Spaun 2.0: Extending the World's Largest Functional Brain Model

by

Feng-Xuan Choo

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2018

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	David C. Noelle Associate Professor, School of Social Sciences, Humanities, and Arts, University of California, Merced
Supervisor(s):	Chris Eliasmith Professor, Department of Philosophy and Department of Systems Design Engineering, University of Waterloo
Internal Member:	Jesse Hoey Associate Professor, David R. Cheriton School of Computer Science, University of Waterloo
Internal Member:	Olga Vechtomova Associate Professor, Department of Management Sciences, University of Waterloo
Internal Member:	Robin Cohen Professor, David R. Cheriton School of Computer Science, University of Waterloo
Internal-External Member:	Shi Cao Assistant Professor, Department of Systems Design Engineering, University of Waterloo

Abstract

Building large-scale brain models is one method used by theoretical neuroscientists to understand the way the human brain functions. Researchers typically use either a bottom-up approach, which focuses on the detailed modelling of various biological properties of the brain and places less importance on reproducing functional behaviour, or a top-down approach, which generally aim to reproduce the behaviour observed in real cognitive agents, but typically sacrifices adherence to constraints imposed by the neuro-biology. The focus of this thesis is Spaun, a large-scale brain model constructed using a combination of the bottom-up and top-down approaches to brain modelling. Spaun is currently the world's largest functional brain model, capable of performing eight distinct cognitive tasks ranging from digit recognition to inductive reasoning. The thesis is organized to discuss three aspects of the Spaun model.

First, it describes the original Spaun model, and explores how a top-down approach, known as the Semantic Pointer Architecture (SPA), has been combined with a bottom-up approach, known as the Neural Engineering Framework (NEF), to integrate six existing cognitive models into a unified cognitive model that is Spaun.

Next, the thesis identifies some of the concerns with the original Spaun model, and show the modifications made to the network to remedy these issues. It also characterizes how the Spaun model was re-organized and re-implemented (to include the aforementioned modifications) as the Spaun 2.0 model. As part of the discussion of the Spaun 2.0 model, task performance results are presented that compare the original Spaun model and the re-implemented Spaun 2.0 model, demonstrating that the modifications to the Spaun 2.0 model have improved its accuracy on the working memory task, and the two induction tasks.

Finally, three extensions to Spaun 2.0 are presented. These extensions take advantage of the re-organized Spaun model, giving Spaun 2.0 new capabilities – a motor system capable of adapting to unknown force fields applied to its arm; a visual system capable of processing 256×256 full-colour images; and the ability to follow general instructions.

The Spaun model and architecture presented in this thesis demonstrate that by using the SPA and the NEF, it is not only possible to construct functional large-scale brain models, but to do so in a manner that supports complex extensions to the model. The final Spaun 2.0 model consists of approximately 6.6 million neurons, can perform 12 cognitive tasks, and has been demonstrated to reproduce behavioural and neurological data observed in natural cognitive agents.

Acknowledgements

This thesis has been a long time in the making, and probably would not have been completed without the support of several awesome people.

First and foremost, I'd like to thank my supervisor, Chris Eliasmith. Chris is always a friend first, and a supervisor second, and he has always encouraged me to do my best, both in and outside of academia (I look forward to beating his high score in Space Pirate Trainer). He is also an incredible role-model and mentor, is knowledgeable beyond words, and always seems to have the right answer for any question.

Next, I'd like to thank my amazing colleagues at the CNRG lab, both past members of the lab (who are now current colleagues at ABR), and the current members of the lab. From the former category are Travis Dewolf, Daniel Rasmussen, and Trevor Bekolay, all of whom made great companions through the journey that has been my Master's and PhD research, and my current journey as a member of the ABR team. In the latter category are Terry Stewart, Jan Gosmann and Peter Duggins. Terry has been the resident "go-to" guy for questions regarding cognitive modelling, and both Jan and Pete make great house mates. I particularly enjoy the musical interludes of Pete as they do wonders in breaking the monotony of thesis writing.

Finally, thanks must be given to my family – my parents for providing me the opportunity to pursue this research, and my twin sister, for whom I can count on always being there.

Dedication

This thesis is dedicated to the tens of thousands (if not hundreds of thousands) of artificial brain models that have been spawned and terminated, all in the name of science. Thankfully, they remain blissfully unaware of their short but meaningful existences, nor their crucial contributions to furthering this research.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Understanding the inner workings of the human brain is, perhaps, the main goal of theoretical neuroscience. One method of achieving this goal is by building computational models that the modeller believes to mimic processes in the brain, and by comparing the behaviour of these models to that of the real brain.

The Blue Brain project [Markram, 2006] and the SyNAPSE project [Modha et al., 2011] take a bottom-up approach to achieving this goal. The Blue Brain project (now Human Brain Project) attempts to replicate many of the biological properties (e.g. the 3-dimensional structure, location of individual synapse, and neuronal connectivity) of a particular arrangement of neurons (a neocortical column) located in one specific location in cortex (the rat somatosensory cortex). Their hope is that emergent behaviour, eventually to the point of intelligence, will manifest as they increase the complexity of their simulations [Markram, 2006]. The SyNAPSE project, in contrast, uses less complex neuron models in their networks. However, they incorporate a learning mechanism known as spike-time-dependent plasticity (STDP) and use their models to explore the computational requirements needed for running large brain-sized models. While both of these projects have succeeded in simulating brain models containing large number of neurons (Human Brain Project – approximately 1 million neurons; SyNAPSE – approximately 1 billion neurons), these approaches have been criticized as unlikely to succeed given the complexity of the brain, their lack of focus on function, and our uncertainty about how best to identify relevant levels of explanation for different behaviours [Eliasmith and Trujillo, 2013].

