although locations are encoded as discrete symbols, they can be associated with representations of 3D points in continuous space for use in low-level motor planning.

Like CERIL, NeuroCERIL is pre-programmed with a knowledge-base of cause-effect relations that describe the implementation of abstract intentions. These causal relations are used during learning to infer a demonstrator's intentions (goals, on left side of arrow) from the demonstrator's actions (right side of arrow). For example, the intention to `relocate` an object (`obj`) to a target location (`loc`) causes a sequence of concrete motor actions: `grasp` the object, `move` it to a target location, and `release` the grasp. This is encoded as a template or schema that can be matched to observed behavior:

```
relocate<obj,loc> →

    grasp<obj, gripper>,

    move<gripper, loc>,

    release<gripper>
```

Here, the right arrow represents causation, and indicates that the intention on the left side of the arrow can cause the ordered sequence of actions on the right side. It is important that parameter names (`obj`, `loc`, `gripper`) are repeated in this schema, because this indicates correspondences between the parameters of the intention and the actions that it causes (e.g., the same gripper is used for each action). NeuroCERIL verifies these correspondences when it infers casual intentions in a demonstration. In addition, each schema may include explicit logical predicates that must be satisfied for a cause-effect relation to be plausible. For example, the intention to open a drawer may cause a sequence of grasping, moving, and releasing, but the grasped object must be a drawer handle, and the drawer must be closed prior to opening. These constraints can be encoded as logical statements that NeuroCERIL must verify while inferring causal intentions.

The effects of a causal intention may include other abstract intentions, allowing causes to be

chained together to create hierarchies of cause-effect relations. For example, the intention to swap the location of two objects may be implemented as a sequence of `relocate` intentions:

```
swap<obj1, obj2> →

    relocate<obj1, temp>,

    relocate<obj2, loc1>,

    relocate<obj1, loc2>
```

A concrete demonstration of this swapping behavior would involve a sequence of `grasp`, `move`, and `release` actions that are caused by intermediate `relocate` intentions. Thus, inferring the causes of demonstrated actions requires a recursive inference process: when an intention is recognized as a plausible cause, it is treated as the effect of plausible higher-level causal intentions.

NeuroCERIL's causal knowledge-base may contain multiple schemas describing different implementations of the same intention. For example, the location of two objects may be swapped without placing one in an intermediate location, by instead keeping one object in hand while relocating the other:

```
swap<obj1, obj2> →

    grasp<obj1, gripper>,

    move<gripper, temp>,

    relocate<obj2, loc1>,

    move<gripper, loc2>,

    release<gripper>
```

Here, `temp` refers to a location in the air to which the demonstrator lifts `obj1` to, holding it there

while `obj2` is relocated to the original location of `obj1` (`loc1`). A key feature of NeuroCERIL's knowledge-base is that causal relations are agnostic to the implementation of their effects: a higher-level intention that is implemented using `swap` does not specify which implementation of `swap` to use. This flexibility affords generalization during imitation; a demonstration involving one implementation of `swap` can be imitated using the other implementation. Thus, causal inference allows the imitator to abstract away circumstantial details of the demonstration environment and adapt learned skills to novel circumstances. This may also be necessary if the embodiment of the imitator differs from that of the demonstrator (e.g., number of arms, dexterity, range of motion), requiring the imitator to implement learned skills in a different but equivalent way.

Finally, a sequence of demonstrated actions may have more than one plausible explanation. This may occur if two sequences of cause-effect relations share the same sequence of effects. For example, given the following three cause-effect relations (shown without parameters for simplicity):

`X` $\rightarrow$ `A, B`

`Y` $\rightarrow$ `C`

`Z` $\rightarrow$ `A, B, C`

a sequence of actions `(A, B, C)` may be caused by the sequence of intentions `(X, Y)`, or the single intention `Z`. In this case, the most parsimonious (i.e., simplest or shortest) explanation is usually preferred: `(A, B, C)` was caused by `Z`.

The next subsection describes the new hypothetico-deductive causal inference algorithm that NeuroCERIL uses to identify the most parsimonious explanation for a demonstration. NeuroCERIL is provided with a pre-programmed knowledge-base of cause-effect relations with optional

logical constraints, as described above. The initial state of the virtual environment is provided as a list of objects encoded as collections of named properties, which may change during the demonstration (e.g., location). The demonstration is encoded as a sequence of parameterized actions, each paired with a list of changes that occur to objects in the environment. The output of this algorithm is a sequence of top-level intentions identified as causes of the demonstrated actions, which serves as an explanation of the demonstration as well as an encoding of the demonstrated skill.

5.1.2 Hypothetico-Deductive Causal Inference

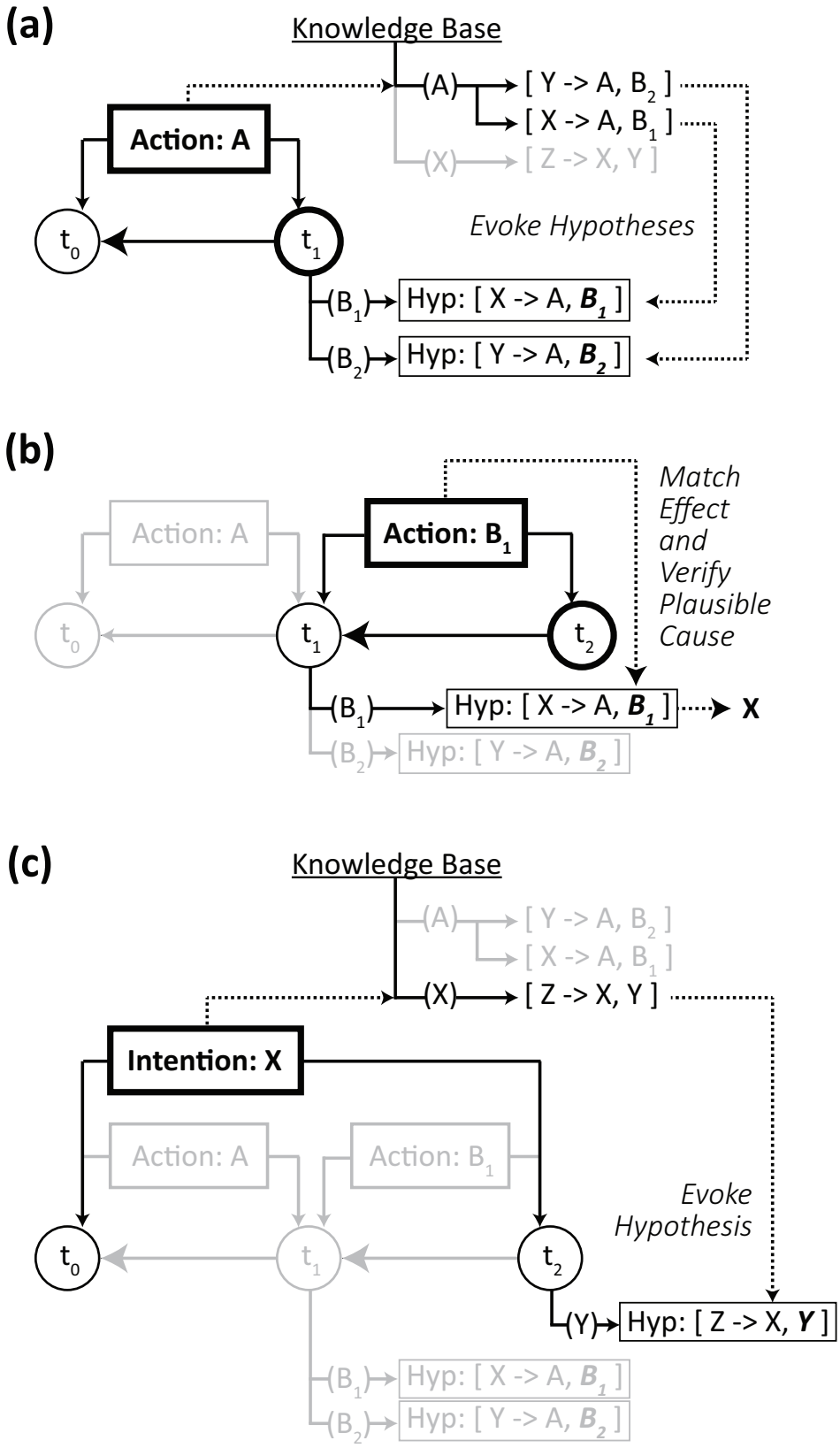
NeuroCERIL's approach to causal inference differs from CERIL's in a way that is more cognitively plausible and memory efficient. Whereas CERIL conducts an exhaustive bottom-up search that makes multiple passes through an entire demonstration, NeuroCERIL uses a more human-like hypothetico-deductive approach that involves a combination of bottom-up and top-down reasoning to iteratively construct a causal explanation for a demonstration as it occurs. When an action is observed, NeuroCERIL consults its cause-effect knowledge-base to generate explicit hypotheses about the demonstrator's causal intentions (bottom-up), and uses them to deduce testable predictions about subsequent actions (top-down). NeuroCERIL's cognitive processing is focused on evaluating these predictions to verify or falsify hypotheses. By organizing active hypotheses based on their predictions, NeuroCERIL can efficiently access those that are relevant to an observation, and avoid considering those that are not. When all of the predictions of a hypothesis are verified by observations, the hypothesized causal intention is treated as an observation and processed recursively to identify plausible higher-level intentions that may have caused it. In this way, NeuroCERIL constructs hierarchies of cause-effect relations that are sup-

ported by observations, and that represent plausible explanations for sequences of demonstrated behavior. As plausible intentions are identified, NeuroCERIL updates *parsimony pointers* that indicate the shortest sequence of intentions that covers the actions observed so far. At the end of the demonstration, these pointers are traced back to identify the most parsimonious explanation for the entire demonstration. This process is illustrated in Figure 5.1, outlined as pseudocode in Algorithm 1, and described in more detail below.

NeuroCERIL uses several different data structures to keep track of observed actions, their relative timing, and hypotheses about their causal explanations. These data structures are organized around a timeline, represented in memory as a chain of discrete time-points that delimit observed actions (circles connected by solid arrows in Figure 5.1). Each action contains pointers to the timepoints immediately before and after it (i.e., start and end points). For concrete primitive actions that are directly observed (e.g., grasping and releasing), these timepoints are adjacent in the timeline (**Action: A** in Figure 5.1a). However, inferred high-level causal intentions can be implemented with multiple lower-level actions, and can therefore span several timepoints (**Intention: X** in Figure 5.1c).

Hypotheses originate from a bottom-up abductive reasoning process referred to as *evocation*; when an action/intention is observed, NeuroCERIL consults its causal knowledge-base to identify relevant cause-effect schemas that might explain it (top right of Figures 5.1a and 5.1c, and first loop of PROCESS_ACTION procedure in Algorithm 1). These schemas are stored in an associative array that maps each action/intention type to a list of schemas that predict it as their first effect. For example, the knowledge-base may contain a schema describing a cause-effect relation between the `relocate` intention and a sequence of `grasp`, `move`, and `release` actions. This schema is stored in the `grasp` list of the knowledge-base, and can be retrieved to evoke a hypothesis

Figure 5.1: (next page) Hypothetico-deductive process for inferring hierarchical intentions during imitation learning. **(a)** An action of type A is observed at timepoint t_1 and added to a chain of timepoints in memory (bold box and circles, left). The knowledge base (top right) is consulted to evoke plausible hypotheses about the action's cause. These hypotheses are added to timepoint t_1 (bottom right) and stored according to their subsequent predictions (B_1 and B_2). **(b)** An action of type B_1 is observed at t_2 (center), and matched to the hypothesis that predicted it, generating a plausible causal intention of type X (bottom right). **(c)** This intention is processed recursively as an observation spanning t_0 to t_2 (left). A hypothesis is evoked that predicts an intention of type Y at t_2 (bottom right).


(a)
Knowledge Base
(A)
[Y -> A, B2]
[X -> A, B1]
(X)
[Z -> X, Y]
Action: A
Evoke Hypotheses
t0
t1
(B1)
Hyp: [X -> A, B1]
(B2)
Hyp: [Y -> A, B2]
(b)
Match Effect and Verify Plausible Cause
Action: A
Action: B1
t0
t1
t2
(B1)
Hyp: [X -> A, B1]
X
(B2)
Hyp: [Y -> A, B2]
(c)
Knowledge Base
(A)
[Y -> A, B2]
[X -> A, B1]
(X)
[Z -> X, Y]
Intention: X
Action: A
Action: B1
Evoke Hypothesis
t0
t1
t2
(Y)
Hyp: [Z -> X, Y]
(B1)
Hyp: [X -> A, B1]
(B2)
Hyp: [Y -> A, B2]

Algorithm 1: Pseudocode for hypothetico-deductive causal inference algorithm

---

procedure EXPLAIN($demo$, $init\_env$) ▷ infers causal explanation for $demo$nstration
  $curr\_env \leftarrow init\_env$ ▷ initial environment state
  $prev\_time \leftarrow$ create timepoint with $init\_env$ ▷ initial timepoint
  for each $action$ and record of environment $changes$ in $demo$ **do**
    $curr\_env \leftarrow$ new environment with $changes$ chained off prior $curr\_env$
    $curr\_time \leftarrow$ new timepoint with $curr\_env$ chained off $prev\_time$
    set $action$ start and end timepoints to $prev\_time$ and $curr\_time$
    PROCESS_ACTION($action$)
    $prev\_time \leftarrow curr\_time$
  end for
  return TRACE($curr\_time$) ▷ reconstruct top-level explanation from timeline
end procedure

procedure PROCESS_ACTION($action$) ▷ updates timeline and hypotheses
  if $action$ has shorter path to initial timepoint **then** ▷ compare with parsimony pointer
    update parsimony pointer for $action$'s end timepoint
  end if
  for each $schema$ predicting $action$ type as first effect **do** ▷ new hypotheses
    $hypothesis \leftarrow$ generate hypothesis from $schema$ ▷ abductive evocation
    VERIFY_HYPOTHESIS($hypothesis, action$)
  end for
  for each $hypothesis$ predicting $action$ type at $action$ end timepoint **do** ▷ old hypotheses
    VERIFY_HYPOTHESIS($hypothesis, action$)
  end for
end procedure

procedure VERIFY_HYPOTHESIS($hypothesis, action$) ▷ verifies a $hypothesis$ for cause of $action$
  if $action$ matches $hypothesis$ prediction **then** ▷ unification and constraint checking
    if $hypothesis$ is fully matched **then** ▷ all predictions verified
      $intent \leftarrow$ generate causal intention from $hypothesis$
      PROCESS_ACTION($intent$) ▷ process inferred intention as an observed action
    else
      update $hypothesis$ prediction
      add $hypothesis$ to $action$'s end timepoint ▷ advance hypothesis in timeline
    end if
  end if
end procedure

```
procedure TRACE(curr_time)          ▷ traces top-level explanation using parsimony pointers
    intent ← parsimony pointer of curr_time
    prev_time ← start timepoint of intent
    if prev_time is initial timepoint then
        return list containing intent                    ▷ first intention in top-level sequence
    else
        prior_intents ← TRACE(prev_time)                                          ▷ recursion
        append intents to prior_intents
        return prior_intents
    end if
end procedure
```

that an observed `grasp` action was caused by the intention to `relocate` the grasped object. This hypothesis must be evaluated to determine if the observed action satisfies the constraints of the schema, including correspondences between parameters with shared names as well as explicit logical predicates that must be true for the causal relation to be plausible (VERIFY_HYPOTHESIS procedure in Algorithm 1). Corresponding parameters are matched with a symbolic pattern matching procedure (unification) that was previously implemented using neural computations (Chapter 4). If these constraints are not satisfied, the hypothesis is immediately abandoned. Otherwise, it is added to the timeline and used to make predictions about subsequent actions, as described below. In Figure 5.1a, the knowledge-base contains two schemas indicating causal relations that might explain the observed action of type A (top right). The evoked hypotheses predict a subsequent action of type B_1 and B_2, respectively (bottom right).

Each timepoint contains a set of hypotheses that make predictions about actions or causal intentions that might occur immediately after it. Like cause-effect schemas in the knowledge-base, these hypotheses are stored in an associative array that maps the predicted action/intention type to the hypotheses that predict it at that timepoint. When an action/intention is observed, the hypothesis set for its starting timepoint is consulted to retrieve the hypotheses that predicted it

(second loop of PROCESS_ACTION procedure in Algorithm 1). For the `relocate` example above, the evoked hypothesis predicts that the demonstrator will `move` the grasping arm immediately after the `grasp` action occurred. When `move` is observed, this hypothesis is retrieved and evaluated to determine if its prediction was satisfied. This involves verifying the schema's logical constraints, as described above. If these constraints are satisfied and the hypothesis predicts further actions, the next prediction is retrieved, and the hypothesis is advanced to the next timepoint. When all of the predictions for a hypothesis are verified, it is used to generate a plausible causal intention that is added to the timeline. This intention is then processed recursively in order to generate and verify further hypotheses about its underlying cause (call to PROCESS_ACTION procedure within VERIFY_HYPOTHESIS procedure of Algorithm 1). In Figure 5.1b, the observed action of type B_1 matches the prediction of a hypothesis at t_1, and the corresponding causal intention of type X is generated (bottom right). This intention is then processed as an observation in Figure 5.1c, and a new hypothesis is evoked proposing that an intention of type Z is the underlying cause. This new hypothesis predicts an intention of type Y at time t_2, and is added to the timeline accordingly (bottom right).