An alternative approach, typically used by the cognitive modelling community, is to analyze the human brain from a behavioural perspective. Notably, the ACT-R [Anderson, 1996] and SOAR [Laird et al., 1987] communities have been able to accurately model human behaviours in tasks ranging from attention and perception, to problem solving and decision making. These approaches generally construct models by defining actions and language-like rules. These are used in conjunction with an architecture that recursively applies these actions and rules, similar to how computer processors are programmed in assembly. Consequently, these approaches generally ignore the underlying neurobiology, and importantly, disregard the constraints that the neurobiology can have on the behaviour of their models.

The cognitive models described in this thesis has been designed and implemented using a different approach to the ones mentioned above, one that combines both the low-level "neuron" approach and the high-level "behavioural" approach. The high-level approach is achieved using the Semantic Pointer Architecture (SPA) [Eliasmith, 2013], which defines a type of representation and a set of computations that can be used to perform cognitive tasks. It also specifies a standard architectural design for cognitive models. This approach is married with a low-level approach, through the use of the Neural Engineering Framework (NEF) [Eliasmith and Anderson, 2003]. The NEF defines a set of methods that dictate how mathematical functions can be converted into neural networks, while, importantly, adhering to the biological constraints of the underlying neural substrate. The SPA and NEF have been used to combine six existing neural models of different cognitive functions (e.g., vision, motor, memory, and so forth) into a unified network called the SPA Unified Network (Spaun) Eliasmith et al. [2012]. At the time of writing, Spaun is the world's largest behaviourally functional brain model, containing roughly 2.5 million neurons.

The purpose of this thesis is four-fold. First, it demonstrates how the NEF and SPA were used to integrate the six existing functional models together to form the Spaun model. Second, it addresses the majority of concerns with the original Spaun model, and shows the changes necessary to generalize the model. The changes to the original Spaun model have been combined to implement a new model, called "Spaun 2.0". Third, it demonstrates the general-use capabilities of the Spaun 2.0 model, through the exploration of three extensions implemented for the model. Last, and most importantly, this thesis serves as an example of how the NEF and SPA can enable an individual to construct large-scale neural networks capable of performing complex cognitive tasks. In the context of this thesis, all work described in this thesis has been developed by the author, with the exception of the NEF and SPA (developed by Chris Eliasmith), and the precursor networks described in Section 3.1 (developed by their respective authors).

1.1 Thesis Organization

This thesis is divided into six chapters, the first of which is this introduction. Next, Chapter 2 provides background on the SPA and NEF. The SPA provides the framework on which Spaun is constructed, by specifying the types of representations and computations that can be performed within the Spaun model, and by defining a standardized structure by the components of Spaun are designed around. In a similar fashion, the NEF specifies a set of principles with which the SPA computations can be converted into realized implementations in networks of spiking neurons.

Chapter 3 details the development and construction of the Semantic Pointer Architecture Unified Network (Spaun) model. The chapter begins with an exploration of the six networks that were used as the initial building blocks for the Spaun model. Next, a description of Spaun's original eight tasks are provided, each detailing how the SPA has been used to perform the computations necessary for each task. Finally, the architecture of the Spaun model is described, showing how the functionality required for each of Spaun's tasks has been implemented in the final Spaun model.

Chapter 4 focuses on a re-implementation of the original Spaun model. The re-implemented Spaun model (dubbed Spaun 2.0) is a result of work done to formalize the original Spaun model, and to address some of the major concerns regarding the original implementation. This chapter systematically progresses through each of Spaun's functional modules, and describe the modifications that have been made to improve the reliability, performance, and general usability of these components.

Chapter 5 explores three extensions constructed for the Spaun 2.0 network. These extensions enhance the functionality of Spaun, adding an adaptive motor controller capable of adapting to unknown force fields applied to its arm; the ability to classify 256×256 pixel full-colour images (as opposed to the original 28×28 pixel grayscale images); and the ability to follow general instructions.

Chapter 6 concludes this thesis with a summary of the contributions made, a discussion of several avenues for future work, and closing remarks.

Chapter 2

The Semantic Pointer Architecture

A naïve approach to combining existing neural models into a larger unified model, such as Spaun, would be to take the output of one neural network and feed it as the input to the next neural network, with some sort of "translation" layer in between. However, this approach cannot guarantee that the individual networks will communicate well with each other and it may be the case that the "translation" layer needs to be overly complex to ensure that the different aspects of the information being communicated (e.g., the content of the data, the timing of the data and control signals, etc.) match the specifications required for each network. In addition, the naïve approach does not leverage the commonalities between the individual networks to construct more efficient and streamlined models.

Another approach to building unified networks is to define a common framework that each of the individual models have to adhere to. This framework needs to define a set of "standards" (for computers, things like: data format, communication protocol, implementation of data processing algorithms, etc.) such that if each network were to follow these standards, it would be possible to construct the unified models with relative ease. Additionally, because each individual network is implemented with the same framework, network components that perform specific tasks (e.g., memory, decision making) should have almost identical implementations and can thus be reused, resulting in unified networks that are more efficient and generalizable than with the naïve approach. To that end, the Semantic Pointer Architecture (SPA) [Eliasmith, 2013] was developed as a framework to be used to build unified cognitive neural networks, while adhering to known neuro-anatomical and physiological constraints. The standards for this framework have thus been devised to result in biologically plausible, unified models.

2.1 Core SPA Principles

At its core, the SPA proposes that compressed representations – referred to as "semantic pointers" – are used to represent every form of information used within the framework. Additionally, the SPA specifies that compressive operators are used to generate semantic pointers, and to perform transformations between semantic pointers, with the latter enabling computations to be performed using the SPA.

In complex systems implemented using the SPA (e.g., Spaun), it is typical that each functionally distinct component (i.e., vision, motor control, etc.) use its own method for generating semantic pointers, and defines its own set of compressive operators to manipulate the semantic pointers. In the sections that follow, the discussion focuses on the SPA formulations used in the core cognitive components of Spaun as it forms the foundation of Spaun's cognitive functionality, with the perceptual and motor aspects of the architecture mirroring those described in [Eliasmith, 2013].

The "cognitive" aspects of the SPA can be understood from multiple viewpoints:

1. As a way to symbolically visualize the underlying, non-symbolic meaning and computation being performed on the information being represented within the unified network.
2. As a numerical implementation of the symbol-like visualization.
3. As a standardized structure that each component of the unified network follows when being physically implemented.

Each of these viewpoints is described in the sections that follow (Sections 2.2, 2.3, and 2.4 respectively). One aspect of network modelling that is not intrinsic to the SPA is the implementation of the SPA-conceptual network as a network of spiking neurons. For this, the Neural Engineering Framework (NEF) [Eliasmith and Anderson, 2003] is used (see Section 2.5.4).

2.2 The Semantic Pointer Architecture: Cognitive Computation with Symbol-like Representations

As alluded to in the introduction, Spaun is constructed as the amalgamation of six distinct models of difference brain functions (vision, motor control, memory, inductive reasoning, cognitive action planning and control, and learning). Since each of the individual models has its own form of representing information, each model was restructured to be used in the SPA, and thus speak the same "language". In this section, this SPA "language" is discussed in the context of the core cognitive components (memory, inductive reasoning, cognitive action planning and control) by using the symbolic representational form of these representations, despite the fact that these representations are implemented in spiking neurons in implemented SPA models. Additionally, this section explores how information (concepts) are represented in the SPA, and discusses how computation is performed on these symbolic forms.

2.2.1 SPA Concepts and SPA Expressions

As mentioned previously, all of the computation in the SPA is performed on abstract conceptual representational forms known as “semantic pointers”. These forms are “semantic” because when compared, semantic pointers will have higher similarity rankings when conceptually alike and lower similarity rankings when conceptually dissimilar (see Section 2.3.1), thus defining a local semantic space. These forms are also known as “pointers” because they behave similarly to the pointers used in coding languages (e.g., C, C++)(see Section 2.2.7.1). Specifically, these “pointers” can be “de-referenced” to extract additional information from the representations.

In this thesis, semantic pointers will be indicated by a word formatted in the following format: **CONCEPT**. As an example, the concept of the colour “blue” is represented in the SPA as:

$$\textsf{\textbf{BLUE}}$$

For the primary purpose of performing computation, SPA concepts are combined with algebraic and SPA operators (see succeeding sections) to form SPA expressions. As a basic example, using just the assignment operator (=), the conceptual assignment of “blue” as a “favourite colour” can be symbolically represented as:

$$\textsf{\textbf{FAV\_COLOUR}} = \textsf{\textbf{BLUE}}$$

2.2.2 SPA Binding

One of most fundamental concepts related to cognitive representations in the SPA is the concept of “binding”. Binding is an operation applied to two or more semantic pointers to produce a new semantic pointer that represents a concept with the combined properties of the bound semantic pointers. In this document, the symbol ($\circledast$) is used to represent the binding operation. As an example, the concept of a “blue square” can be constructed as the bound result of a “blue” and a “square” semantic pointer:

$$\textsf{\textbf{BLUE\_SQUARE}} = \textsf{\textbf{BLUE}} \circledast \textsf{\textbf{SQUARE}}.$$

It should be noted that the binding operator discussed here is assumed to be commutative, and as such (using the example above),

$$\begin{aligned}\textsf{\textbf{BLUE\_SQUARE}} &= \textsf{\textbf{BLUE}} \circledast \textsf{\textbf{SQUARE}} \\ &= \textsf{\textbf{SQUARE}} \circledast \textsf{\textbf{BLUE}}\end{aligned}$$

Practically, the binding operator is not only used to bind concrete concepts (e.g., “blue”, “red”, “square”, “triangle”) together to form new conceptual semantic pointers; it is also used to bind concrete and abstract concepts (e.g., “colour”, “shape”) together to form more complex representations (see Section 2.2.6).

The SPA binding operator does not limit what can be bound together and it is sometimes useful to bind identical semantic pointers together, creating “bound powers” of semantic pointers. As a shorthand, these bound powers are denoted with a superscript numeral indicating how many times the semantic pointer has been bound with itself. For example,

$$\begin{aligned}\textsf{\textbf{BLUE}} \circledast \textsf{\textbf{BLUE}} &= \textsf{\textbf{BLUE}}^2\text{, and} \\ \textsf{\textbf{BLUE}} \circledast \textsf{\textbf{BLUE}} \circledast \textsf{\textbf{BLUE}} &= \textsf{\textbf{BLUE}}^3\end{aligned}$$

2.2.3 SPA Collections

While the SPA binding operator combines semantic pointers to create new semantic pointers that merge the conceptual properties of their constituent semantic pointers, the SPA also defines a collection operator that serves to create semantic pointers that function more like groups of semantic pointers. In this thesis, the SPA collection operator is represented by the algebraic addition (+) operator.

An unordered list of numbers can be used to demonstrate the typical use of the SPA collection operator. For example, with the semantic pointer representations of the numbers $1, 2$, and 3 (**ONE**, **TWO**, **THREE**), the semantic pointer that represents the unordered list of these numbers can be created like so:

$$\textsf{\textbf{LIST}} = \textsf{\textbf{ONE}} + \textsf{\textbf{TWO}} + \textsf{\textbf{THREE}}$$

Mathematically, the SPA collection operator functions identically to the algebraic addition operator. That being the case, scalar numbers and the algebraic multiplication operator ($\times$) can be used with semantic pointers to denote collections of identical semantic pointers. For example,

$$\begin{aligned}\textsf{\textbf{ONE}} + \textsf{\textbf{ONE}} &= 2 \times \textsf{\textbf{ONE}}\text{, and} \\ \textsf{\textbf{ONE}} + \textsf{\textbf{ONE}} + \textsf{\textbf{TWO}} + \textsf{\textbf{TWO}} + \textsf{\textbf{TWO}} + \textsf{\textbf{TWO}} &= 2 \times \textsf{\textbf{ONE}} + 4 \times \textsf{\textbf{TWO}}\end{aligned}$$

It should also be noted that like the binding operator, the SPA collection operator is commutative.