Each timepoint also contains a representation of the state of the environment that is consulted during hypothesis verification (Figure 5.2). The environment contains several objects with named properties that can change over the course of a demonstration. At the beginning of the demonstration, NeuroCERIL is provided with a full specification of the initial state of the environment. When an action is observed, NeuroCERIL is also provided with a list of changes to object properties that were caused by the action. Rather than maintaining full copies of the environment state at each timepoint, which would require substantial memory, NeuroCERIL stores a record of these changes that can be consulted to determine the state of the environment at a given timepoint

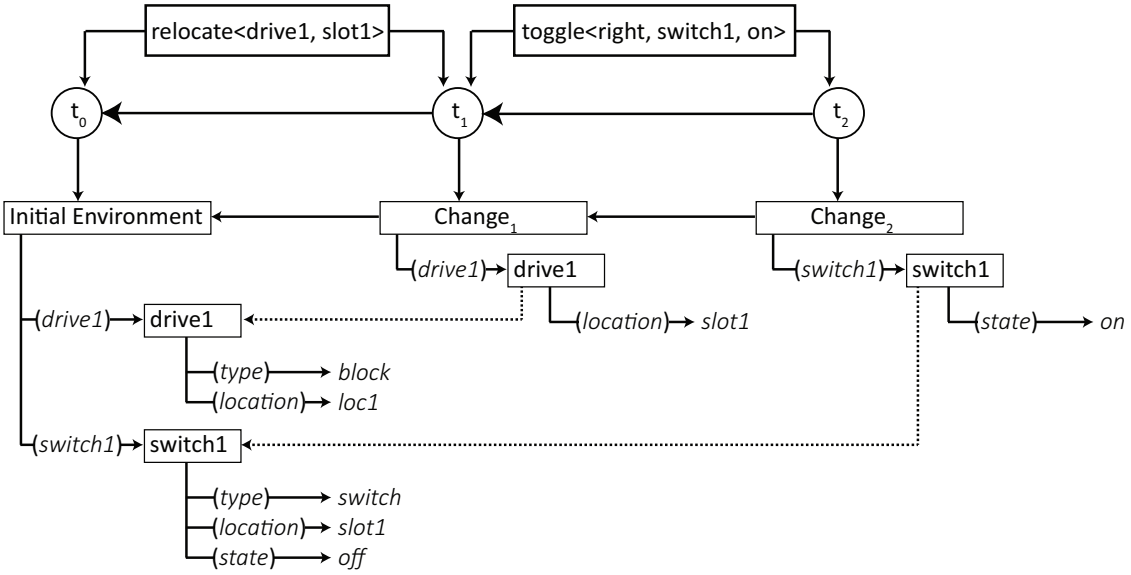


Figure 5.2: Representing a changing environment in memory during a simple demonstration, in which a hard drive (*drive1*) is moved from an initial location (*loc1*) to a slot (*slot1*), and an adjacent switch (*switch1*) is toggled *on* with the *right* hand. Each action is stored in a timeline of discrete timepoints (circles connected by solid arrows, top; see Figure 5.1). The initial timepoint (t_0, top left) stores a representation of the initial environment as a nested associative array ("Initial Environment", left). Each entry in the array maps a symbolic name to an object (*drive1* or *switch1*, bottom left). Objects are stored as inner associative arrays, which map symbolic names of properties to their corresponding values (e.g., the *location* of *drive1* is initially *loc1*). Subsequent timesteps store records of changes that occur in the environment ("Change$_1$" and "Change$_2$", center and right). Like timepoints, these records are chained together in reverse chronological order, and each record is stored like the initial environment as a nested associative array. The inner associative arrays of corresponding objects are also chained together (dotted lines). This compact representation uses minimal memory, but affords access to the state of the environment at each timepoint.

(or its initial value if it was not changed). To query the state of an object property at a given timepoint, NeuroCERIL retrieves the most recent change that occurred to that property prior to that timepoint. To support this operation, each timepoint stores a nested associative array, where the outer array stores entries for changed objects, and each inner array stores entries for a particular object's changed properties. Importantly, the inner arrays representing changes to the same object at different timepoints are chained together to allow an efficient search for the most recent change

to a specific property (dotted lines in Figure 5.2).

Finally, NeuroCERIL maintains pointers in memory that can be used to retrieve the most parsimonious explanation for the actions observed so far. The best explanation is the shortest sequence of intentions that covers all directly observed primitive actions without gaps or overlaps. This is represented by a chain of alternating timepoints and intentions that leads from the last timepoint to the first timepoint. Thus, each timepoint maintains a *parsimony pointer* to the intention that provides the shortest path back to the first timepoint in the demonstration. Whenever a plausible intention is identified, it is compared with the current best intention for the intention's end timepoint (beginning of PROCESS_ACTION procedure in Algorithm 1). NeuroCERIL performs this comparison by iterating through the paths simultaneously until the initial timepoint is reached. If the newly identified intention provides a shorter path to the initial timepoint, it is replaced as the current best intention for the end timepoint. When the demonstration is complete, the best explanation for the full sequence of observed actions can be reconstructed by following the chain of parsimony pointers from the final timepoint back to the initial timepoint (TRACE procedure in Algorithm 1).

5.1.3 Neural Implementation

NeuroCERIL's architecture (shown in Figure 5.3) is an extension of NeuroLISP, a programmable neural network that learns to store and evaluate programs written in a subset of the Common LISP programming language. Many of the details of NeuroCERIL's functionality are shared with NeuroLISP and can be found in Chapter 4. This section provides a brief overview and highlight the novel features of NeuroCERIL's architecture that extend its computational capabilities beyond

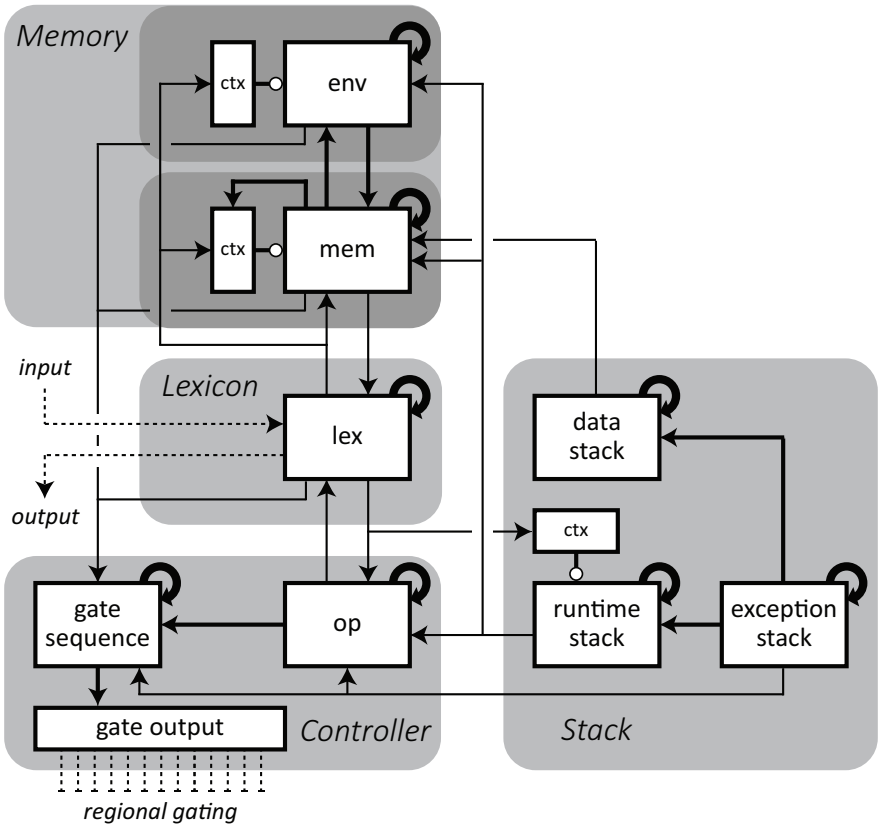


Figure 5.3: NeuroCERIL's neurocognitive architecture that learns to perform hypothetico-deductive causal inference. This architecture is an extension of NeuroLISP (Chapter 4), and is made up of several recurrent neural regions (boxes) with inter-regional connections (solid arrows) that are divided into sub-networks (grey background boxes). Like NeuroLISP, NeuroCERIL implements an interpreter for a LISP-like programming language that is used to implement high-level algorithms. Programs and other data are stored as systems of learned attractors in the `mem` region (center), and are evaluated via top-down control of connection gates (*regional gating*, bottom left). Inputs and outputs to the model are mediated by the `lex` region (center), which represents symbols as distributed patterns of activity that can be dynamically associated with activity patterns in adjacent regions (e.g., data structures in `mem`). NeuroCERIL implements a new class system using existing circuitry for variable bindings (connectivity between the `mem` and `env` regions, top), and also includes a new `exception stack` region (bottom right) that supports exception handling. These new features allow for efficient implementation of the causal inference algorithms described in this chapter (see text for details).

NeuroLISP.

Like NeuroLISP, NeuroCERIL represents programs and other symbolic data structures as

learned systems of dynamical attractor states and associative transitions between attractor states (attractor graphs; Chapter 3). Programs are evaluated via top-down control of gated connectivity between and within neural regions. This guides the flow of activity according to instructions retrieved from neural memory, much like a conventional computer architecture controls data flow according to instruction opcodes. Importantly, NeuroCERIL also controls its own learning in this way, allowing it to construct, access, and modify data structures stored in memory during program evaluation. Inputs and outputs are mediated by gated connectivity between the outer environment and a special region that represents discrete symbols as unique patterns of activity (`lex`, center of Figure 5.3). These connections allow NeuroCERIL to read symbolic inputs, including representations of programs, and output the results of program evaluation. During imitation learning, a demonstration recorded in SMILE is provided as a sequence of symbolic inputs, and NeuroCERIL outputs a sequence of symbolic outputs that encodes the inferred causal explanation.

NeuroCERIL is initialized with a learned program-independent virtual machine composed of procedures that implement the primitive operations of its programming language [97]. After initialization, NeuroCERIL is programmed with the causal inference algorithm described in Section 5.1.2, which is expressed in the language of NeuroCERIL's virtual machine. The details of initialization and program learning can be found in Chapter 4.

NeuroCERIL's virtual machine supports two major innovations that extend its computational capabilities beyond NeuroLISP and ease the implementation of its causal inference algorithm: a class system and an exception handling system. The class system allows specification of reusable programs (i.e., class methods) for initializing and modifying instances of complex data structures such as causal hypotheses, cause-effect knowledge, and observed actions. Instances of classes, called objects, are stored as collections of named pointers to other memories (i.e., class attributes).

The underlying implementation of objects makes use of the existing mechanisms for variable binding in NeuroCERIL's virtual machine; objects have corresponding lexical namespaces that store attributes as variable bindings (see Chapter 4 for details on variable binding in NeuroLISP).

Exceptions are errors that occur during program evaluation, and are triggered by events such as attempted access to undefined variables, attributes, or class methods. The exception handling system provides a mechanism for specifying dynamic responses to exceptions. This obviates the need for excessive program expressions that perform checks on data before access; a program instead can specify what should be done if retrieval fails. For example, when evoking hypotheses to explain an observed action, NeuroCERIL consults its causal knowledge-base to retrieve cause-effect recipes that are relevant to the observed action (see Section 5.1.2). If the knowledge-base does not contain any entry for the observed action type, retrieval will result in an exception that can be easily handled by skipping the evocation process.

Exception handling is supported by the `exception stack` region (bottom right of Figure 5.3), which maintains pointers to activity states in other regions that represent the state of the virtual machine. This region functions like the `runtime` and `data stack` regions (shared with NeuroLISP), which represent stack frames as distributed patterns of activity that have learned associations with activity patterns in other regions. Responses to exceptions are specified in programs with "try" expressions that include a primary sub-expression to evaluate, and an additional sub-expression representing the response. When a "try" expression is evaluated, the virtual machine first stashes its state on the exception stack, which involves learning associations in the pathways exiting the `exception stack` region. Then, the virtual machine attempts to evaluate the primary sub-expression. If an exception occurs, the virtual machine retrieves its prior state from the exception stack, and evaluates the response sub-expression. Upon completion, the top of

the exception state is popped, and evaluation of the program continues.

5.1.4 Experimental Evaluation

Empirical experiments were conducted to evaluate NeuroCERIL using a battery of test demonstrations that was used to test CERIL. These tests include procedural maintenance tasks involving replacing, swapping, and discarding mock hard drives in a docking assembly, as well as toy block stacking tasks (see [95] for details). NeuroCERIL's output was compared with CERIL's to confirm that it performs comparably, and carried out additional analysis on its memory usage and runtime to determine how well it scales with the length of demonstrations.

Runtime was measured as the number of timesteps in model simulations, and memory usage was evaluated by monitoring each simulation to count the number of associations that were learned during causal inference. Specifically, learning of attractor states and transitions was monitored in the underlying neural networks (stored in the recurrent connectivity of the `mem` region in Figure 5.3), as well as associations between namespaces and memory states that represent variable bindings for both local variables and object attributes (stored in the connection from `env` to `mem` in Figure 5.3). These associations represent the core data structures used during causal inference, such as observed actions, hypotheses, and inferred causes. The reported results include the associations formed specifically during the inference process, and exclude those that represent the causal inference programs and cause-effect knowledge that is shared across demonstrations.

NeuroCERIL's memory access patterns were further examined to gain a better understanding of its memory usage. This was guided by the hypothesis that the majority of memories constructed during inference would be highly transient memories that are only accessed across brief intervals

of time, such as abandoned causal hypotheses. This would indicate that NeuroCERIL might benefit from a functionally distinct short-term memory system in which memories rapidly fade if they are not refreshed by retrieval, much like human working memory. To test this hypothesis, instances of memory construction and access were recorded during inference, excluding memories such as program representations and cause-effect knowledge that are shared across demonstrations. For each recorded memory, its "lifespan" was determined as the interval between its initial learning and the final time it was retrieved during the simulation (i.e., a memory is "born" when it is first learned, and "dies" after its last retrieval during the simulation). This was then used to calculated the number of "living" memories over the course of the inference process and compared it to the total number of memories constructed. This provides a metric for the proportion of memories that are being actively utilized for causal inference.

5.2 Results

Table 5.1 shows the results for the same benchmark battery of procedural maintenance task demonstrations that were used to verify CERIL's functionality. For each task, the following quantities are reported: the number of actions recorded in the demonstration (**Act**), the number of top-level intentions in NeuroCERIL's causal interpretation (**Interp**), the number of timesteps of neural network simulation required for causal inference (**Timesteps**), and three measurements of learned associations that indicate model memory usage: the number of learned attractors (**Attr**) and attractor transitions (**Transit**) in the `mem` region, and the number of learned variable bindings (**Bindings**). NeuroCERIL produced causal interpretations (sequences of top-level intentions) equivalent to the minimum cardinality explanations identified by CERIL for each of the tests.

Figure 5.4 shows an example of the causal interpretation inferred for the *replace red with spare (1)* task.

Table 5.1: NeuroCERIL performance on battery of robotic imitation learning tasks

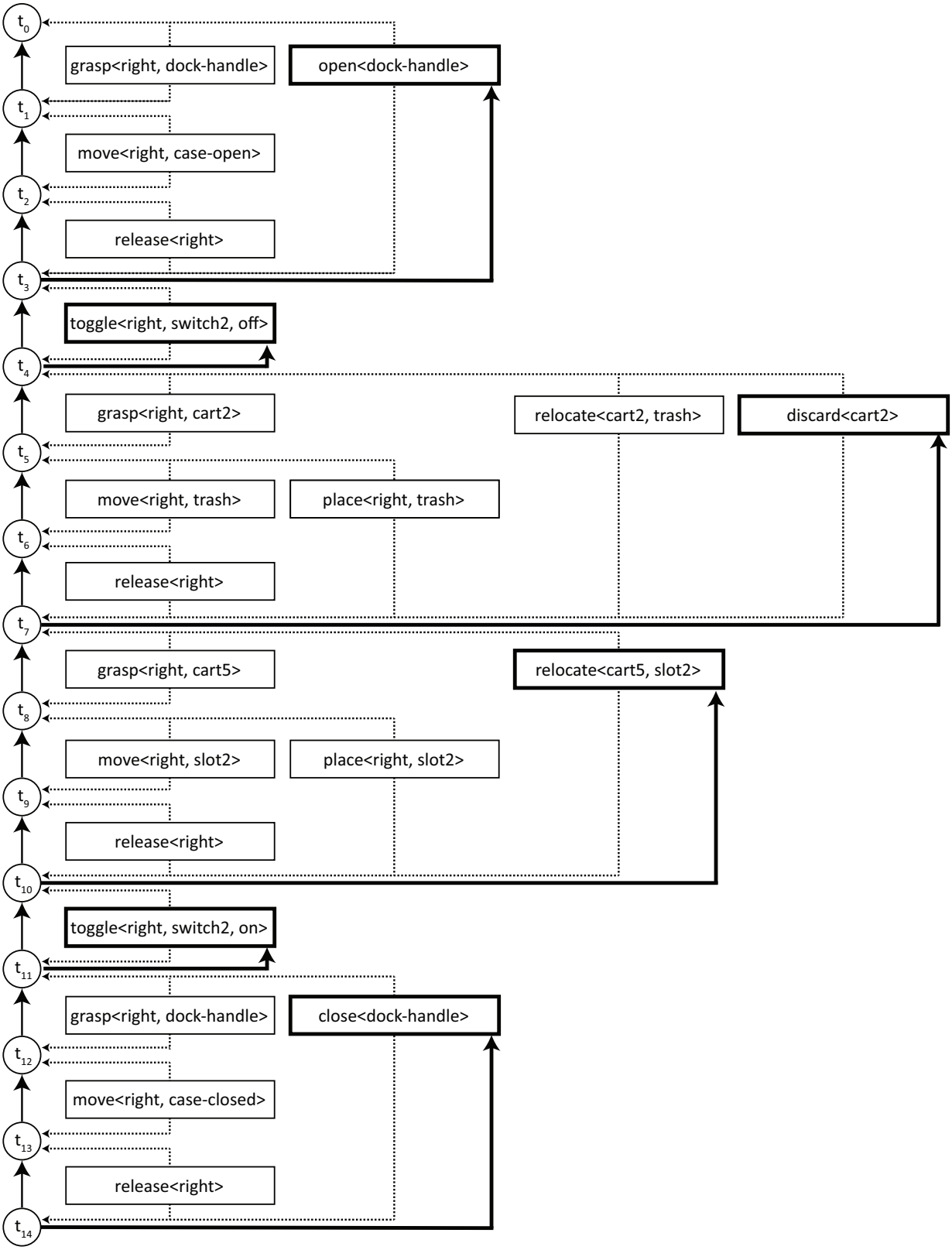
Demonstrated Task	Act	Interp	Timesteps	Attr	Transit	Bindings
Remove red drive (1)	7	3	353825	197	362	661
Remove red drive (2)	10	4	490085	250	468	917
Replace red with spare (1)	14	6	653758	341	642	1243
Replace red with spare (2)	14	6	653758	341	642	1243
Replace red with green (1)	15	7	668955	356	670	1276
Replace red with green (2)	15	7	668955	356	670	1276
Swap red with green (1)	16	8	668889	357	672	1278
Swap red with green (2)	16	8	668981	361	680	1285
Toy blocks (IL)	24	8	1224377	591	1150	2326
Toy blocks (AI)	30	10	1524253	735	1434	2905
Toy blocks (UM)	39	13	1975927	945	1848	3763

Runtime and memory usage results provide an empirical estimate of the complexity of NeuroCERIL's hypothetico-deductive causal inference algorithm. Figure 5.5 shows runtime and memory usage relative to the length of input demonstrations (**Act** in Table 5.1). Each datapoint corresponds to a row in Table 5.1, and the dashed lines show the results of linear regression computed for each metric. These results can be compared to Table 1 in [95][2], as well as the theoretical analysis of CERIL's complexity. Whereas CERIL exhibits a super-linear scaling of runtime and memory usage (indicated by the number of recognized top-level covers), Neuro-CERIL's runtime and memory usage scale linearly with the length of the demonstration. This is due to its online processing of demonstrations and incremental updating of data structures in memory that implicitly represent possible explanations.