2.2.4 Special Semantic Pointers

Apart from the standard semantic pointers described in Section 2.2.1, the SPA also defines two special semantic pointers (the identity semantic pointer and the null semantic pointer) with unique properties.

2.2.4.1 The Identity Semantic Pointer

The identity semantic pointer ($\mathbf{I}$) is defined such that binding a semantic pointer with the identity semantic pointer results in no change to the former semantic pointer. I.e.

$$\mathbf{CONCEPT} \circledast \mathbf{I} = \mathbf{CONCEPT} \tag{2.1}$$

2.2.4.2 The Null Semantic Pointer

The null semantic pointer ($\boldsymbol{\emptyset}$) is defined such that any semantic pointer bound with the null semantic pointer always results in the null semantic pointer. I.e.

$$\mathbf{CONCEPT} \circledast \boldsymbol{\emptyset} = \boldsymbol{\emptyset} \tag{2.2}$$

Because the definition above is not particularly useful for computational purposes, the null semantic pointer is usually also defined such that adding the null semantic pointer to an SPA collection results in no change to the collection:

$$\mathbf{CONCEPT} + \boldsymbol{\emptyset} = \mathbf{CONCEPT} \tag{2.3}$$

It should be noted that with the definition above, the null semantic pointer can be considered equivalent to an empty SPA collection.

2.2.5 SPA Extraction Operators

Thus far, the SPA operators presented (binding and collecting) serve to combine semantic pointers into new semantic pointers that are conceptually distinct from their constituent parts. In the SPA, it is also possible to take such a semantic pointer, and extract these constituent semantic pointers from them.

2.2.5.1 Unbinding

The unbinding operation details how to take a semantic pointer generated using the binding operation and extract information from it. In order to do so, the concept of a semantic pointer inverse must be introduced. The semantic pointer inverse is defined such that binding a semantic pointer and its inverse (denoted by a "¬" symbol preceding the semantic pointer) results in the identity semantic pointer:

$$\mathbf{CONCEPT} \circledast \neg\mathbf{CONCEPT} = \mathbf{I} \tag{2.4}$$

With this definition, it can be shown that extracting a semantic pointer from a bound result can be achieved by binding the bound result with the inverse of all semantic pointers that are to be excluded from the extraction operation. This is best illustrated using the bound result of three semantic pointers. For example, given

$$\mathbf{D} = \mathbf{A} \circledast \mathbf{B} \circledast \mathbf{C},$$

A can be extracted from **D** by binding **D** with the semantic pointer inverses of all semantic pointers excluded from the extraction operation (in this case, **B** and **C**),

$$\begin{aligned}\mathbf{D} \circledast \neg\mathbf{B} \circledast \neg\mathbf{C} &= (\mathbf{A} \circledast \mathbf{B} \circledast \mathbf{C}) \circledast \neg\mathbf{B} \circledast \neg\mathbf{C} \\ &= \mathbf{A} \circledast (\mathbf{B} \circledast \neg\mathbf{B}) \circledast (\mathbf{C} \circledast \neg\mathbf{C}) \\ &= \mathbf{A} \circledast \mathbf{I} \circledast \mathbf{I} = \mathbf{A}\end{aligned}$$

2.2.5.2 Un-collecting

Because the SPA collection operator behaves identically to the algebraic addition operator, semantic pointers can be extracted from an SPA collection by subtracting all semantic pointers that are to be excluded from the extraction operation from the SPA collection. For example, given

$$\mathbf{D} = \mathbf{A} + \mathbf{B} + \mathbf{C},$$

A can be extract from **D** by subtracting of all semantic pointers excluded from the extraction operation (in this case, **B** and **C**) from **D**,

$$\begin{aligned}\mathbf{D} - \mathbf{B} - \mathbf{C} &= (\mathbf{A} + \mathbf{B} + \mathbf{C}) - \mathbf{B} - \mathbf{C} \\ &= \mathbf{A} + (\mathbf{B} - \mathbf{B}) + (\mathbf{C} - \mathbf{C}) \\ &= \mathbf{A} + \emptyset + \emptyset = \mathbf{A}\end{aligned}$$

As demonstrated above, the key limitation for both the unbinding and un-collecting operations is that the extraction of semantic pointers can only take place if the semantic pointers to be excluded from the extraction are known. Without this information, it is impossible to deconstruct a semantic pointer into the semantic pointers that were used to create the combined result.

2.2.6 Complex SPA Representation and Computation

Thus far, the computations demonstrated have been relatively simple, consisting of the independent use of either the binding operator, or the collection operator. To demonstrate how the SPA can be used to represent more complex concepts, and perform more complex computations, the SPA will be used to describe Figure 2.1.