Further analysis of memory usage focused on learned memory attractors, as they are a

[2]The experiments reported here used slightly more complex versions of the IL and AI block stacking tasks that include more blocks and actions than those reported in [95].

Figure 5.4: (next page) Causal interpretation produced by NeuroCERIL for the *replace red with spare (1)* task, which involves replacing a broken disk cartridge (*cart2*) in a mock disk drive drawer with a fresh cartridge (*cart5*). Actions and causal intentions are represented by rectangles that indicate the type of action/intention along with its parameters. Each action/intention points to its start and end timepoints, represented by circles (left side), which delineate concrete observed actions (leftmost column of boxes). The top-level explanation, composed mostly of abstract intentions, is indicated by bold boxes. NeuroCERIL reconstructs this explanation by following the shortest path from the final (t_{14}) to initial (t_0) timepoints using parsimony pointers (bold arrows, shown only for relevant timepoints; see Section 5.1.2).


t0
grasp<right, dock-handle>
open<dock-handle>
t1
move<right, case-open>
t2
release<right>
t3
toggle<right, switch2, off>
t4
grasp<right, cart2>
relocate<cart2, trash>
discard<cart2>
t5
move<right, trash>
place<right, trash>
t6
release<right>
t7
grasp<right, cart5>
relocate<cart5, slot2>
t8
move<right, slot2>
place<right, slot2>
t9
release<right>
t10
toggle<right, switch2, on>
t11
grasp<right, dock-handle>
close<dock-handle>
t12
move<right, case-closed>
t13
release<right>
t14

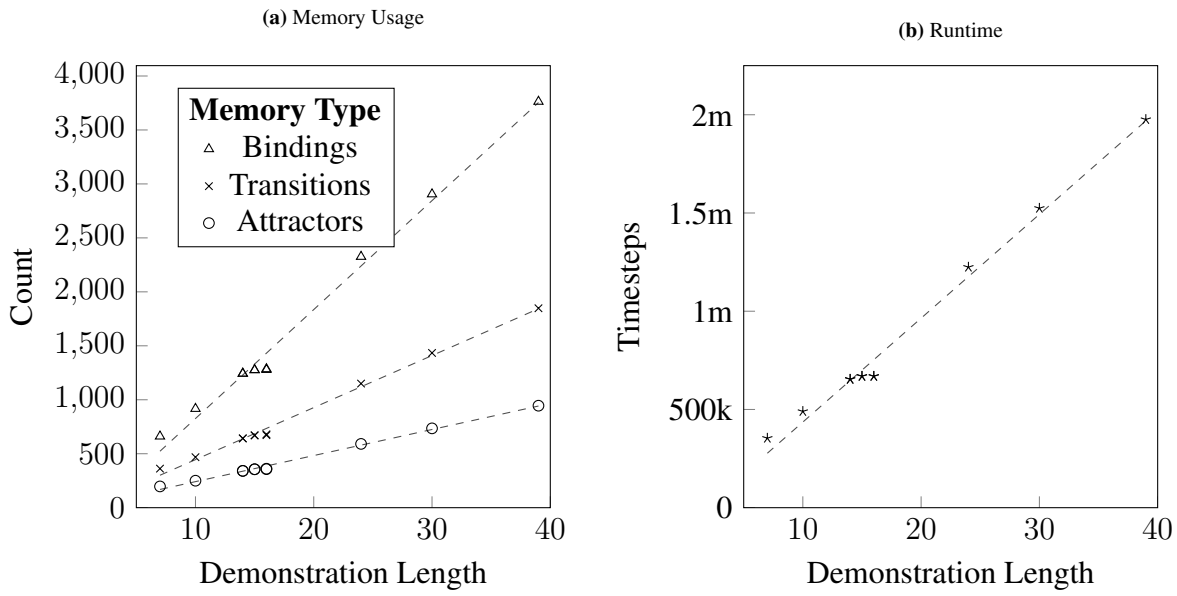


Figure 5.5: NeuroCERIL's memory usage and runtime during causal inference. Each data point corresponds to an individual imitation learning test task (rows in Table 5.1). Dashed lines are lines of best fit computed with linear regression, and show that memory usage and runtime scale linearly with the length of the demonstration (x-axis). **(a)** Memory usage is reported as the number of learned attractor states, attractor transitions, and variable bindings generated during causal inference (see text for details). **(b)** Runtime is reported as the number of timesteps of neural model simulation required for causal inference.

bottleneck for neural attractor memory (Chapter 3). The results for the *replace red with spare (1)* demonstration are reported here, but the results for other demonstrations are comparable. Figure 5.6a shows the "lifespans" of memory attractors constructed during causal inference for this demonstration, computed as the interval between initial learning and final retrieval. The x-axis indexes timesteps in which a memory attractor is constructed or retrieved, and each horizontal line indicates the lifespan of one memory attractor, indexed along the y-axis. Some memories remain alive through the majority of the inference process, such as representations of the environment and inferred causes that make up the final top-level cover. Others have relatively short lifespans, such as falsified causal hypotheses. The "living memories" at a given timestep refers to the set of mem-

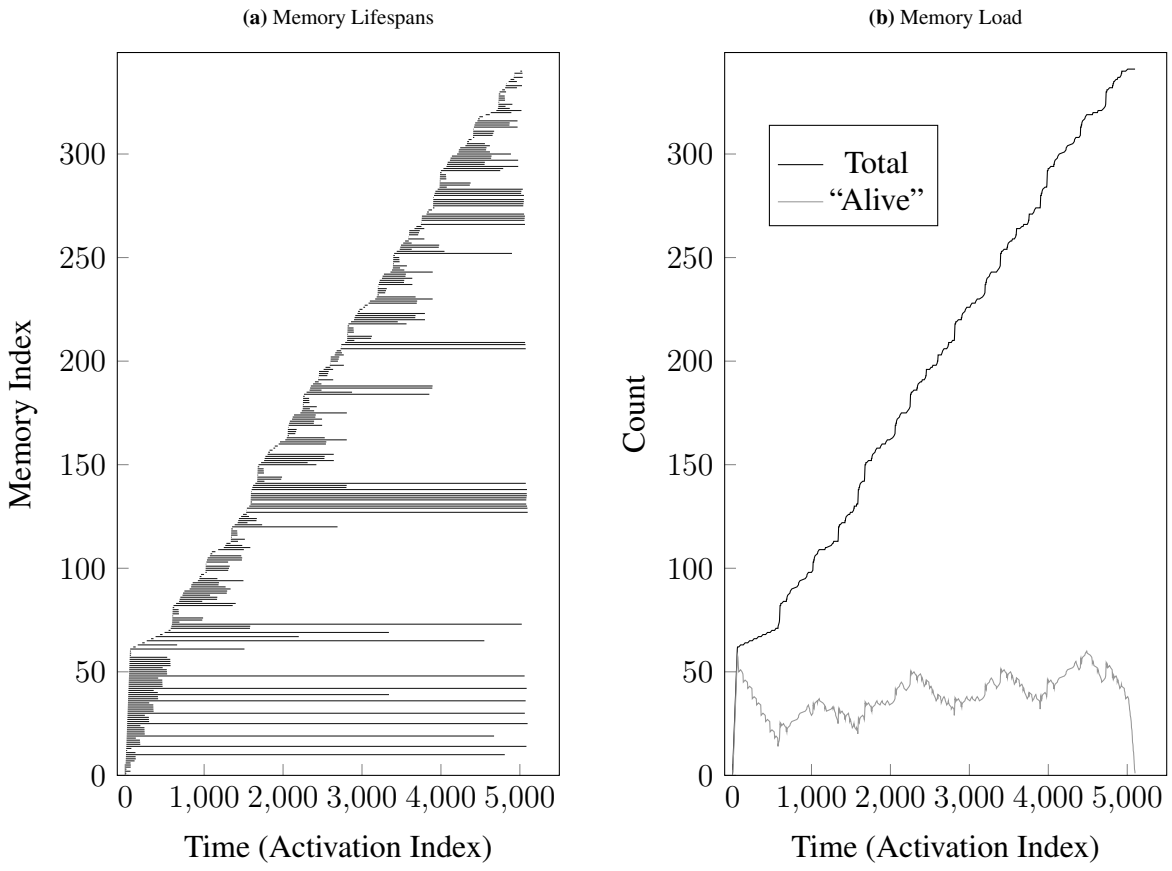


Figure 5.6: "Lifespans" of memory attractors constructed during causal inference on the *replace red with spare (1)* task, reported as the interval between initial learning and final retrieval. The x-axis indexes model simulation timesteps in which an attractor is learned or retrieved. **(a)** Each horizontal line represents the lifespan of one memory attractor, indexed along the y-axis. Shorter lines indicate that a memory attractor is only accessed over a brief interval, while longer lines indicate memories that are utilized over longer periods of time. **(b)** Memory load, reported as the total number of memories learned over time compared to the number of "living" memory attractors (i.e., attractors that have been learned at or before a given time and will be retrieved at a later time). While the total number of memories steadily increases over time, the majority of these memories are rapidly abandoned, and are only accessed over a brief period of time.

ories that have been learned prior to that timestep, and that will be accessed at a later timestep (i.e., a memory "dies" after the final timestep in which it is accessed). Figure 5.6b shows the total number of memory attractors learned over the course of the inference process, along with the number

of "living" memories at each timestep, which corresponds to the number of overlapping horizontal lines at each point along the x-axis in Figure 5.6a. Although the total number of learned memories increases steadily over time, the majority of these memories have relatively short lifespans. As a result, the number of "living" memories remains fairly stable over time, and never exceeds 20% of the total learned memories.

5.3 Discussion

This chapter presented NeuroCERIL, a brain-inspired neurocognitive controller for social robots that learn procedural tasks from human-provided demonstrations (i.e., robotic imitation learning). NeuroCERIL infers the intentions underlying demonstrated behavior using a novel causal inference algorithm based on human-like hypothetico-deductive reasoning, which combines bottom-up abductive inference with top-down predictive verification. This approach allows NeuroCERIL to iteratively construct plausible interpretations of demonstrated behavior as it is observed, make verifiable predictions about subsequent behavior, and generate compact explanations in terms of abstract intentions that can be generalized to novel environments. NeuroCERIL was evaluated on a benchmark battery of procedural maintenance and toy block-stacking tasks recorded in a virtual environment, demonstrating that it works effectively in robotic imitation learning domains. Empirical results also show that the model scales well with the length of demonstrated action sequences, and that the majority of its memory usage during causal inference is dedicated to transient short-term memories, much like human working memory.

NeuroCERIL is distinguished from prior approaches to robotic imitation learning by its use of neural computations to understand demonstrated behavior in terms of causal relations that

are directly related to high-level planning and cognitive-motor control. This not only affords generalization during imitation, but also facilitates an understanding of roles and perspectives that is critical to human-robot collaboration [197]. In addition, NeuroCERIL maintains a model of the external environment in memory and tracks changes that are induced by demonstrated motor activity. NeuroCERIL's understanding of demonstrations therefore provides an awareness of the physical consequences of behavior that is critical for safe and effective deployment of robots in sensitive environments.

Causal reasoning and compositionality are widely considered to be critical components of human cognition that are challenging for contemporary neural models to learn [82, 111, 113, 119]. NeuroCERIL performs causal reasoning with compositional models in working memory that represent the external environment and encode high-level behavioral plans, and is therefore a significant step toward developing neural networks with human-like reasoning capabilities. In addition, it has previously been proposed that neural models of working memory control, particularly in humanoid robots, provide a promising avenue to understanding conscious cognitive processing and its underlying basis in neural computations [160, 161]. NeuroCERIL is therefore also relevant to investigations of consciousness in machines and biological agents because it implements human-like cognitive algorithms in a brain-inspired neural architecture.

NeuroCERIL has several important limitations that suggest directions for future research. This chapter has focused on the causal inference component of imitation learning (left side of Figure 2.1), and did not address the generation of motor plans to implement learned skills during imitation. Prior work has shown that programmable neural networks can implement basic hierarchical planning (Chapter 3), and can perform adaptable motor control in simulated robots [99]. It is therefore feasible to integrate NeuroCERIL with low-level neural models of perception and

motor control to create a complete neurocognitive imitation learning system that performs both causal inference and plan generation.

The hypothetico-deductive causal reasoning algorithm proposed here relies on constraints in demonstrated behavior. In particular, implementations of abstract intentions must be performed in a fixed order as specified in the causal knowledge-base, and cannot be broken up by unrelated actions. In reality, procedural tasks might involve interleaved action sequences performed with both hands, and may include steps that can be performed in arbitrary arrangements. Thus, future work might involve modifying the causal inference algorithm to support these variations. This might also permit generalization to additional cognitive domains in which hypothetico-deductive reasoning is relevant, such as visual scene understanding and linguistic processing.

Finally, NeuroCERIL uses a unified memory system that does not include functionally distinct short-term and long-term memory. This means that long-term memories such as programs and causal knowledge may be gradually degraded as new short-term memories are constructed during program evaluation. Empirical results show that the majority of memories constructed during causal inference are only accessed during a narrow window of time, and are therefore highly transient short-term memories. This suggests that NeuroCERIL would benefit from a functional separation of short-term and long-term memory to protect the latter from interference.

6

Discussion

6.1 Summary

Despite recent progress, contemporary machine learning models struggle to capture key qualities of human cognition that are considered crucial for human-level machine intelligence. Many of these systems include deep neural networks, which are difficult to interpret and require data-intensive and computationally expensive training procedures. These issues are partially alleviated by the use of hybrid models that combine neural networks with traditional symbolic algorithms to leverage the unique advantages of both approaches. However, the cognitive capabilities of biological nervous systems indicate that human-level intelligence should be possible in artificial neural networks without the support of non-neural symbolic algorithms. Furthermore, the computational explanatory gap between cognitive and neural algorithms is a major obstacle to understanding the neural basis of cognition, an endeavor that is mutually beneficial to researchers in both AI and neuroscience.

Although prominent AI researchers disagree on specific strategies, they agree on several qualities of human cognition that are necessary for human-level AI, but exceed the capabilities of existing neurocomputational models. This dissertation was motivated by the hypothesis that these qualities can be captured in recurrent neural networks that represent symbolic information

as dynamical attractor states. This hypothesis is supported by the work presented in Chapters 3-5, which addresses the specific features outlined in Chapter 1:

- *Compositionality* is readily achieved in *attractor graph networks*, presented in Chapter 3. These models are capable of representing key data structures of symbolic programming, such as lists, trees, and associative arrays. Programmable neural networks can utilize attractor graphs to represent hierarchical plans (Chapter 3), programmatic and logical expressions (Chapter 4), and structured models of the environment and behavior of other agents (Chapter 5).

- *Causal reasoning* is supported by the high-level programmability of attractor neural networks. *NeuroCERIL*, presented in Chapter 5, implements a hypothetico-deductive causal reasoning algorithm that combines bottom-up abductive reasoning with top-down predictive verification. This procedure is effective for robotic imitation learning, but is more broadly relevant to problem-solving domains such as diagnostic and scientific reasoning.