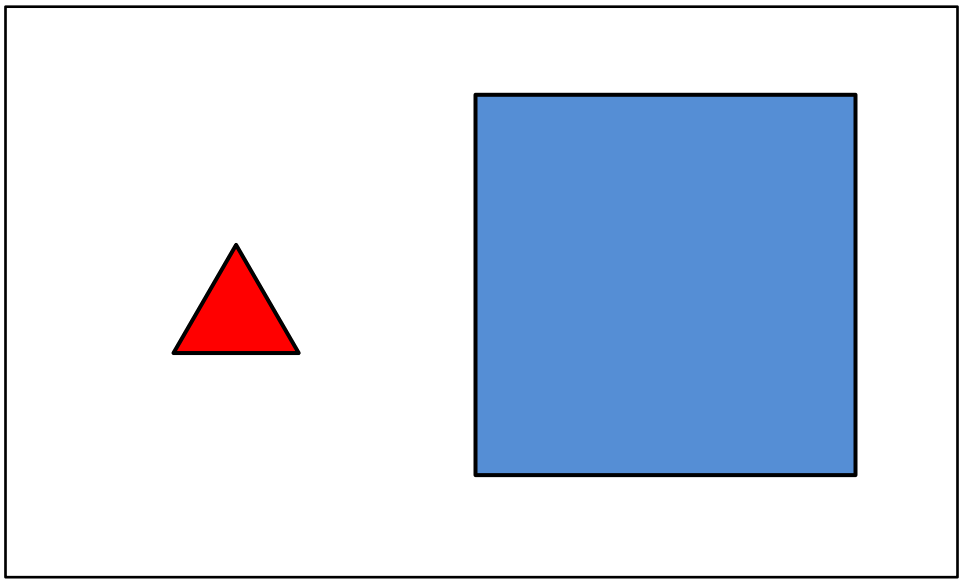


Figure 2.1: Diagram of a small red triangle and a big blue square used to demonstrate how the SPA is used to represent this scene and how to extract information of this scene from the SPA representation.

Before the SPA can be used to construct a representation of the figure, or perform computations on the SPA representation, a predefined vocabulary of semantic pointers must be identified. It is necessary to define this vocabulary to constrain the results of the SPA computations to produce meaningful concepts. The mechanism for performing this constraint procedure is further discussed in Sections 2.3.7 and 2.5.5.5. For the example in Figure 2.1, the SPA vocabulary is defined as: {**BLUE**, **RED**, **TRIANGLE**, **SQUARE**, **SMALL**, **BIG**}.

Considering just the shape and the colour of the objects in the figure, the SPA representation of the figure can be computed as such:

$$\textsf{\textbf{FIGURE}} = \textsf{\textbf{RED}} \circledast \textsf{\textbf{TRIANGLE}} + \textsf{\textbf{BLUE}} \circledast \textsf{\textbf{SQUARE}} \tag{2.5}$$

With this representation, it is possible to query the shape of blue object by binding the SPA representation of the figure with the inverse of the **BLUE** semantic pointer:

$$\begin{aligned}\textsf{\textbf{FIGURE}} \circledast \neg\textsf{\textbf{BLUE}} &= (\textsf{\textbf{RED}} \circledast \textsf{\textbf{TRIANGLE}} + \textsf{\textbf{BLUE}} \circledast \textsf{\textbf{SQUARE}}) \circledast \neg\textsf{\textbf{BLUE}} \\ &= (\textsf{\textbf{RED}} \circledast \textsf{\textbf{TRIANGLE}} \circledast \neg\textsf{\textbf{BLUE}}) + (\textsf{\textbf{BLUE}} \circledast \textsf{\textbf{SQUARE}} \circledast \neg\textsf{\textbf{BLUE}}) \\ &= (\textsf{\textbf{RED}} \circledast \textsf{\textbf{TRIANGLE}} \circledast \neg\textsf{\textbf{BLUE}}) + \textsf{\textbf{SQUARE}} \\ &\approx \textsf{\textbf{SQUARE}}\end{aligned} \tag{2.6}$$

Equation (2.6) demonstrates that binding the SPA representation of the figure with the inverse of **BLUE** results in an SPA collection comprised of **BLUE** and the superfluous semantic pointer (**RED** $\circledast$ **TRIANGLE** $\circledast$ $\neg$**BLUE**). While the result of the binding is not exactly **SQUARE** (because of the addition of the superfluous semantic pointer), computing the similarity between the result and **SQUARE** will result in a high similarity value, if the expected similarity between the **SQUARE** semantic pointer and the superfluous semantic pointer is low. This places some constraints on the specific operators that can be used for the binding operation (see Section 2.3.2).

The result of the binding operation can be improved by comparing it to the semantic pointers in the vocabulary. With this comparison the term (**RED** $\circledast$ **TRIANGLE** $\circledast$ $\neg$**BLUE**) is ignored (as it is not in one of the members of the vocabulary), and the resulting answer is exactly **SQUARE**. This operation of comparing a semantic pointer to a fixed vocabulary and choosing the semantic pointer(s) that best match the input is known as "cleanup" and will henceforth be represented in the SPA formulations as (*cleanup*{}). For the example figure, this is written as:

$$\textit{cleanup}\{(\textsf{\textbf{RED}} \circledast \textsf{\textbf{TRIANGLE}} \circledast \neg\textsf{\textbf{BLUE}}) + \textsf{\textbf{SQUARE}}\}$$

Since the semantic pointer (**RED** $\circledast$ **TRIANGLE** $\circledast$ $\neg$**BLUE**) is not a member of the vocabulary, it can be ignored, resulting in the desired answer of **SQUARE**:

$$\begin{aligned}&\textit{cleanup}\{(\textsf{\textbf{RED}} \circledast \textsf{\textbf{TRIANGLE}} \circledast \neg\textsf{\textbf{BLUE}}) + \textsf{\textbf{SQUARE}}\} \\ &\quad = [\cancel{(\textsf{\textbf{RED}} \circledast \textsf{\textbf{TRIANGLE}} \circledast \neg\textsf{\textbf{BLUE}})} + \textsf{\textbf{SQUARE}}] \\ &\quad = \textsf{\textbf{SQUARE}}\end{aligned}$$

While the SPA representation presented above is adequate for simple 2-concept forms, more complex formulations are required to represent more detailed information. For example, if the colour, shape, and size of the objects are considered, a simplistic SPA representation could be formulated as such:

$$\textsf{\textbf{FIGURE2}} = \textsf{\textbf{RED}} \circledast \textsf{\textbf{TRIANGLE}} \circledast \textsf{\textbf{SMALL}} + \textsf{\textbf{BLUE}} \circledast \textsf{\textbf{SQUARE}} \circledast \textsf{\textbf{BIG}}$$