- The behavior of programmable attractor neural networks corresponds to symbolic algorithms and data structures, and is therefore *interpretable*. NeuroCERIL illustrates how such networks can provide structured explanations to end-users, promoting transparency that eases the diagnosis of errors.

- NeuroCERIL learns the causal inference algorithm that it uses to learn from demonstrated behavior, and therefore exhibits a form of *meta-learning*. Notably, this skill is not built into NeuroCERIL's architecture; instead, it is represented in a shared program/data memory, and is therefore subject to improvement via learning without architectural changes. In addition,

NeuroCERIL recycles learned causal relations to construct explanations for various demonstrated skills, and is therefore capable of generalizing its knowledge to new situations.

- NeuroCERIL represents structured causal models of the environment and the intentions of human demonstrators, and therefore demonstrates *intuitive physics and psychology*. Although NeuroCERIL's grasp of physical and psychological dynamics is simple in comparison to humans, it illustrates how attractor neural networks provide a principled framework for embedding "start-up" software in neural models.

This dissertation therefore contributes to bridging the computational explanatory gap by presenting effective neural models of high-level cognitive abilities that are typically implemented with non-neural symbolic programs. This was accomplished by incorporating several important aspects of high-level symbolic programming into programmable neural networks, such as compositional data structures and scoped variable binding. Importantly, these models rely on biologically-plausible neurocomputational processes such as itinerant attractor dynamics, fast local learning, and multiplicative gating.

A significant benefit of programmable neural networks is that their behavior is supported by learned virtual machines that can be reconfigured without changes to their underlying architecture. This contrasts with most deep learning research, in which functionality is rigidly determined by structure. Instead, programmable neural networks are made up of general-purpose components that are flexibly combined via top-down control of neural gates, much like conventional computer architectures implement instruction opcodes by gating the flow of information through their underlying circuitry. This is in line with the suggestion that neural networks might benefit from integrating microprocessor-like operations [123, 125].

Unlike previous programmable neural networks, NeuroLISP and NeuroCERIL feature shared program/data memory, in which programs can be constructed and modified by other programs. These models are therefore capable of learning algorithms for *reasoning about reasoning* (i.e., *metacognition*). As discussed in Chapter 2, causal imitation learning resembles program synthesis in that it requires reasoning backwards from outputs to infer the algorithmic procedures that might have generated them. Thus, NeuroCERIL is a precursor to purely neural program synthesis, an important direction for future research.

6.2 Contributions

This dissertation makes the following specific contributions to the field.

- The first contribution is a method for representing compositional data structures as systems of dynamical attractors in recurrent neural networks. These systems, called *attractor graphs*, are capable of representing a diverse set of data structures based on labeled directed multi-graphs, including linked lists, trees, and associative arrays. Unlike many neural models of working memory, attractor graphs do not rely on persistent activity maintenance in slot-like neural populations. Instead, they are composed of learned associations that are established with fast local learning rules. This allows structured memories to be created, retrieved, and modified as needed in a shared neural population.

- The second contribution is a neural architecture called *NeuroLISP* that learns to store and evaluate programs written in a subset of Common LISP. NeuroLISP supports a variety of high-level programming constructs, including native support for compositional data structures, scoped variable binding, recursion, and the ability to manipulate programs as data.

Notably, this functionality is supported by a virtual machine that is learned with the same associative learning rules that are used to modify data in memory. NeuroLISP's implemented language can therefore be modified without changes to its underlying architecture, and potentially modified with experience.

- The third contribution is a hypothetico-deductive algorithm for compositional causal reasoning during imitation learning. This approach combines bottom-up abductive inference with top-down predictive verification in a way that resembles human problem-solving. By focusing cognitive processing on plausible explanations and their testable predictions, this algorithm avoids exhaustive enumeration of potential solutions and is computationally efficient.

- The fourth and final contribution is a neural architecture called *NeuroCERIL* that learns to perform intentional inference for imitation learning. This architecture extends NeuroLISP to support classes and exception handling, allowing it to learn the hypothetico-deductive reasoning algorithm described above. NeuroCERIL successfully learns a battery of procedural maintenance tasks that demonstrate its viability as a neural controller for cognitive robots that learn via imitation.

6.3 Limitations and Future Work

The models presented in this dissertation have a number of limitations that suggest avenues for future research.

A major bottleneck for attractor-based models is memory capacity, which scales linearly with the number of neurons in the network, but sub-linearly with respect to the number of weights

(Figure 3.6). This might be addressed in a number of ways, including the use of very sparse attractor states [141], alternate connectivity schemes (e.g., localized or patchy connectivity instead of full connectivity) [89], and allowing attractors to drift to reduce interference in memory [172].

A related issue is catastrophic forgetting, in which old memories are degraded as new memories are learned [143, 163]. For example, as NeuroLISP constructs new memories during evaluation, learned programs gradually deteriorate. While this is a desirable quality for working memory, it is a significant limitation for long-term memory, which has an extraordinary capacity in human beings. Future research should address methods for integrating short-term and long-term memory. One possibility is to include multiple weights per connection with varying learning rates [22, 65, 104, 110]. This might allow long-term memory to be updated gradually with repeated activation of representations in working memory, as in rehearsal-based methods [7].

NeuroLISP and NeuroCERIL currently require human-authored programs, and do not implement algorithms for discovering such programs on their own. However, because their shared program/data memory allows manipulation of programs as data, they are theoretically capable of implementing algorithms for program induction and synthesis, an important direction for future work. In addition, NeuroCERIL uses causal reasoning to infer behavioral plans from observations, which resembles program synthesis from program outputs. Thus, its hypothetico-deductive algorithm might be adapted for use with general programming languages such as Common LISP.

NeuroCERIL's causal reasoning might also be generalized for use in other cognitive domains beyond procedural learning, such as language processing and visual scene understanding, both of which require reasoning about the hidden structure underlying perceptual observations. Adapting to these domains requires loosening the constraints imposed by NeuroCERIL's inference algorithm, such as the requirement that effects are ordered and contiguous in time. In addition,

the predictive verification component of the hypothetico-deductive approach might be adapted to include explicit actions that gather new information (e.g., controlling eye movements based on predicted objects in the visual field). More ambitiously, this process might be adapted for use in general scientific experimentation [101, 102].

Finally, NeuroCERIL focuses on the causal inference component of imitation learning, and does not address plan generation. Although Chapter 3 presents a simple model of hierarchical planning, CERIL is capable of identifying objects in the environment, matching them to abstract intentions, and generating sophisticated motor sequences that are sensitive to obstacles in the environment. Accomplishing this in a neural model requires the integration of high-level cognitive processes with low-level perception and motor control. The latter are readily learned by conventional machine learning methods such as deep neural networks, which can be integrated with programmable attractor networks [46, 99, 191]. An integrated model that includes these components might benefit from learning methods beyond those discussed in this dissertation, such as reinforcement learning [98].

Appendix

A.1 Planning Task Domain

The hierarchical planning task described in Section 3.1.3.6 involves rule-based decomposition of sequential behaviors according to environmental conditions. This appendix section provides the decomposition rules, environmental bindings, and sequences of top-level actions used for the experiments in Section 3.2.6.

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). The following notation is used 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 $\star$ are only learned in

the "extended knowledge-base" condition in Section 3.2.6, and are not used during decomposition of the top-level sequences that were tested.

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

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

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

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

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

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

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

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 3.2.6, 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):

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

A.2 NeuroLISP Architecture Details

Table A.1 lists the neural regions in NeuroLISP and the region-specific gates utilized by the flashed interpreter firmware. The $g_r^{converge}$ gate (last column) is a special gate that simplifies attractor convergence in the *mem* region. When this gate is active, recurrent dynamics are run repeatedly using the auto-associative matrix until activity converges to a stable attractor, or until a pre-specified number of timesteps has elapsed (10 was used for testing). The meaning of each other gate can be found in Section 4.1.1 (see Equations 4.1 - 4.4).

Table A.2 lists the connections between and within regions in NeuroLISP and their functional purpose in model execution. Each connection links a source region to a target region, and

Table A.1: Neural regions in NeuroLISP (see Figure 4.2) and their implemented region-specific gates.

Region	g_r^{bias}	g_r^{noise}	g_r^{read}	g_r^{print}	$g_r^{saturate}$	g_r^{ϵ}	$g_r^{converge}$
data stack					✓		
runtime stack					✓		
op					✓	✓	
gate sequence	✓						
gate output							
lex		✓	✓	✓	✓	✓	
mem		✓			✓	✓	✓
mem-ctx		✓			✓	✓	
env		✓			✓	✓	
env-ctx		✓			✓	✓	

connections with shared source/target regions are distinguished by unique labels. Some connections (marked in the "Lrn" column) have learning gates that allow them to be updated during model execution ($g_{r,q[\ell]}^{learn}(t)$; see Equation 4.5). Connections from context regions (marked with a *) are unweighted one-to-one multiplicative connections (see Equation 4.2). All other connections are weighted all-to-all connections (see Equation 4.1).

A.3 Cons Cell Implementation

A cons cell is stored in neural memory as a trajectory from a unique memory state through the elements stored in the cons cell. Because the trajectory runs through the memory states stored in the cell, constructing a cons cell only requires the addition of a single attractor, and a data structure can be stored in more than one cons cell without being copied. For example, if a memory state is stored as the car element of two cons cells, it has two outgoing transitions linking it to the corresponding cdr elements (shown in Figure A.1). Each of these transitions is differentially accessible because it is contextualized by the unique state of the corresponding cons cell.

Table A.2: List of connections in NeuroLISP and their functions.

Target	Source	Label	Lrn	Function
data stack				
	data stack	fwd		pushing data stack
	data stack	bwd		popping data stack
runtime stack				
	runtime stack	fwd		pushing runtime stack
	runtime stack	bwd		popping runtime stack
op				
	lex			associating op sequences with symbolic labels
	op			advancing through operation sequences
	runtime stack		✓	returning from an operation sub-call
gate sequence				
	op			associating gate sequences with op sequence states
	gh			advancing through gate sequences
	lex		✓	comparisons on lexicon patterns
	mem		✓	comparisons on memory patterns
	env		✓	comparisons on namespace patterns
gate output				
	gate sequence			retrieving gate patterns from gate sequence states
lex				
	lex			checking if a symbol is built-in
	op			associating symbolic arguments with op sequence states
	mem		✓	retrieving the symbol stored in a memory state
mem				
	mem	auto	✓	memory state attractor convergence
	mem	hetero	✓	memory state hetero-associative transitioning
	*mem-ctx			multiplicative contextual gating of mem dynamics
	lex		✓	retrieving the dedicated memory state for a lexicon symbol
	env		✓	binding memory states to variables in namespaces
	runtime stack		✓	temporary storage during operation sequence execution
	data stack		✓	temporary storage during operation sequence execution
mem-ctx				
	mem		✓	cons cell and associative array key-value associations
	lex		✓	associative array map-key associations
env				
	env	auto	✓	binding-specific namespace attractor convergence
	env	hetero	✓	namespace nesting
	*env-ctx			multiplicative contextual gating of env dynamics
	mem		✓	closure binding
	runtime stack		✓	temporary storage during operation sequence execution
env-ctx				
	lex		✓	variable binding associations

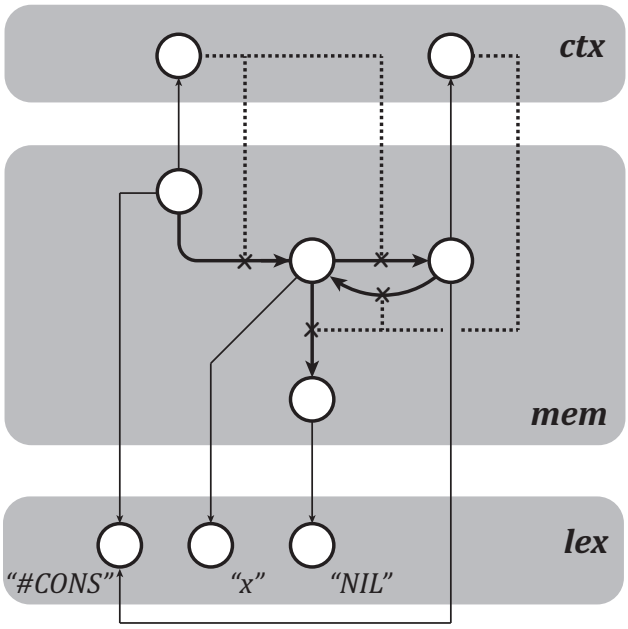


Figure A.1: Graphical depiction of nested cons cells stored in neural memory, illustrating state recycling for lists with repeat elements. Each circle represents a distributed activity pattern, and solid arrows represent learned associations/transitions between these patterns. The represented list contains two copies of the symbol "x", and can be constructed by the expression `(cons 'x (cons 'x NIL))`. Two *mem* states representing cons cells can be identified by their associations with the reserved "#CONS" symbol, represented by a unique activity state in *lex* (bottom left). In addition, each cons cell memory state has an associated context state (top) that contextualizes the transitions linking it with its corresponding car and cdr elements (see Figure 4.3b). Because both cons cells contain the same car element, the *mem* state associated with the "x" symbol contains two outgoing transitions with unique contexts (center circle in *mem* rectangle). Thus, in the context of the outer cons cell, this state transitions to the inner cons cell, and in the context of the inner cons cell, it transitions to the *mem* state for the null symbol (bottom of *mem* rectangle, associated with "nil" pattern in *lex*).

The car and cdr operations that can be performed on cons cells are implemented as follows. First, the cons cell memory state is activated, and the corresponding context state is retrieved via the pathway from *mem* to *ctx*. This state is then used to contextualize a transition from the cons cell to the car (first) element, which completes the car operation. To complete a cdr operation, an additional transition is then executed from the car element to the cdr (second) element. These

operations can be nested: if the car or cdr element is also a cons cell, its context state can be retrieved in order to access one of its elements.

Construction of a cons cell is carried out using several gates, including the noise (Equation 4.1), eligibility trace (Equation 4.4), and plasticity gates (Equation 4.6). The memory states to be linked as car and cdr elements are computed by sub-expressions during program evaluation, and pushed onto the data stack for retrieval during cons cell construction. Once these states are available, the cons cell context state is generated in *ctx*, and the transitions are constructed in reverse order. First, the cdr element is retrieved, masked by the context state (via multiplicative gating; see Equation 4.2), and stashed in the *mem* eligibility trace ($\epsilon_{mem}(t)$). Then, the car element is retrieved and masked, and the transition from car to cdr element is learned. Next, a new *mem* state is generated to represent the cons cell, and the process is repeated to link the cons cell state to the car state. Finally, the cons cell *mem* state is linked to the generated *ctx* state, making it accessible for subsequent car and cdr operations. A similar process is used for the "list" operation, which is equivalent to nested cons operations (i.e., `(list a b c)` == `(cons a (cons b (cons c NIL)))`, where `NIL` is a reserved symbol for the null state that serves as a list terminator).

A.4 Associative Array Implementation

The organization of associative arrays in memory makes it possible to check whether a key is contained in a map: if the transition from the map yields the key memory state, then the key is contained in the map (Figure A.2a), and the corresponding value can be retrieved via a transition from the key state (Figure A.2b). This can be confirmed with a comparison operation (Section 4.1.3.2). Adding or updating a key/value pair involves learning the transitions described above

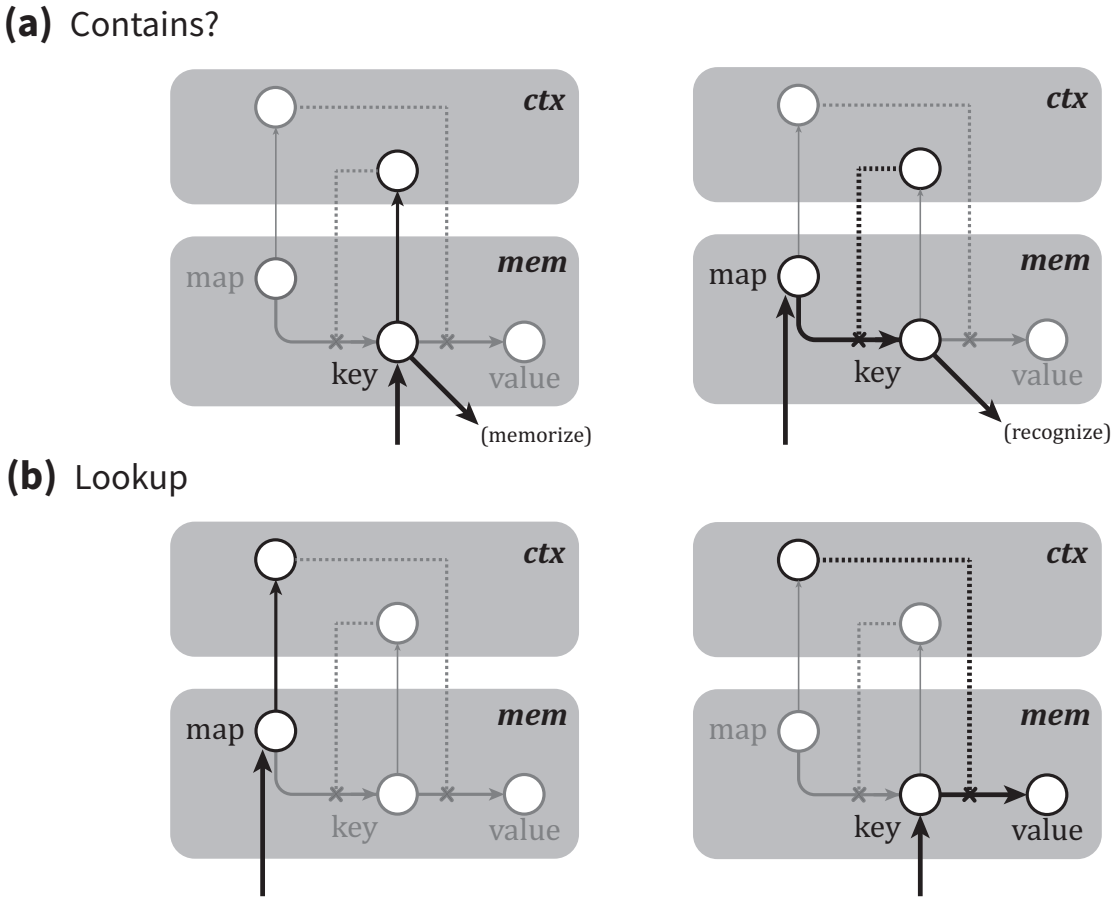


Figure A.2: Graphical depiction of the neural operations for checking for a key and retrieving its value from an associative array / map (see Figure 4.3c). Circles represent distributed activity patterns. Arrows entering *mem* states from below represent inputs from other regions that are not shown in the image (i.e., *lex* or *data stack*). The depicted operations are agnostic to the source of these inputs. (a) To check if a key is contained in a map, the key memory state (labeled "key") is retrieved and memorized for subsequent comparison (Section 4.1.3.2), and the corresponding context state in *ctx* is retrieved (bottom right circle in *ctx* rectangle). Next, the map state (labeled "map") is retrieved, and the key context state is used to execute a transition. If the key is contained in the map, this transition will yield the key memory state, which can then be recognized via comparison. Otherwise, a random state will be retrieved, producing a false comparison (i.e., failed recognition). Memorization and recognition occur in the pathway from *mem* to *gate sequence*. (b) Assuming the key is contained in the map, its value can be retrieved as follows. First the map state is retrieved, along with its corresponding context state (top left circle in *ctx* rectangle). Then, the key state is retrieved, and the context state is used to execute a transition to the value state (labeled "value", bottom right of *mem* rectangle).