An issue arises when information is extracted from the **FIGURE2** semantic pointer. In order to obtain the shape of the blue object, both the colour and the size of the object need to be provided. Working through the calculations, it can be shown that querying the **FIGURE2** semantic pointer with just the inverse of **BLUE** results in the bound result of both **SQUARE** and **BIG**:

$$\begin{aligned}
\textsf{\textbf{FIGURE2}} \circledast \neg\textsf{\textbf{BLUE}} &= (\textsf{\textbf{RED}} \circledast \textsf{\textbf{TRIANGLE}} \circledast \textsf{\textbf{SMALL}} + \textsf{\textbf{BLUE}} \circledast \textsf{\textbf{SQUARE}} \circledast \textsf{\textbf{BIG}}) \circledast \neg\textsf{\textbf{BLUE}} \\
&= (\textsf{\textbf{RED}} \circledast \textsf{\textbf{TRIANGLE}} \circledast \textsf{\textbf{SMALL}} \circledast \neg\textsf{\textbf{BLUE}}) + \\
&\quad\; (\textsf{\textbf{BLUE}} \circledast \textsf{\textbf{SQUARE}} \circledast \textsf{\textbf{BIG}} \circledast \neg\textsf{\textbf{BLUE}}) \\
&= (\textsf{\textbf{RED}} \circledast \textsf{\textbf{TRIANGLE}} \circledast \textsf{\textbf{SMALL}} \circledast \neg\textsf{\textbf{BLUE}}) + \textsf{\textbf{SQUARE}} \circledast \textsf{\textbf{BIG}}
\end{aligned}$$

Since the semantic pointer ($\textsf{\textbf{SQUARE}} \circledast \textsf{\textbf{BIG}}$) is not a member of the predefined vocabulary, feeding the result of ($\textsf{\textbf{FIGURE2}} \circledast \neg\textsf{\textbf{BLUE}}$) into the cleanup operation would result an empty SPA collection since no semantic pointer matches would be found.

For these more complex SPA representations, additional conceptual "tags" are commonly used. For the example above, the semantic pointers for each of the concepts of shape, colour and size (**SHAPE**, **COLOUR**, **SIZE**) are used in the following schema to generate the representations for each of the shapes:

$$\textsf{\textbf{OBJECT}} = \textsf{\textbf{SHAPE}} \circledast \mathit{shape} + \textsf{\textbf{COLOUR}} \circledast \mathit{colour} + \textsf{\textbf{SIZE}} \circledast \mathit{size} \tag{2.7}$$

With this schema, the SPA representation for the big blue square can be constructed as such:

$$\textsf{\textbf{BLUE\_SQUARE}} = \textsf{\textbf{SHAPE}} \circledast \textsf{\textbf{SQUARE}} + \textsf{\textbf{COLOUR}} \circledast \textsf{\textbf{BLUE}} + \textsf{\textbf{SIZE}} \circledast \textsf{\textbf{BIG}}$$

Querying one or multiple attributes from the SPA representation of a single object generated with schema (2.7) can be done as before, by binding the semantic pointer with the inverse of the semantic pointers to be queried (in this case, one of the "abstract concept" semantic pointers).

For example, querying the semantic pointer $\textsf{\textbf{BLUE\_SQUARE}}$ for its size and shape is as follows:

$$\begin{aligned}
&\textsf{\textbf{BLUE\_SQUARE}} \circledast (\neg\textsf{\textbf{SHAPE}} + \neg\textsf{\textbf{SIZE}})\\
&\quad= (\textsf{\textbf{SHAPE}} \circledast \textsf{\textbf{SQUARE}} + \textsf{\textbf{COLOUR}} \circledast \textsf{\textbf{BLUE}} + \textsf{\textbf{SIZE}} \circledast \textsf{\textbf{BIG}}) \circledast (\neg\textsf{\textbf{SHAPE}} + \neg\textsf{\textbf{SIZE}})\\
&\quad= [(\textsf{\textbf{SHAPE}} \circledast \textsf{\textbf{SQUARE}} \circledast \neg\textsf{\textbf{SHAPE}}) + (\textsf{\textbf{COLOUR}} \circledast \textsf{\textbf{BLUE}} \circledast \neg\textsf{\textbf{SHAPE}}) +\\
&\qquad\qquad (\textsf{\textbf{SIZE}} \circledast \textsf{\textbf{BIG}} \circledast \neg\textsf{\textbf{SHAPE}})] +\\
&\qquad [(\textsf{\textbf{SHAPE}} \circledast \textsf{\textbf{SQUARE}} \circledast \neg\textsf{\textbf{SIZE}}) + (\textsf{\textbf{COLOUR}} \circledast \textsf{\textbf{BLUE}} \circledast \neg\textsf{\textbf{SIZE}}) +\\
&\qquad\qquad (\textsf{\textbf{SIZE}} \circledast \textsf{\textbf{BIG}} \circledast \neg\textsf{\textbf{SIZE}})]\\
&\quad= [(\textsf{\textbf{SQUARE}}) + (\textsf{\textbf{COLOUR}} \circledast \textsf{\textbf{BLUE}} \circledast \neg\textsf{\textbf{SHAPE}}) + (\textsf{\textbf{SIZE}} \circledast \textsf{\textbf{BIG}} \circledast \neg\textsf{\textbf{SHAPE}})] +\\
&\qquad [(\textsf{\textbf{SHAPE}} \circledast \textsf{\textbf{SQUARE}} \circledast \neg\textsf{\textbf{SIZE}}) + (\textsf{\textbf{COLOUR}} \circledast \textsf{\textbf{BLUE}} \circledast \neg\textsf{\textbf{SIZE}}) + (\textsf{\textbf{BIG}})]
\end{aligned}$$