(from map to key and key to value), and deleting a key/value pair involves changing the transition from the map state such that it targets a state other than the key state (e.g., the `NIL` state), thereby causing a mismatch that can be detected via comparison. Note that the above operations can be performed with constant-time complexity, as they do not require iteration through key/value pairs in memory. This is possible because of the underlying implementation of maps as attractor graphs with context-dependent transitions, rather than unbranched attractor sequences.

A.5 Organization of Interpreter Memory

The relationship between LISP programs and interpreter operation sequences can be seen in Figure A.3, where a LISP expression is represented by a cons cell in the *mem* region (top), and the low-level assembly op-sequences are represented as sequences of activity in the *op* region (bottom right). Each pattern of activity in *op* represents an assembly instruction that is associated with an opcode and an optional operand via the pathways from *op* to *gate sequence* and *lex*, respectively. The opcode corresponds to a sequence of states in the *gate sequence* region, each of which is associated with a pattern of activity in the *gate output* region that specifies which model components are active or inactive in a given timestep (bottom left of Figure A.3). The operand corresponds to a pattern of activity in the *lex* region that represents a discrete symbol (center right). These symbols serve various functions in the model; for example, they can be printed to the environment as output, used to contextualize environment lookups (Section 4.1.3.4), or used to retrieve a new *op* sequence during recursive evaluation (i.e., an op-sequence call). Retrieval and usage of the optional operand is directed by the gating sequence specified by the opcode. For example, an instruction that prints a symbol involves opening the pathway from *op* to *lex* to retrieve the instruc-

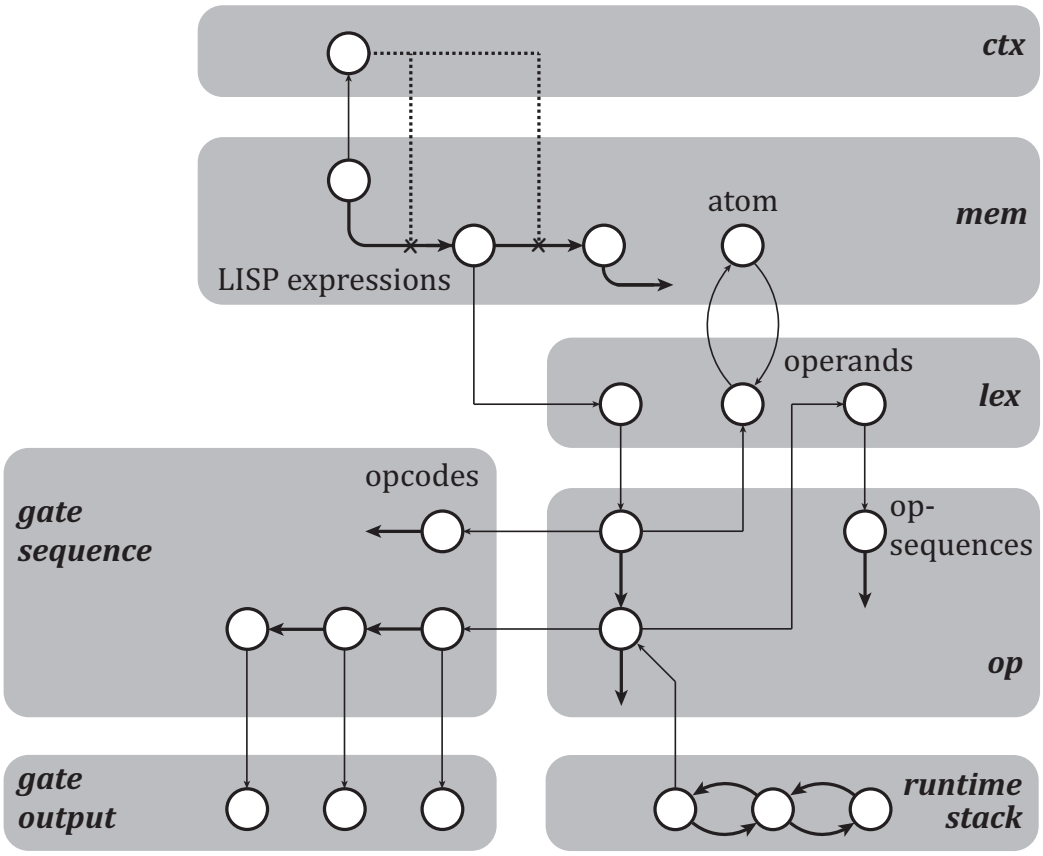


Figure A.3: Relations between states in various regions of the NeuroLISP architecture that implement interpreter functions. Circles represent distributed activity patterns, and arrows represent learned transitions between activity patterns within or between regions. A LISP expression is represented as a cons cell in the *mem* region (see Figure 4.3b), and its first element represents a LISP operator. This operator can be used to retrieve the corresponding sequence in the *op* region that implements the operation (bottom right *op* rectangle). Each *op* state has an opcode that corresponds to a unique sequence of activity in the *gate sequence* region, which specifies a temporal sequence of gating values via associations with *gate output* states (bottom left). These sequences control the behavior of the relevant model components, including the recurrent dynamics of the *gate sequence* and *op* regions themselves. Some *op* states are associated with an optional operand via the pathway from *op* to *lex* (center right). The corresponding opcode sequence determines what is to be done with this operand; for example, it may be used to retrieve an atomic memory state (top center state labeled "atom" in the *mem* rectangle), or to retrieve a new *op* sequence during recursive evaluation (right side arrow from *lex* state to *op* state). During recursive op-sequence evaluation, states in the *runtime stack* region are temporarily associated with the calling *op* state, which allows the model to return to the calling op-sequence instruction upon completion. For clarity, some associations are omitted from the diagram, including associations between *runtime stack* and *mem* states, as well as associations between some *op*, *gate sequence*, and *gate output* states.

tion operand, and then opening a special *lex* output gate that signals to the environment that an output is ready to be printed (see Section 4.1.3.3 for further details on input/output operations).

Stack regions are initialized with a bi-directional chain of associated states, each of which serves as a pointer that can be dynamically bound to activity states in other regions (*mem*, *env*, or *op*). Pushing onto a stack involves advancing its activity to the next pointer in the chain, and learning an association between the stack pointer state and an activity state in a target region. The associated activity state can be retrieved as needed during program execution by opening the activity gate from the stack region to the target region, and popped off the stack by advancing the stack region to the previous pointer state in the chain.

During recursive program evaluation, the Controller stores the current memory state and *op* state on the runtime stack, advances to the sub-expression memory state, and jumps to the beginning of the *eval* op-sequence. Upon completion of sub-expression evaluation, the Controller retrieves the *op* state from the stack, returning to the operation that initiated recursive evaluation. At this point, the current memory state represents the return value from evaluating the sub-expression, and the parent expression's memory state is available for retrieval from the runtime stack. If the parent expression contains multiple sub-expressions, the calling op-sequence can then stash the return value on the data stack, retrieve the parent expression, and advance to the next sub-expression for another round of recursive evaluation. Once all of the return values of sub-expressions are stored on the data stack, the parent expression's operation can then be performed. For example, the cons operation described in Section 4.1.2.1 will retrieve the memory states representing elements of the new cons cell, link them together in memory, and return the generated cons cell state upon completion. The returned cons cell is then passed up for use in the calling expression (e.g., another cons operation may use it as an element in another cons cell).

To evaluate an expression with a built-in operator, the *lex* pattern associated with the car element of the expression is retrieved, the head of the cons cell in *mem* is recovered, and the *lex* pattern is used to retrieve the operator's *op* sequence. Retrieval of the cons cell memory state makes the remainder of the expression available for access during execution of the operation.

A.6 Comparison Operations

Formally, comparisons are accomplished in the following manner. Unlike with other weight matrices, learning that occurs on the comparison pathways completely overrides the previous weight matrix, leaving only the most recently learned association for recognizing the memorized pattern. The eligibility trace of the *gate sequence* region is initialized to an activity state $\mathbf{v}_{gs}[true]$, which is the start of the jumping gate sequence that occurs when a comparison is successful. Thus, memorizing an activity pattern in a source region (*src*) at time t for later comparison can be expressed as:

$$\Delta W_{gs,src}(t+1) = \underbrace{\frac{1}{||\mathbf{v}_{src}(t)||}}_{\text{norm}} \underbrace{\mathbf{v}_{gs}[true]}_{\text{jump state}} \underbrace{\mathbf{v}_{src}(t)^{\top}}_{\text{source}} - W_{gs,src}(t) \tag{A.1}$$

where $W_{gs,src}(t)$ is the weight matrix for the pathway from a source region (*src*) to the gate sequence region (*gs*) at time t, $\mathbf{v}_{src}(t)$ is the *src* activity pattern to memorize, and $\mathbf{v}_{gs}[true]$ is the gate sequence activity state for op-sequence jumping. The updated weight matrix can then be used to determine if another activity pattern in *src* is similar enough to the memorized pattern to yield a successful comparison. This is done by presenting a new input pattern to *gate sequence* while attempting to drive *gate sequence* activity away from the "true" state toward an alternative "false"

state that corresponds to a false comparison:

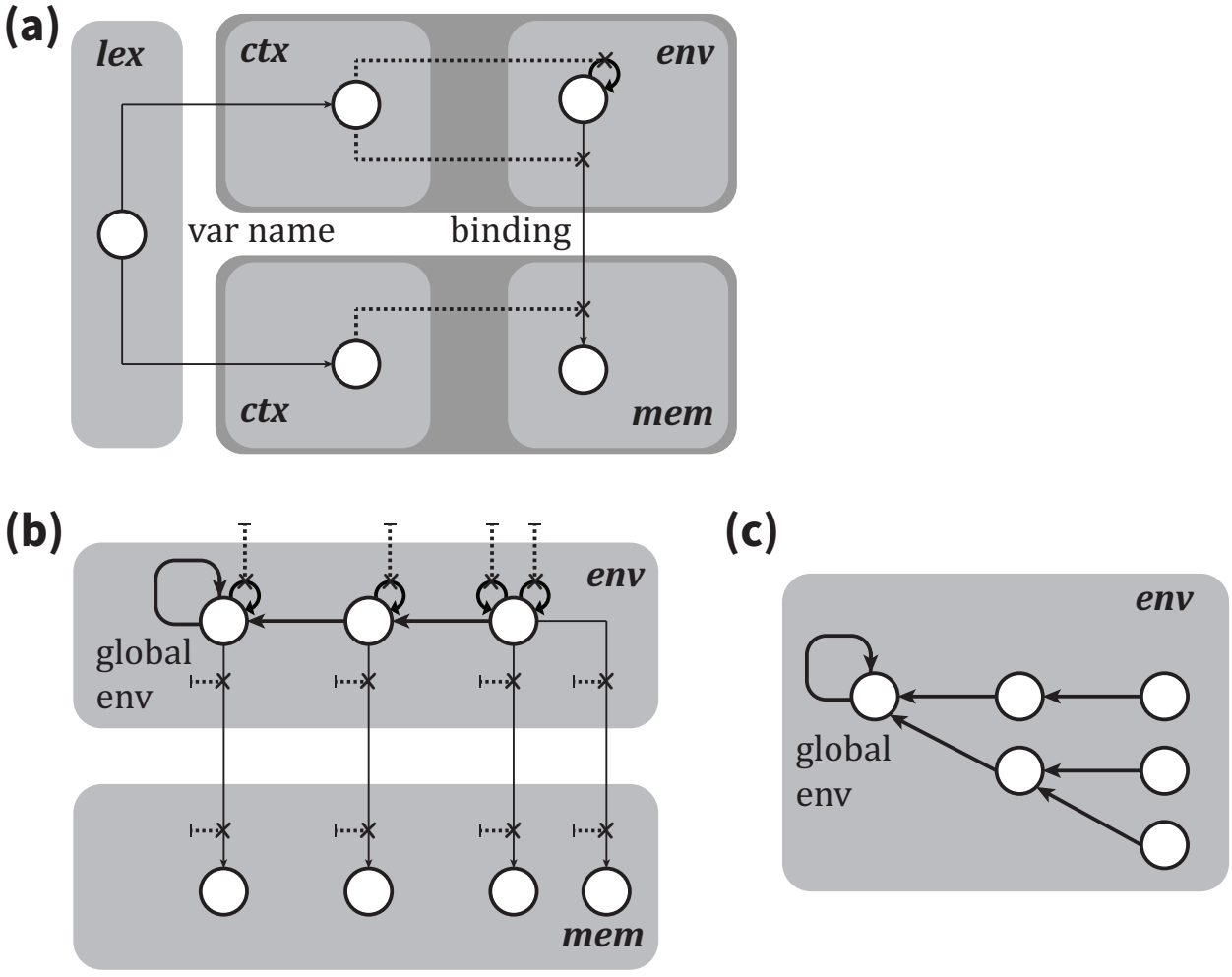
$$\mathbf{v}_{gs}(T+1) = \sigma_{gs}\Big(\underbrace{W_{gs,src}(T)\ \mathbf{v}_{src}(T)}_{\text{recognition}} - \underbrace{\theta\ \mathbf{v}_{gs}[false]}_{\text{bias}}\Big) \tag{A.2}$$

where $\mathbf{v}_{gs}(T+1)$ is the post-comparison activity state in the gate sequence region (*gs*) at time $T+1$, σ_{gs} is the activation function in *gs*, $W_{gs,src}(T)$ is the weight matrix described above, $\mathbf{v}_{src}(T)$ is the activity pattern in *src* that is compared to the memorized pattern, θ is the similarity threshold, and $\mathbf{v}_{gs}[false]$ is the gate sequence activity state for non-jumping operation advancement that occurs when a comparison fails. Equation A.2 is a special form of Equations 4.1 - 4.3, where the bias term $\mathbf{b}_{gs} = \theta\ \mathbf{v}_{gs}[false]$. The comparison threshold θ determines how similar the two *src* activation states must be to yield a successful comparison (typically $\theta = 0.95$). Note that when a non-threshold activation function such as the hyperbolic tangent is used, the activity pattern resulting from Equation A.2 will require saturation to match the magnitude of $\mathbf{v}_{gs}[true]$ or $\mathbf{v}_{gs}[false]$.

A.7 Lexical Environments

The organization of environments in neural memory is graphically depicted in Figure A.4. Environment namespaces are represented as activity states in the *env* region of the model (top of Figure 4.2). Each variable binding is maintained as a context-dependent association between a namespace and an activity state in the *mem* region that represents a variable's value (arrow labeled "binding" in Figure A.4a). This association is contextualized by activity states in the *ctx* regions (one for *mem* and one for *env*) that are derived from a *lex* pattern that represents the variable name ("var name" on left side of Figure A.4a). Thus, to retrieve a variable, an inter-regional transition is performed from *env* to *mem* in the context of the variable name. As with associative arrays / maps (Section 4.1.2.2), it is necessary to validate that a namespace contains a binding for a particular

Figure A.4: Graphical depiction of various components of environment management in NeuroLISP. Circles represent distributed activity patterns, and arrows represent learned associations/transitions between activity patterns. (a) Variable bindings are stored as contextualized associations between activity states in the *env* and *mem* regions (arrow labeled "binding", center). The *env* state represents an environmental namespace that may store bindings for several variables. A *lex* activity pattern representing the variable name ("var name", left) is associated with unique patterns in the context regions for *env* and *mem* that are each used to contextualize the corresponding region during variable retrieval (dashed lines connected to "binding" line). In addition, the context state for the *env* region also contextualizes an auto-association of the *env* state with itself (looped arrow in *env* circle, top). This makes it possible to determine if there is a binding for a particular variable in a particular namespace: if the namespace's *env* state is stable under contextualized auto-associative dynamics, a binding exists. This can be determined via a comparison operation (Section 4.1.3.2). (b) For dynamically scoped variable binding, environment namespaces are chained together into a sequence that terminates with the global environment ("global env", right side). When looking up a variable, the namespaces are inspected in order until a binding is found, or until a transition is executed from the global environment back to itself (i.e., no environments contain a binding for the variable). (c) For lexically scoped variable binding, namespaces are organized into an inverted tree, where different branches are created and maintained for function closures. When a function is defined, a namespace is created and bound to the memory state representing its closure (Section 4.1.3.5, Figure A.5). When the function is called, this namespace is retrieved and a new namespace is created to store argument bindings (Section 4.1.3.5).


(a)
lex
ctx
env
var name
binding
ctx
mem
(b)
env
global
env
mem
(c)
env
global
env

variable name before retrieving it. This is done using context-dependent auto-associative learning: namespaces that contain a binding for a variable are stable attractors under recurrent dynamics when the variable name's context pattern is present. Formally, this is expressed as:

$$\mathbf{v}_{env}[binding] = \mathbf{v}_{ctx-env}[var] \odot \mathbf{v}_{env}[namespace]$$
$$\mathbf{v}_{mem}[binding] = \mathbf{v}_{ctx-mem}[var] \odot \mathbf{v}_{mem}[value]$$

$$\sigma_{env}\Big(\mathbf{v}_{ctx-env}[var] \odot \big(W_{env,env[auto]}\ \mathbf{v}_{env}[binding]\big)\Big) = \mathbf{v}_{env}[binding] \tag{A.3}$$

$$\sigma_{mem}\Big(\mathbf{v}_{ctx-mem}[var] \odot (W_{mem,env}\ \mathbf{v}_{env}[binding])\Big) = \mathbf{v}_{mem}[binding] \tag{A.4}$$

where:

- $\mathbf{v}_{env}[namespace]$ is the activity state in the *env* region representing the namespace.
- $\mathbf{v}_{ctx-env}[var]$ and $\mathbf{v}_{ctx-mem}[var]$ are the variable-specific activity patterns in the *ctx* regions for *env* and *mem*, respectively.
- $\mathbf{v}_{mem}[value]$ is the activity state in the *mem* region representing the value of the binding.
- $\mathbf{v}_{env}[binding]$ and $\mathbf{v}_{mem}[binding]$ are the contextually-masked activity patterns in the *env* and *mem* regions. The binding is stored as an association between these states (Equation A.4).
- σ_{env} and σ_{mem} are the activation functions for neurons in the *env* and *mem* regions, respectively.
- $W_{env,env[auto]}$ is an auto-associative recurrent weight matrix in the *env* region that is updated when new bindings are created.
- $W_{mem,env}$ is an inter-regional weight matrix from the *env* region to the *mem* region that is updated when new bindings are created.

The relation expressed in Equation A.3 makes it possible to determine whether a binding exists for a given variable in a given namespace: the activity states before and after auto-associative dynamics can be compared (Section 4.1.3.2). If they match, the binding exists, and the relation expressed in Equation A.4 can be used to retrieve the corresponding value in *mem*. Because bindings are stored as context-dependent associations, a namespace may contain bindings for several different variables, much like an associative array can contain multiple key/value pairs.

Namespaces are organized in a nested fashion: when a variable is looked up, and no binding exists for the innermost namespace, the lookup proceeds to the encapsulating namespace. Once the outermost namespace is reached (referred to as the "global environment") and no binding is found, the lookup fails and the program returns an error. For dynamic scoping, namespaces can be maintained in a sequential chain (Figure A.4b), similar to those of the stack regions. Lexical scoping requires a branched organization (Figure A.4c) because function closures maintain the bindings that existed during function definition. Thus, lexical scoping allows access to bindings that would otherwise fall out of a dynamic scope and become inaccessible.

Operations that create new variable bindings evaluate sub-expressions and store the resulting values (*mem* states) on the data stack. Once all the values are computed, a new environment namespace (*env* state) is created to store the new bindings, and is associated with the prior namespace (arrows between *env* states in Figures A.4b and A.4c). Each binding is created by retrieving the variable/argument name (*lex* state), retrieving the corresponding value from the data stack, and learning the associations outlined above to create the binding (Equations A.3 and A.4). These bindings are then available during subsequent evaluation (i.e., the sub-expression of a `let` operation, or the body of a function). Upon completion, the newly created namespace is abandoned by advancing *env* to the prior namespace. However, if a function was defined before completion (i.e.,

a `defun` within a `let` expression), the abandoned namespace may be retrievable via a closure, as described in the following section.

A.8 Function Closures

Figure A.5 shows the associations making up a closure stored in memory, as well as the environmental bindings created during a function call. Closures for anonymous functions (lambdas) are stored similarly, but do not include a binding for a function name (arrow from "defun env" to "closure" in Figure A.5, top left). When a function is called, a new namespace is created to store argument bindings ("call env"), and is associated with the namespace that was active when the function was defined ("defun environment"). The caller's argument expressions are evaluated and bound to the variables listed in the function definition, which are retrieved from the closure memory structure ("args list"). The body expression is then retrieved and evaluated with the bindings in scope. Upon completion, the resulting *mem* state is returned to the caller, and the caller's namespace is retrieved from the stack (not shown).

A.9 Model Parameters for NeuroLISP Testing

The model parameters used for the tests outlined in Section 4.2 are listed in Table A.3. Cells labeled "variable" were experimentally varied (see Sections 4.1.4 and 4.2). The sizing of the stack regions was set according to the demands of the test, as determined by the NeuroLISP emulator. Either 256 or 1024 neurons was used (four times the number of orthogonal patterns representing stack frames, for stability purposes). The *op* and *gate sequence* regions were sized according to the number of states necessary for the interpreter firmware sequences (four neurons

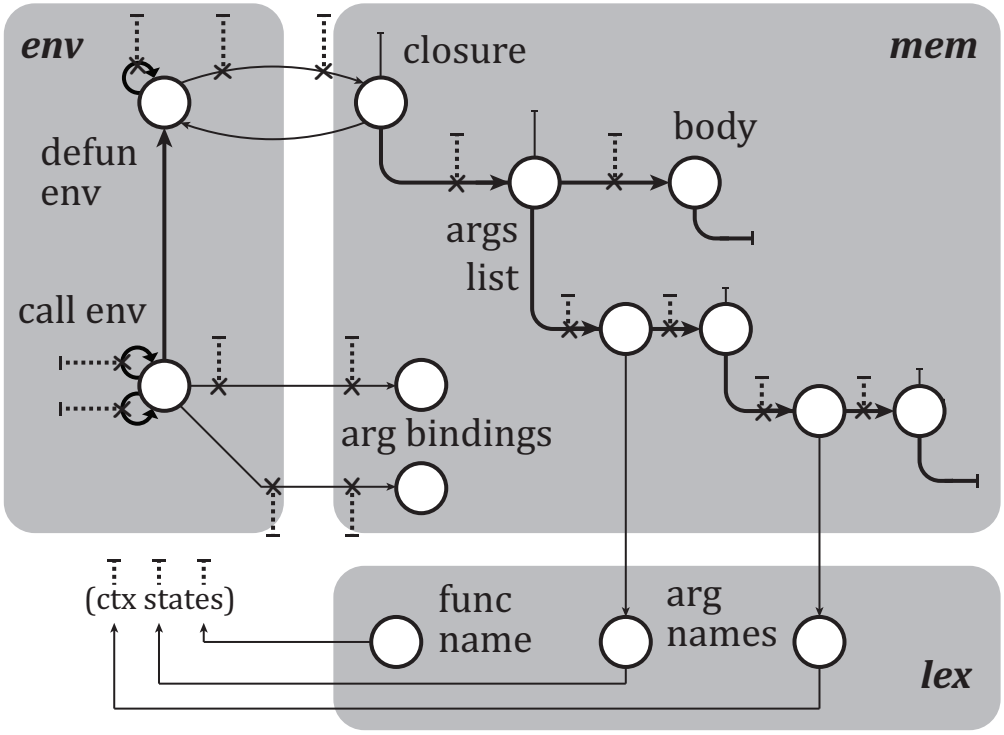


Figure A.5: Graphical depiction of the learned associations that make up closures and argument bindings for function calls. A closure is a cons cell (center top state labeled "closure") containing a list of argument names ("args list", center), and an expression for the body of the function ("body", top right). The closure state is also associated with the environmental namespace that was active when the function was defined ("defun env", top left). In return, the closure is bound to the function name within this environment; a *lex* state representing the function name contextualizes this binding ("func name", bottom). For simplicity, the context states are omitted from the diagram, and are abbreviated in the bottom left ("ctx states", see Figure A.4 for details). When the function is called, the interpreter creates a new environment to store argument bindings ("call env", left), evaluates sub-expressions for argument values, and binds them with their associated argument name ("arg bindings", center). Each binding is contextualized by the corresponding *lex* pattern representing the argument name ("arg" names, bottom right), which can be retrieved from the argument list contained in the closure.

Table A.3: Model parameters used for testing. "Size" refers to the number of neurons contained in a region. "Learning Type" refers to the source of learned patterns and associations ("flashed" memories are established during the one-time initialization process, and "combo" refers to a combination of flashed memories and memories learned online during model execution). "Pattern Type" refers to the organization of learned representations: "ortho" patterns are orthogonal vectors established during one-time initialization for high efficiency, "local" patterns are one-hot vectors (used in the gate output region to refer to individual model gates), and "random" patterns are generated with a Bernoulli process. "Activ Func" refers to the activation function used for neurons in the region. "Lambda" refers to the density parameter used for random pattern generation, and determines the probability of generating an individual neural activation value of 1.

Region	Size	Learning Type	Pattern Type	Activ Func	Lambda
runtime stack	256 / 1024	flashed	ortho	sign	N/A
data stack	256 / 1024	flashed	ortho	sign	N/A
op	1536	flashed	ortho	sign	N/A
gate sequence	496	flashed	ortho	sign	N/A
gate output	70	flashed	local	heaviside	N/A
lex	variable	combo	random	sign	0.5
mem	variable	combo	random	sign	0.5
mem-ctx	=size(mem)	combo	random	heaviside	0.25
env	variable	combo	random	sign	0.5
env-ctx	=size(env)	combo	random	heaviside	variable

per orthogonal state pattern). These regions use orthogonal patterns because orthogonality reduces the required region sizes ([97]), and because the learned states are only established during one-time initialization. Cells in Table A.3 labeled "flashed" indicate regions that learn solely during the one-time initialization process, which flashes the interpreter firmware. Cells labeled "combo" indicate regions that learn associations in an online fashion during model execution in addition to a small set of flashed associations (e.g., patterns for `NIL`, `true`, and `false` are flashed in the *mem* region, and a pattern is created for the default environment in the *env* region).

A.10 Multiway Tree Library

The code implementing the multiway tree functions (including helper functions) is listed in Figure A.6. The corresponding test cases are listed in Figure A.7.

A.11 Unification Test Case Generation

Unification test cases were produced by randomly generating trees and converting them to s-expressions as follows. First, a random tree is generated using the `random_tree` function in the Python `networkx` package[1]. The size of the initial tree is the expression complexity parameter described in Section 4.2.5 (x-axis of Figures 4.12 and 4.13). The tree is rooted by selecting the node with the greatest number of edges, the leaf nodes are labeled with randomly generated symbols, and the tree is copied to produce an identical pair. Then, a set of variable substitutions is generated, each mapping a variable name to a small randomly generated subtree representing the value of that variable. These substitutions are introduced to the pair of trees by randomly selecting a leaf node,

[1] https://networkx.org/documentation/stable/reference/generated/networkx.generators.trees.random_tree.html

```
(defun expr-equal? (x y)
  (cond ((or (atom x) (atom y)) (eq x y))
    ((and (listp x) (listp y))
      (and (expr-equal? (car x) (car y))
           (expr-equal? (cdr x) (cdr y))))
    (true false)))

(defun tree? (expr)
  (or (atom expr)
    (and (listp expr) (atom (car expr))
      (cdr expr) (forest? (cdr expr)))))
(defun forest? (expr)
  (or (not expr)
    (and (tree? (car expr))
      (forest? (cdr expr)))))

(defun copy-tree (tree)
  (if (atom tree) tree
    (cons (car tree)
      (copy-forest (cdr tree)))))
(defun copy-forest (subtrees)
  (if (not subtrees) NIL
    (cons (copy-tree (car subtrees))
      (copy-forest (cdr subtrees)))))

(defun tree-member (elm tree)
  (cond ((atom tree) (eq elm tree))
    (true (or (eq (car tree) elm)
      (forest-member elm (cdr tree))))))
(defun forest-member (elm forest)
  (and forest
    (or (tree-member elm (car forest))
      (forest-member elm (cdr forest)))))

(defun tree-prefix (tree)
  (tree-prefix-helper tree NIL))
(defun tree-prefix-helper (tree seq)
  (if (atom tree)
    (cons tree seq)
    (cons (car tree)
      (forest-prefix-helper
        (cdr tree) seq))))
(defun forest-prefix-helper (subtrees seq)
  (if subtrees
    (tree-prefix-helper
      (car subtrees)
      (forest-prefix-helper
        (cdr subtrees) seq))
      seq))
```

```
(defun tree-subst (new old tree)
  (let
    ((ret (tree-subst-helper new old tree)))
    (if ret ret tree)))
(defun tree-subst-helper (new old tree)
  (cond
    ((expr-equal? tree old) new)
    ((atom tree) NIL)
    (true
      (let ((subtrees (forest-subst-helper new old (cdr tree))))
        (if subtrees
          (cons (car tree) subtrees)
          NIL)))))
(defun forest-subst-helper (new old subtrees)
  (if (not subtrees) NIL
    (let ((curr (tree-subst-helper new old (car subtrees)))
          (rest (forest-subst-helper new old (cdr subtrees))))
      (if (or curr rest)
        (cons (if curr curr (car subtrees))
          (if rest rest (cdr subtrees)))
        NIL))))

(defun tree-sublis (subs tree)
  (let ((ret (tree-sublis-helper subs tree)))
    (if ret ret tree)))
(defun tree-sublis-replace (subs tree)
  (if subs
    (if (expr-equal? (car (car subs)) tree)
      (cadr (car subs))
      (tree-sublis-replace (cdr subs) tree))
    NIL))
(defun tree-sublis-helper (subs tree)
  (let ((replacement
          (tree-sublis-replace subs tree)))
    (cond
      (replacement replacement)
      ((atom tree) NIL)
      (true
        (let ((subtrees (forest-sublis-helper subs (cdr tree))))
          (if subtrees
            (cons (car tree) subtrees)
            NIL))))))
(defun forest-sublis-helper (subs subtrees)
  (if (not subtrees) NIL
    (let ((curr (tree-sublis-helper subs (car subtrees)))
          (rest (forest-sublis-helper subs (cdr subtrees))))
      (if (or curr rest)
        (cons
          (if curr curr (car subtrees))
          (if rest rest (cdr subtrees)))
        NIL))))
```

Figure A.6: Library of multiway tree processing functions.

```
(defun test (expr target)
  (if (not (eq (eval expr) target))
    (error (list target 'NOT_EQUAL expr))))

(test '(expr-equal? 'a 'b) false)
(test '(expr-equal? 'a 'a) true)
(test '(expr-equal? '(a (b c)) '(a (b c))) true)
(test '(expr-equal? '(a (b (c))) '(a (b c))) false)

(setq tree1 'a)
(setq tree2 '(a b))
(setq tree3 '(a (b c)))
(setq tree4 '(b d e))
(setq tree5 '(a (f g) c (b d e)))
(setq tree6 '(x y z))
(setq nottree1 '(a))
(setq nottree2 '(a (b c) (d) e))
(setq nottree3 '((a b c) (d e)))

(test '(is-tree? tree1) true)
(test '(is-tree? tree2) true)
(test '(is-tree? tree3) true)
(test '(is-tree? tree4) true)
(test '(is-tree? tree5) true)
(test '(is-tree? nottree1) false)
(test '(is-tree? nottree2) false)
(test '(is-tree? nottree3) false)

(test '(tree-contains? 'a tree1) true)
(test '(tree-contains? 'd tree5) true)
(test '(tree-contains? 'h tree5) false)

(test '(expr-equal? (tree-prefix tree1) '(a)) true)
(test '(expr-equal? (tree-prefix tree2) '(a b)) true)
(test '(expr-equal? (tree-prefix tree3) '(a b c)) true)
(test '(expr-equal? (tree-prefix tree4) '(b d e)) true)
(test '(expr-equal? (tree-prefix tree5) '(a f g c b d e)) true)

(test '(expr-equal? (tree-subst 'z 'a tree1) 'z) true)
(test '(expr-equal? (tree-subst '(z a b) 'a tree1) '(z a b)) true)
(test '(expr-equal? (tree-subst 'z '(b c) tree3) '(a z)) true)
(test '(expr-equal? (tree-subst 'z 'g tree5) '(a (f z) c (b d e))) true)

(setq subs '((a (x y z)) ((b d e) y) (c z)))
(test '(expr-equal? (tree-sublis subs tree1) '(x y z)) true)
(test '(expr-equal? (tree-sublis subs tree5) '(a (f g) z y)) true)
(test '(expr-equal? (tree-sublis subs tree6) tree6) true)

(test '(expr-equal? (copy-tree tree1) tree1) true)
(test '(expr-equal? (copy-tree tree5) tree5) true)
```

Figure A.7: Test cases for multiway tree processing functions.

and replacing it with a variable name in one tree, and the corresponding value subtree. In 20% of the test cases, a mismatch was introduced to the pair by randomly mutating a node in one tree such that they can no longer be unified. Below is an example of a randomly generated test case with an initial tree size of 10 nodes, leaf symbols drawn from a-j, variable names drawn from V-Z, up to 3 variable substitutions, and a maximum variable value size of up to 5 nodes.