$$\begin{aligned}
&\mathit{cleanup}\{\textsf{\textbf{BLUE\_SQUARE}} \circledast (\neg\textsf{\textbf{SHAPE}} + \neg\textsf{\textbf{SIZE}})\}\\
&\quad= [(\textsf{\textbf{SQUARE}}) + \cancel{(\textsf{\textbf{COLOUR}} \circledast \textsf{\textbf{BLUE}} \circledast \neg\textsf{\textbf{SHAPE}})} + \cancel{(\textsf{\textbf{SIZE}} \circledast \textsf{\textbf{BIG}} \circledast \neg\textsf{\textbf{SHAPE}})}] +\\
&\qquad [\cancel{(\textsf{\textbf{SHAPE}} \circledast \textsf{\textbf{SQUARE}} \circledast \neg\textsf{\textbf{SIZE}})} + \cancel{(\textsf{\textbf{COLOUR}} \circledast \textsf{\textbf{BLUE}} \circledast \neg\textsf{\textbf{SIZE}})} + (\textsf{\textbf{BIG}})]\\
&\quad= \textsf{\textbf{SQUARE}} + \textsf{\textbf{BIG}}
\end{aligned}$$

Using schema (2.7) with multiple objects presents a minor problem. Given

$$\begin{aligned}
&\textsf{\textbf{RED\_TRIANGLE}} = \textsf{\textbf{SHAPE}} \circledast \textsf{\textbf{TRIANGLE}} + \textsf{\textbf{COLOUR}} \circledast \textsf{\textbf{RED}} + \textsf{\textbf{SIZE}} \circledast \textsf{\textbf{SMALL}}\\
&\qquad \text{and,}\\
&\textsf{\textbf{BLUE\_SQUARE}} = \textsf{\textbf{SHAPE}} \circledast \textsf{\textbf{SQUARE}} + \textsf{\textbf{COLOUR}} \circledast \textsf{\textbf{BLUE}} + \textsf{\textbf{SIZE}} \circledast \textsf{\textbf{BIG}},
\end{aligned}$$

the result of using the SPA collection operator to amalgamate the different semantic pointers into one semantic pointer is:

$$\begin{aligned}
\textsf{\textbf{FIGURE3}} = \textsf{\textbf{SHAPE}} \circledast \textsf{\textbf{TRIANGLE}} + \textsf{\textbf{COLOUR}} \circledast \textsf{\textbf{RED}} + \textsf{\textbf{SIZE}} \circledast \textsf{\textbf{SMALL}} +\\
\textsf{\textbf{SHAPE}} \circledast \textsf{\textbf{SQUARE}} + \textsf{\textbf{COLOUR}} \circledast \textsf{\textbf{BLUE}} + \textsf{\textbf{SIZE}} \circledast \textsf{\textbf{BIG}}
\end{aligned} \tag{2.8}$$

Since the SPA collection operator is commutative (i.e., the order of the operands do not matter), the SPA representation of a big blue triangle and a small red square (compared to the small red triangle and a big blue square in the example figure) will result in an identical set of tag-attribute pairs. In order to differentially group the semantic pointers that belong to one object or the other, additional conceptual object "tags" are required. For the example figure, the tags $\textsf{\textbf{POS}}_\triangle$ and $\textsf{\textbf{POS}}_\square$ will be used to denote the position of the object within the figure. Thus, the complete figure is represented as:

$$\mathsf{\mathbf{FIGURE4}} = \mathsf{\mathbf{POS}}_{\triangle} \circledast \mathsf{\mathbf{RED\_TRIANGLE}} + \mathsf{\mathbf{POS}}_{\square} \circledast \mathsf{\mathbf{BLUE\_SQUARE}} \tag{2.9}$$

A straightforward query for information from the SPA representation (2.9) requires two pieces of information: the position of the object in question, and the attribute to be queried. For example, extracting the colour of the leftmost object is computed as:

$$\begin{aligned}
&\mathsf{\mathbf{FIGURE4}} \circledast \neg\mathsf{\mathbf{POS}}_{\triangle} \circledast \neg\mathsf{\mathbf{COLOUR}} \\
&\quad = (\mathsf{\mathbf{POS}}_{\triangle} \circledast \mathsf{\mathbf{RED\_TRIANGLE}} + \mathsf{\mathbf{POS}}_{\square} \circledast \mathsf{\mathbf{BLUE\_SQUARE}}) \circledast \neg\mathsf{\mathbf{POS}}_{\triangle} \circledast \neg\mathsf{\mathbf{COLOUR}} \\
&\quad = (\mathsf{\mathbf{POS}}_{\triangle} \circledast \mathsf{\mathbf{RED\_TRIANGLE}} \circledast \neg\mathsf{\mathbf{POS}}_{\triangle} \circledast \neg\mathsf{\mathbf{COLOUR}}) + \\
&\quad\quad (\mathsf{\mathbf{POS}}_{\square} \circledast \mathsf{\mathbf{BLUE\_SQUARE}} \circledast \neg\mathsf{\mathbf{POS}}_{\triangle} \circledast \neg\mathsf{\mathbf{COLOUR}}) \\
&\quad = (\mathsf{\mathbf{RED\_TRIANGLE}} \circledast \neg\mathsf{\mathbf{COLOUR}}) + \\
&\quad\quad (\mathsf{\mathbf{POS}}_{\square} \circledast \mathsf{\mathbf{BLUE\_SQUARE}} \circledast \neg\mathsf{\mathbf{POS}}_{\triangle} \circledast \neg\mathsf{\mathbf{COLOUR}}) \\
&\quad = (\mathsf{\mathbf{SHAPE}} \circledast \mathsf{\mathbf{TRIANGLE}} \circledast \neg\mathsf{\mathbf{COLOUR}} + \mathsf{\mathbf{RED}} + \mathsf{\mathbf{SIZE}} \circledast \mathsf{\mathbf{SMALL}} \circledast \neg\mathsf{\mathbf{COLOUR}}) + \\
&\quad\quad (\mathsf{\mathbf{POS}}_{\square} \circledast \mathsf{\mathbf{BLUE\_SQUARE}} \circledast \neg\mathsf{\mathbf{POS}}_{\triangle} \circledast \neg\mathsf{\mathbf{COLOUR}}) \\
\\
&\mathit{cleanup}\{\mathsf{\mathbf{FIGURE4}} \circledast \neg\mathsf{\mathbf{POS}}_{\triangle} \circledast \neg\mathsf{\mathbf{COLOUR}}\} \\
&\quad = (\cancel{\mathsf{\mathbf{SHAPE}} \circledast \mathsf{\mathbf{TRIANGLE}} \circledast \neg\mathsf{\mathbf{COLOUR}}} + \mathsf{\mathbf{RED}} + \cancel{\mathsf{\mathbf{SIZE}} \circledast \mathsf{\mathbf{SMALL}} \circledast \neg\mathsf{\mathbf{COLOUR}}}) + \\
&\quad\quad \cancel{(\mathsf{\mathbf{POS}}_{\square} \circledast \mathsf{\mathbf{BLUE\_SQUARE}} \circledast \neg\mathsf{\mathbf{POS}}_{\triangle} \circledast \neg\mathsf{\mathbf{COLOUR}})} \\
&\quad = \mathsf{\mathbf{RED}}
\end{aligned} \tag{2.10}$$