Initial expression:

```
(((c) (f h)) (b) i)
```

Substitutions:

```
V ⟶ ((g) j i)

W ⟶ i

Y ⟶ (g a)
```

Expression pair with substitutions:

```
((((var W)) (f ((g) j i))) (b) (var Y))

(((i) (f (var V))) (b) (g a))
```

Mismatch mutation (optional):

```
((((var W)) (f ((g) j i))) (b) (var Y))

(((i) (f (var V))) (d) (g a))
```

Bibliography

[1] Pieter Abbeel, Adam Coates, and Andrew Y Ng. Autonomous helicopter aerobatics through apprenticeship learning. *The International Journal of Robotics Research*, 29(13):1608–1639, 2010.

[2] Kenneth Aizawa. The productivity of thought. In *The Systematicity Arguments*, pages 43–55. Springer, 2003.

[3] Daniel J Amit. *Modeling brain function: The world of attractor neural networks*. Cambridge university press, 1992.

[4] Daniel J Amit, Hanoch Gutfreund, and Haim Sompolinsky. Storing infinite numbers of patterns in a spin-glass model of neural networks. *Physical Review Letters*, 55(14):1530, 1985.

[5] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 39–48, 2016.

[6] Brenna Argall, Brett Browning, and Manuela Veloso. Learning mobile robot motion control from demonstrated primitives and human feedback. In *Robotics Research*, pages 417–432. Springer, 2011.

[7] Craig Atkinson, Brendan McCane, Lech Szymanski, and Anthony Robins. Pseudo-rehearsal: Achieving deep reinforcement learning without catastrophic forgetting. *Neurocomputing*, 428:291–307, 2021.

[8] Jimmy Ba, Geoffrey E Hinton, Volodymyr Mnih, Joel Z Leibo, and Catalin Ionescu. Using fast weights to attend to the recent past. In *Advances in Neural Information Processing Systems*, pages 4331–4339, 2016.

[9] Joris Baan, Jana Leible, Mitja Nikolaus, David Rau, Dennis Ulmer, Tim Baumgärtner, Dieuwke Hupkes, and Elia Bruni. On the realization of compositionality in neural networks. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 127–137, 2019.

[10] Bernard J Baars. The global workspace theory of consciousness. *The Blackwell companion to consciousness*, pages 236–246, 2007.

[11] Bernard J Baars and Stan Franklin. How conscious experience and working memory interact. *Trends in Cognitive Sciences*, 7(4):166–172, 2003.

[12] R Harald Baayen. Productivity in language production. *Language and Cognitive Processes*, 9(3):447–469, 1994.

[13] Alan Baddeley. Working memory and conscious awareness. *Theories of Memory*, pages 11–28, 1993.

[14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015*, 2015.

[15] Dare A Baldwin and Jodie A Baird. Discerning intentions in dynamic human action. *Trends in cognitive sciences*, 5(4):171–178, 2001.

[16] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[17] Albert Bandura. *Psychological modeling: Conflicting theories*. Transaction Publishers, New Jersey, USA, 2017.

[18] Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher order logic theorem proving. In *International Conference on Machine Learning*, pages 454–463. PMLR, 2019.

[19] Joao Barbosa, Heike Stein, Rebecca Martinez, Adria Galan, Kirsten Adam, Shai Li, Josep Valls-Solé, Christos Constantinidis, and Albert Compte. Interplay between persistent activity and activity-silent dynamics in prefrontal cortex during working memory. *bioRxiv*, page 763938, 2019.

[20] João José Oliveira Barros, Vítor Manuel Ferreira dos Santos, and Filipe Miguel Teixeira Pereira da Silva. Bimanual haptics for humanoid robot teleoperation using ros and v-rep. In *2015 IEEE International Conference on Autonomous Robot Systems and Competitions*, pages 174–179. IEEE, 2015.

[21] Lawrence W Barsalou. Flexibility, structure, and linguistic vagory in concepts: Manifestations of compositional system of perceptual symbols. *Theories of Memory*, page 29, 1993.

[22] Marcus K Benna and Stefano Fusi. Computational principles of synaptic memory consolidation. *Nature neuroscience*, 19(12):1697–1706, 2016.

[23] Tarek R Besold, Artur d'Avila Garcez, Sebastian Bader, Howard Bowman, Pedro Domingos, Pascal Hitzler, Kai-Uwe Kühnberger, Luis C Lamb, Daniel Lowd, Priscila Machado Vieira Lima, et al. 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*, 2015.

[24] David Bieber, Charles Sutton, Hugo Larochelle, and Daniel Tarlow. Learning to execute programs with instruction pointer attention graph neural networks. *arXiv preprint arXiv:2010.12621*, 2020.

[25] Elie Bienenstock, Stuart Geman, and Daniel Potter. Compositionality, MDL priors, and object recognition. In *Advances in Neural Information Processing Systems*, pages 838–844, 1997.

[26] Aude Billard, Sylvain Calinon, Ruediger Dillmann, and Stefan Schaal. Survey: Robot programming by demonstration. Technical report, Springer, 2008.

[27] Aude G Billard, Sylvain Calinon, and Rüdiger Dillmann. Learning from humans. In *Springer handbook of robotics*, pages 1995–2014. Springer, 2016.

[28] Peter Blouw, Eugene Solodkin, Paul Thagard, and Chris Eliasmith. Concepts as semantic pointers: A framework and computational model. *Cognitive Science*, 40(5):1128–1162, 2016.

[29] Roman Borisyuk, David Chik, Yakov Kazanovich, and João da Silva Gomes. Spiking neural network model for memorizing sequences with forward and backward recall. *Biosystems*, 112(3):214–223, 2013.

[30] Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a differentiable forth interpreter. In *International conference on machine learning*, pages 547–556. PMLR, 2017.

[31] Matthew M Botvinick. Hierarchical models of behavior and prefrontal function. *Trends in cognitive sciences*, 12(5):201–208, 2008.

[32] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

[33] Rudy R Bunel, Alban Desmaison, Pawan K Mudigonda, Pushmeet Kohli, and Philip Torr. Adaptive neural compilation. *Advances in Neural Information Processing Systems*, 29: 1444–1452, 2016.

[34] Michael Burke, Svetlin Penkov, and Subramanian Ramamoorthy. From explanation to synthesis: Compositional program induction for learning from demonstration. *Robotics: Science and Systems XV*, June 2019. doi: 10.15607/RSS.2019.XV.015.

[35] Guillermo Campitelli, Fernand Gobet, Kay Head, Mark Buckley, and Amanda Parker. Brain localization of memory chunks in chessplayers. *International Journal of Neuroscience*, 117 (12):1641–1659, 2007.

[36] Diogo V Carvalho, Eduardo M Pereira, and Jaime S Cardoso. Machine learning interpretability: A survey on methods and metrics. *Electronics*, 8(8):832, 2019.

[37] Daniel C Castro, Ian Walker, and Ben Glocker. Causality matters in medical imaging. *Nature Communications*, 11(1):1–10, 2020.

[38] Timur Chabuk and James A. Reggia. The added value of gating in evolved neurocontrollers. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, August 2013. doi: 10.1109/IJCNN.2013.6706895.

[39] Eugene Charniak and Robert P Goldman. A bayesian model of plan recognition. *Artificial Intelligence*, 64(1):53–79, 1993.

[40] Antonio Chella, Haris Dindo, and Ignazio Infantino. Learning high-level tasks through imitation. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3648–3654. IEEE, 2006.

[41] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[42] Roberto Colom, Irene Rebollo, Antonio Palacios, Manuel Juan-Espinosa, and Patrick C Kyllonen. Working memory is (almost) perfectly predicted by g. *Intelligence*, 32(3):277–296, 2004.

[43] Andrew RA Conway, Michael J Kane, and Randall W Engle. Working memory capacity and its relation to general intelligence. *Trends in cognitive sciences*, 7(12):547–552, 2003.

[44] Nelson Cowan. The magical mystery four: How is working memory capacity limited, and why? *Current directions in psychological science*, 19(1):51–57, 2010.

[45] Ivo Danihelka, Greg Wayne, Benigno Uria, Nal Kalchbrenner, and Alex Graves. Associative long short-term memory. In *International Conference on Machine Learning*, pages 1986–1994, 2016.

[46] Gregory P Davis, Garrett E Katz, Daniel Soranzo, Nathaniel Allen, Matthew J Reinhard, Rodolphe J Gentili, Michelle E Costanzo, and James A Reggia. A neurocomputational model of posttraumatic stress disorder. In *2021 10th International IEEE/EMBS Conference on Neural Engineering (NER)*, pages 107–110. IEEE, 2021.

[47] Stanislas Dehaene and Jean-Pierre Changeux. A hierarchical neuronal network for planning behavior. *Proceedings of the National Academy of Sciences*, 94(24):13293–13298, 1997.

[48] Marc Denecker and Antonis Kakas. Abduction in logic programming. In *Computational logic: Logic programming and beyond*, pages 402–436. Springer, 2002.

[49] Travis DeVautl, Seth Forrest, Ian Tanimoto, Terence Soule, and Robert Heckendorn. Learning from demonstration for distributed, encapsulated evolution of autonomous outdoor robots. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1381–1382. ACM, 2015.

[50] Haris Dindo, Antonio Chella, Giuseppe La Tona, Monica Vitali, Eric Nivel, and Kristinn R Thórisson. Learning problem solving skills from demonstration: An architectural approach. In *International Conference on Artificial General Intelligence*, pages 194–203. Springer, 2011.

[51] Jiafei Duan, Arijit Dasgupta, Jason Fischer, and Cheston Tan. A survey on machine learning approaches for modelling intuitive physics. *arXiv preprint arXiv:2202.06481*, 2022.

[52] Joshua T Dudman and Charles R Gerfen. The basal ganglia. In *The rat nervous system*, pages 391–440. Elsevier, 2015.

[53] Boris Durán, Yulia Sandamirskaya, and Gregor Schöner. A dynamic field architecture for the generation of hierarchically organized sequences. In *International Conference on Artificial Neural Networks*, pages 25–32. Springer, 2012.

[54] Chris Eliasmith, Terrence C Stewart, Xuan Choo, Trevor Bekolay, Travis DeWolf, Yichuan Tang, and Daniel Rasmussen. A large-scale model of the functioning brain. *Science*, 338 (6111):1202–1205, 2012.

[55] Wolfram Erlhagen and Gregor Schöner. Dynamic field theory of movement preparation. *Psychological Review*, 109(3):545, 2002.

[56] Kutluhan Erol. *Hierarchical task network planning: formalization, analysis, and implementation*. PhD thesis, 1996.

[57] Christoph Feichtenhofer, Axel Pinz, and Andrew Zisserman. Convolutional two-stream network fusion for video action recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1933–1941, 2016.

[58] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The racket manifesto. *1st Summit on Advances in Programming Languages*, page 113, 2015.

[59] Tesca Fitzgerald, Ashok K Goel, and Andrea L Thomaz. Representing skill demonstrations for adaptation and transfer. In *2014 AAAI Fall Symposium Series*, 2014.

[60] Pierre Flener. Inductive logic program synthesis with dialogs. In *International Conference on Inductive Logic Programming*, pages 175–198. Springer, 1996.

[61] Jerry A. Fodor and Zenon W. Pylyshyn. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28(1):3–71, March 1988. ISSN 0010-0277. doi: 10.1016/0010-0277(88)90031-5.

[62] Leonardo Fogassi, Pier Francesco Ferrari, Benno Gesierich, Stefano Rozzi, Fabian Chersi, and Giacomo Rizzolatti. Parietal lobe: from action organization to intention understanding. *Science*, 308(5722):662–667, 2005.

[63] Robert M French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

[64] Abram L Friesen and Rajesh PN Rao. Imitation learning with hierarchical actions. In *2010 IEEE 9th International Conference on Development and Learning*, pages 263–268. IEEE, 2010.

[65] Stefano Fusi, Patrick J Drew, and Larry F Abbott. Cascade models of synaptically stored memories. *Neuron*, 45(4):599–611, 2005.

[66] Ross W Gayler. Vector symbolic architectures answer jackendoff's challenges for cognitive neuroscience. In *International Conference on Cognitive Science*. Citeseer, 2003.

[67] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Elsevier, 2004.

[68] Leilani H Gilpin, David Bau, Ben Z Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. Explaining explanations: An overview of interpretability of machine learning. In *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*, pages 80–89. IEEE, 2018.

[69] Robert P Goldman, Christopher W Geib, and Christopher A Miller. A new model of plan recognition. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 245–254. Morgan Kaufmann Publishers Inc., 1999.

[70] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[71] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[72] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[73] Theresa C Hauge, Garrett E Katz, Gregory P Davis, Kyle J Jaquess, Matthew J Reinhard, Michelle E Costanzo, James A Reggia, and Rodolphe J Gentili. 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(aop):1–20, 2019.

[74] Theresa C Hauge, Garrett E Katz, Gregory P Davis, Di-Wei Huang, James A Reggia, and Rodolphe J Gentili. High-level motor planning assessment during performance of complex action sequences in humans and a humanoid robot. *International Journal of Social Robotics*, pages 1–18, 2020.

[75] Rich Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, pages 1–1, 2008.

[76] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *Advances in neural information processing systems*, pages 4565–4573, 2016.

[77] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[78] Osamu Hoshino, Noriaki Usuba, Yoshiki Kashimori, and Takeshi Kambara. Role of itinerancy among attractors as dynamical map in distributed coding scheme. *Neural Networks*, 10(8):1375–1390, 1997.

[79] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-learning in neural networks: A survey. *arXiv preprint arXiv:2004.05439*, 2020.

[80] Di-Wei Huang, Garrett Katz, Joshua Langsfeld, Rodolphe Gentili, and James Reggia. A virtual demonstrator environment for robot imitation learning. In *2015 IEEE International Conference on Technologies for Practical Robot Applications (TePRA)*, pages 1–6. IEEE, 2015.

[81] Dieuwke Hupkes, Anand Singh, Kris Korrel, German Kruszewski, and Elia Bruni. Learning compositionally through attentive guidance. *arXiv preprint arXiv:1805.09657*, 2018.

[82] Dieuwke Hupkes, Verna Dankers, Mathijs Mul, and Elia Bruni. Compositionality decomposed: How do neural networks generalise? *Journal of Artificial Intelligence Research*, 67:757–795, 2020.

[83] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):1–35, 2017.

[84] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. Deepmath-deep sequence models for premise selection. *Advances in Neural Information Processing Systems*, 29:2235–2243, 2016.

[85] Philip L Jackson, Andrew N Meltzoff, and Jean Decety. Neural circuits involved in imitation and perspective-taking. *Neuroimage*, 31(1):429–439, 2006.

[86] Susanne M Jaeggi, Martin Buschkuehl, John Jonides, and Walter J Perrig. Improving fluid intelligence with training on working memory. *Proceedings of the National Academy of Sciences*, 105(19):6829–6833, 2008.

[87] Bart Jansen and Tony Belpaeme. A computational model of intention reading in imitation. *Robotics and Autonomous Systems*, 54(5):394–402, 2006.

[88] Ole Jensen. Maintenance of multiple working memory items by temporal segmentation. *Neuroscience*, 139(1):237–249, 2006.

[89] Christopher Johansson, Martin Rehn, and Anders Lansner. Attractor neural networks with patchy connectivity. *Neurocomputing*, 69(7-9):627–633, 2006.

[90] Susan S Jones. The development of imitation in infancy. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 364(1528):2325–2335, 2009.

[91] Antonis C Kakas and Fabrizio Riguzzi. Abductive concept learning. *New Generation Computing*, 18(3):243–294, 2000.

[92] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186*, 2018.

[93] Hans Kamp and Barbara Partee. Prototype theory and compositionality. *Cognition*, 57(2): 129–191, 1995.

[94] Garrett Katz, Di-Wei Huang, Rodolphe Gentili, and James Reggia. An empirical characterization of parsimonious intention inference for cognitive-level imitation learning. In *Proceedings on the International Conference on Artificial Intelligence (ICAI)*, pages 83–89. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2017.

[95] Garrett Katz, Di-Wei Huang, Theresa Hauge, Rodolphe Gentili, and James Reggia. A novel parsimonious cause-effect reasoning algorithm for robot imitation and plan recognition. *IEEE Transactions on Cognitive and Developmental Systems*, 10(2):177–193, 2017.

[96] Garrett E Katz, Dale Dullnig, Gregory P Davis, Rodolphe J Gentili, and James A Reggia. Autonomous causally-driven explanation of actions. In *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 772–778. IEEE, 2017.

[97] Garrett E Katz, Gregory P Davis, Rodolphe J Gentili, and James A Reggia. A programmable neural virtual machine based on a fast store-erase learning rule. *Neural Networks*, 119:10–30, 2019.

[98] Garrett E Katz, Khushboo Gupta, and James A Reggia. Reinforcement-based program induction in a neural virtual machine. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020.

[99] Garrett E Katz, . Akshay, Gregory P Davis, Rodolphe J Gentili, and James A Reggia. Tunable neural encoding of a symbolic robotic manipulation algorithm. *Frontiers in Neurorobotics*, page 167, 2021.

[100] Ronald Kemker, Marc McClure, Angelina Abitino, Tyler Hayes, and Christopher Kanan. Measuring catastrophic forgetting in neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[101] Ross D King, Kenneth E Whelan, Ffion M Jones, Philip GK Reiser, Christopher H Bryant, Stephen H Muggleton, Douglas B Kell, and Stephen G Oliver. Functional genomic hypothesis generation and experimentation by a robot scientist. *Nature*, 427(6971):247, 2004.

[102] Ross D King, Jem Rowland, Stephen G Oliver, Michael Young, Wayne Aubrey, Emma Byrne, Maria Liakata, Magdalena Markham, Pinar Pir, Larisa N Soldatova, et al. The automation of science. *Science*, 324(5923):85–89, 2009.

[103] Thomas Kipf, Yujia Li, Hanjun Dai, Vinicius Zambaldi, Alvaro Sanchez-Gonzalez, Edward Grefenstette, Pushmeet Kohli, and Peter Battaglia. Compile: Compositional imitation learning and execution. In *International Conference on Machine Learning (ICML)*, pages 3418–3428, 2019.

[104] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

[105] Ekaterina Komendantskaya. Unification neural networks: unification by error-correction learning. *Logic Journal of the IGPL*, 19(6):821–847, 2011.

[106] Philip Koopman. *Stack Computers: the new wave*. Carnegie Mellon University, 1989.

[107] Moritz Köster, Miriam Langeloh, Christian Kliesch, Patricia Kanngiesser, and Stefanie Hoehl. Motor cortex activity during action observation predicts subsequent action imitation in human infants. *NeuroImage*, 218:116958, 2020.

[108] Oliver Kroemer, Christian Daniel, Gerhard Neumann, Herke Van Hoof, and Jan Peters. Towards learning hierarchical skills for multi-phase manipulation tasks. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1503–1510. IEEE, 2015.

[109] James R Kubricht, Keith J Holyoak, and Hongjing Lu. Intuitive physics: Current research and controversies. *Trends in cognitive sciences*, 21(10):749–759, 2017.

[110] Subhaneil Lahiri and Surya Ganguli. A memory frontier for complex synapses. *Advances in neural information processing systems*, 26:1034–1042, 2013.

[111] Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International Conference on Machine Learning*, pages 2873–2882, 2018.

[112] Brenden M Lake. Compositional generalization through meta sequence-to-sequence learning. In *Advances in Neural Information Processing Systems*, pages 9788–9798, 2019.

[113] Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and brain sciences*, 40, 2017.

[114] Anton E Lawson. The generality of hypothetico-deductive reasoning: Making scientific thinking explicit. *The American Biology Teacher*, 62(7):482–495, 2000.

[115] Anton E Lawson. How do humans acquire knowledge? and what does that imply about the nature of knowledge? *Science & Education*, 9(6):577–598, 2000.

[116] Hoang M Le, Nan Jiang, Alekh Agarwal, Miroslav Dudík, Yisong Yue, and Hal Daumé III. Hierarchical imitation and reinforcement learning. *arXiv preprint arXiv:1803.00590*, 2018.

[117] Pantelis Linardatos, Vasilis Papastefanopoulos, and Sotiris Kotsiantis. Explainable ai: A review of machine learning interpretability methods. *Entropy*, 23(1):18, 2021.

[118] Zachary C Lipton. The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery. *Queue*, 16(3):31–57, 2018.

[119] Joao Loula, Marco Baroni, and Brenden M Lake. Rearranging the familiar: Testing compositional generalization in recurrent networks. *arXiv preprint arXiv:1807.07545*, 2018.

[120] James MacGlashan and Michael L Littman. Between imitation and intention learning. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[121] Sanjay G Manohar, Nahid Zokaei, Sean J Fallon, Tim Vogels, and Masud Husain. Neural mechanisms of attending to items in working memory. *Neuroscience & Biobehavioral Reviews*, 2019.

[122] James A Marcum. An integrated model of clinical reasoning: dual-process theory of cognition and metacognition. *Journal of evaluation in clinical practice*, 18(5):954–961, 2012.

[123] Gary Marcus. Deep learning: A critical appraisal. *arXiv:1801.00631*, January 2018.

[124] Gary Marcus. The next decade in AI: four steps towards robust artificial intelligence. *arXiv preprint arXiv:2002.06177*, 2020.

[125] Gary F Marcus. *The algebraic mind: Integrating connectionism and cognitive science*. MIT Press, 2001.

[126] Nicolas Y Masse, Gregory D Grant, and David J Freedman. Alleviating catastrophic forgetting using context-dependent gating and synaptic stabilization. *Proceedings of the National Academy of Sciences*, 115(44):E10467–E10475, 2018.

[127] John McCarthy, Michael I Levin, Paul W Abrahams, Daniel J Edwards, and Timothy P Hart. *LISP 1.5 programmer's manual*. MIT press, 1965.

[128] Andrew N Meltzoff. Understanding the intentions of others: re-enactment of intended acts by 18-month-old children. *Developmental psychology*, 31(5):838, 1995.

[129] Andrew N Meltzoff, Patricia K Kuhl, Javier Movellan, and Terrence J Sejnowski. Foundations for a new science of learning. *Science*, 325(5938):284–288, 2009.

[130] Thomas Miconi, Kenneth Stanley, and Jeff Clune. Differentiable plasticity: training plastic neural networks with backpropagation. In *International Conference on Machine Learning*, pages 3559–3568, 2018.

[131] Paul Miller. Itinerancy between attractor states in neural systems. *Current Opinion in Neurobiology*, 40:14–22, 2016.

[132] Gianluigi Mongillo, Omri Barak, and Misha Tsodyks. Synaptic theory of working memory. *Science*, 319(5869):1543–1546, 2008.

[133] Ryan M Nefdt. A puzzle concerning compositionality in machines. *Minds and Machines*, pages 1–29, 2020.

[134] João Pedro Neto, Hava T Siegelmann, and J Félix Costa. Symbolic processing in neural networks. *Journal of the Brazilian Computer Society*, 8:58–70, 2003.

[135] Hwee Tou Ng and Raymond J Mooney. Abductive plan recognition and diagnosis: A comprehensive empirical evaluation. *KR*, 92:499–508, 1992.

[136] Peter Norvig. *Paradigms of artificial intelligence programming: case studies in Common LISP*. Morgan Kaufmann, 1992.

[137] Klaus Oberauer. Design for a working memory. *Psychology of learning and motivation*, 51: 45–100, 2009.

[138] Ziad Obermeyer and Ezekiel J Emanuel. Predicting the future—big data, machine learning, and clinical medicine. *The New England journal of medicine*, 375(13):1216, 2016.

[139] Takayuki Osa, Joni Pajarinen, Gerhard Neumann, J Andrew Bagnell, Pieter Abbeel, Jan Peters, et al. An algorithmic perspective on imitation learning. *Foundations and Trends® in Robotics*, 7(1-2):1–179, 2018.

[140] Erhan Oztop, Mitsuo Kawato, and Michael A Arbib. Mirror neurons: functions, mechanisms and models. *Neuroscience letters*, 540:43–55, 2013.

[141] Günther Palm. Neural associative memories and sparse coding. *Neural Networks*, 37:165–171, 2013.

[142] André Parent and Lili-Naz Hazrati. Functional anatomy of the basal ganglia. i. the cortico-basal ganglia-thalamo-cortical loop. *Brain research reviews*, 20(1):91–127, 1995.

[143] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural Networks*, 113:54–71, 2019.

[144] Judea Pearl. Theoretical impediments to machine learning with seven sparks from the causal revolution. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, pages 3–3, 2018.

[145] Francis Jeffry Pelletier. The principle of semantic compositionality. *Topoi*, 13(1):11–24, 1994.

[146] Yun Peng and James A Reggia. *Abductive inference models for diagnostic problem-solving*. Springer Science & Business Media, 1990.

[147] Trang Pham, Truyen Tran, and Svetha Venkatesh. Graph memory networks for molecular activity prediction. In *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 639–644. IEEE, 2018.

[148] Steven T Piantadosi, Joshua B Tenenbaum, and Noah D Goodman. The logical primitives of thought: Empirical foundations for compositional cognitive models. *Psychological Review*, 123(4):392, 2016.

[149] Tony A Plate. Holographic reduced representations. *IEEE Transactions on Neural networks*, 6(3):623–641, 1995.

[150] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *ACM SIGPLAN Notices*, volume 51, pages 522–538. ACM, 2016.

[151] Jordan B Pollack. Recursive distributed representations. *Artificial Intelligence*, 46(1-2): 77–105, 1990.

[152] Sindhu Raghavan and Raymond J Mooney. Abductive plan recognition by extending bayesian logic programs. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 629–644. Springer, 2011.

[153] Sindhu V Raghavan and Raymond J Mooney. Bayesian abductive logic programs. In *Workshops at the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[154] Kanaka Rajan, Christopher D Harvey, and David W Tank. Recurrent network models of sequence generation and memory. *Neuron*, 90(1):128–142, 2016.

[155] Harish Ravichandar, Athanasios S Polydoros, Sonia Chernova, and Aude Billard. Recent advances in robot learning from demonstration. *Annual Review of Control, Robotics, and Autonomous Systems*, 3:297–330, 2020.

[156] Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

[157] James Reggia, Di-Wei Huang, and Garrett Katz. Exploring the computational explanatory gap. *Philosophies*, 2(1):5, 2017.

[158] James A Reggia and Yun Peng. Modeling diagnostic reasoning: a summary of parsimonious covering theory. *Computer methods and programs in biomedicine*, 25(2):125–134, 1987.

[159] James A Reggia, Derek Monner, and Jared Sylvester. The computational explanatory gap. *Journal of Consciousness Studies*, 21(9-10):153–178, 2014.

[160] James A Reggia, Garrett E Katz, and Gregory P Davis. Humanoid cognitive robots that learn by imitating: implications for consciousness studies. *Frontiers in Robotics and AI*, 5: 1, 2018.

[161] James A Reggia, Garrett E Katz, and Gregory P Davis. Modeling working memory to identify computational correlates of consciousness. *Open Philosophy*, 2(1):252–269, 2019.

[162] James A Reggia, Garrett E Katz, and Gregory P Davis. Artificial conscious intelligence. *Journal of Artificial Intelligence and Consciousness*, 7(01):95–107, 2020.

[163] James A Reggia, Garrett E Katz, Gregory P Davis, and Rodolphe J Gentili. Avoiding catastrophic forgetting with short-term memory during continual learning. In *International Conference on Artificial Intelligence (ICAI)*, 2021.

[164] Carlo Reverberi, Kai Görgen, and John-Dylan Haynes. Compositionality of rule representations in human prefrontal cortex. *Cerebral cortex*, 22(6):1237–1246, 2012.

[165] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Model-agnostic interpretability of machine learning. *arXiv preprint arXiv:1606.05386*, 2016.

[166] Maximilian Riesenhuber and Tomaso Poggio. Hierarchical models of object recognition in cortex. *Nature Neuroscience*, 2(11):1019–1025, 1999.

[167] Rajeev V Rikhye, Aditya Gilra, and Michael M Halassa. Thalamic regulation of switching between cortical representations enables cognitive flexibility. *Nature Neuroscience*, 21(12): 1753–1763, 2018.

[168] Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. *Advances in Neural Information Processing Systems*, 30, 2017.

[169] Edmund T Rolls. Attractor networks. *Wiley Interdisciplinary Reviews: Cognitive Science*, 1(1):119–134, 2010.

[170] Nathan S Rose, Joshua J LaRocque, Adam C Riggall, Olivia Gosseries, Michael J Starrett, Emma E Meyering, and Bradley R Postle. Reactivation of latent working memories with transcranial magnetic stimulation. *Science*, 354(6316):1136–1139, 2016.

[171] Saul Rosen. Electronic computers: A historical survey. *ACM Computing Surveys (CSUR)*, 1(1):7–36, 1969.

[172] Michael E Rule, Timothy O'Leary, and Christopher D Harvey. Causes and consequences of representational drift. *Current opinion in neurobiology*, 58:141–147, 2019.

[173] Stuart Russell and Peter Norvig. *Artificial intelligence: a modern approach*. 2002.

[174] Yulia Sandamirskaya, Stephan KU Zibner, Sebastian Schneegans, and Gregor Schöner. Using dynamic field theory to extend the embodiment stance toward higher cognition. *New Ideas in Psychology*, 31(3):322–339, 2013.

[175] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In *International conference on machine learning*, pages 1842–1850. PMLR, 2016.

[176] Stefan Schaal. Is imitation learning the route to humanoid robots? *Trends in cognitive sciences*, 3(6):233–242, 1999.

[177] Bernhard Schölkopf, Francesco Locatello, Stefan Bauer, Nan Rosemary Ke, Nal Kalchbrenner, Anirudh Goyal, and Yoshua Bengio. Toward causal representation learning. *Proceedings of the IEEE*, 109(5):612–634, 2021.

[178] Peter Seibel. *Practical common lisp*. Apress, 2006.

[179] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[180] Parag Singla and Raymond J Mooney. Abductive markov logic for plan recognition. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[181] Jan Sprenger. Hypothetico-deductive confirmation. *Philosophy Compass*, 6(7):497–508, 2011.

[182] Kenneth O Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1(1):24–35, 2019.

[183] Terrence C Stewart, Trevor Bekolay, and Chris Eliasmith. Neural representations of compositional structures: Representing and manipulating vector spaces with spiking neurons. *Connection Science*, 23(2):145–153, 2011.

[184] Mark E Stickel. A prolog-like inference system for computing minimum-cost abductive explanations in natural-language interpretation. *Annals of Mathematics and Artificial Intelligence*, 4(1-2):89–105, 1991.

[185] Mark G Stokes. 'activity-silent'working memory in prefrontal cortex: a dynamic coding framework. *Trends in Cognitive Sciences*, 19(7):394–405, 2015.

[186] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3645–3650, 2019.

[187] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[188] Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. End-to-end memory networks. In *Advances in Neural Information Processing Systems*, pages 2440–2448, 2015.

[189] Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph Lim. Neural program synthesis from diverse demonstration videos. In *International Conference on Machine Learning*, pages 4797–4806, 2018.

[190] Jared Sylvester. *Neurocomputational Methods for Autonomous Cognitive Control*. PhD thesis, 2014.

[191] Jared Sylvester and James Reggia. Engineering neural systems for high-level problem solving. *Neural Networks*, 79:37–52, 2016.

[192] Jared Sylvester, Jim Reggia, and Scott Weems. Cognitive control as a gated cortical network. In *Proceedings second international conference on biologically-inspired cognitive architectures*, pages 371–376. IOP Press, 2011.

[193] Jared C Sylvester, James A Reggia, Scott A Weems, and Michael F Bunting. Controlling working memory with learned instructions. *Neural Networks*, 41:23–38, 2013.

[194] Zoltan Szabó. The case for compositionality. *The Oxford Handbook of Compositionality*, 64:80, 2012.

[195] Michael Tomasello, Ann Cale Kruger, and Hilary Horn Ratner. Cultural learning. *Behavioral and brain sciences*, 16(3):495–511, 1993.

[196] Lisa Torrey and Jude Shavlik. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pages 242–264. IGI Global, 2010.

[197] J Gregory Trafton, Nicholas L Cassimatis, Magdalena D Bugajska, Derek P Brock, Farilee E Mintz, and Alan C Schultz. Enabling effective human-robot interaction using perspective-taking in robots. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 35(4):460–470, 2005.

[198] Frank Van der Velde and Marc de Kamps. Neural blackboard architectures of combinatorial structures in cognition. *Behavioral and Brain Sciences*, 29(01):37–70, 2006.

[199] Joaquin Vanschoren. Meta-learning. In *Automated Machine Learning*, pages 35–61. Springer, Cham, 2019.

[200] Gül Varol, Ivan Laptev, and Cordelia Schmid. Long-term temporal convolutions for action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 40(6):1510–1517, 2017.

[201] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[202] Nicolas Vecoven, Damien Ernst, Antoine Wehenkel, and Guillaume Drion. Introducing neuromodulation in deep neural networks to learn adaptive behaviours. *PloS One*, 15(1): e0227922, 2020.

[203] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. *arXiv preprint arXiv:1804.02477*, 2018.

[204] Jane X Wang. Meta-learning in natural and artificial intelligence. *Current Opinion in Behavioral Sciences*, 38:90–95, 2021.

[205] Limin Wang, Yuanjun Xiong, Zhe Wang, Yu Qiao, Dahua Lin, Xiaoou Tang, and Luc Van Gool. Temporal segment networks: Towards good practices for deep action recognition. In *European conference on computer vision*, pages 20–36. Springer, 2016.

[206] Andrew P Witkin and Jay M Tenenbaum. On the role of structure in vision. In *Human and Machine Vision*, pages 481–543. Elsevier, 1983.

[207] Yan Wu, Yanyu Su, and Yiannis Demiris. A morphable template framework for robot learning by demonstration: Integrating one-shot and incremental learning approaches. *Robotics and Autonomous Systems*, 62(10):1517–1530, 2014.

[208] Danfei Xu, Suraj Nair, Yuke Zhu, Julian Gao, Animesh Garg, Li Fei-Fei, and Silvio Savarese. Neural task programming: Learning to generalize across hierarchical tasks. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8. IEEE, 2018.

[209] Yuichi Yamashita and Jun Tani. Emergence of functional hierarchy in a multiple timescale neural network model: a humanoid robot experiment. *PLoS Comput Biol*, 4(11):e1000220, 2008.

[210] Rafael Yuste. From the neuron doctrine to neural networks. *Nature reviews neuroscience*, 16(8):487, 2015.

[211] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.