It is possible to query the SPA representation (2.9) without first knowing the position of the object. However, the information extraction process becomes a two-step procedure. First, the given piece of information is used to determine the position of the object. Next, the position of the object and the attribute to be queried is used to extract the desired information (as seen in (2.10)). To extract the position of the object from the representation (2.9), the semantic pointer is bound with the inverse of the given attribute's tag-value pair. For example, to query for the colour of the small object (given information: the object's size is small), the process is as follows:

$$\begin{aligned}
&\mathsf{FIGURE4} \circledast \neg(\mathsf{SIZE} \circledast \mathsf{SMALL})\\
&\quad= (\mathsf{POS}_{\triangle} \circledast \mathsf{RED\_TRIANGLE} + \mathsf{POS}_{\square} \circledast \mathsf{BLUE\_SQUARE}) \circledast \neg(\mathsf{SIZE} \circledast \mathsf{SMALL})\\
&\quad= [\mathsf{POS}_{\triangle} \circledast \mathsf{RED\_TRIANGLE} \circledast \neg(\mathsf{SIZE} \circledast \mathsf{SMALL})] +\\
&\qquad [\mathsf{POS}_{\square} \circledast \mathsf{BLUE\_SQUARE} \circledast \neg(\mathsf{SIZE} \circledast \mathsf{SMALL})]\\
&\quad= [\mathsf{POS}_{\triangle} \circledast (\mathsf{SHAPE} \circledast \mathsf{TRIANGLE} + \mathsf{COLOUR} \circledast \mathsf{RED} + \mathsf{SIZE} \circledast \mathsf{SMALL}) \circledast\\
&\qquad\qquad \neg(\mathsf{SIZE} \circledast \mathsf{SMALL})] +\\
&\qquad [\mathsf{POS}_{\square} \circledast \mathsf{BLUE\_SQUARE} \circledast \neg(\mathsf{SIZE} \circledast \mathsf{SMALL})] \qquad (2.11)\\
&\quad= \{[(\mathsf{POS}_{\triangle} \circledast \mathsf{SHAPE} \circledast \mathsf{TRIANGLE}) + (\mathsf{POS}_{\triangle} \circledast \mathsf{COLOUR} \circledast \mathsf{RED}) +\\
&\qquad\qquad (\mathsf{POS}_{\triangle} \circledast \mathsf{SIZE} \circledast \mathsf{SMALL})] \circledast \neg(\mathsf{SIZE} \circledast \mathsf{SMALL})\} +\\
&\qquad [\mathsf{POS}_{\square} \circledast \mathsf{BLUE\_SQUARE} \circledast \neg(\mathsf{SIZE} \circledast \mathsf{SMALL})]\\
&\quad= [\cdots + (\mathsf{POS}_{\triangle} \circledast \mathsf{SIZE} \circledast \mathsf{SMALL} \circledast \neg\mathsf{SIZE} \circledast \neg\mathsf{SMALL}) + \cdots] +\\
&\qquad [\cdots + (\mathsf{POS}_{\square} \circledast \mathsf{SIZE} \circledast \mathsf{BIG} \circledast \neg\mathsf{SIZE} \circledast \neg\mathsf{SMALL}) + \cdots]\\
&\quad= \cdots + (\mathsf{POS}_{\triangle}) + \cdots + (\mathsf{POS}_{\square} \circledast \mathsf{SIZE} \circledast \mathsf{BIG} \circledast \neg\mathsf{SIZE} \circledast \neg\mathsf{SMALL}) + \cdots
\end{aligned}$$

Since the desired information is the position of the object, performing the cleanup operation on this result with a vocabulary consisting of all possible values of the position attribute (i.e., $\{\mathsf{POS}_{\triangle}, \mathsf{POS}_{\square}\}$) yields:

$$\begin{aligned}
&\mathit{cleanup}\{\mathsf{FIGURE4} \circledast \neg(\mathsf{SIZE} \circledast \mathsf{SMALL})\}\\
&\quad= \cancel{\cdots} + (\mathsf{POS}_{\triangle}) + \cancel{\cdots} + \cancel{(\mathsf{POS}_{\square} \circledast \mathsf{SIZE} \circledast \mathsf{BIG} \circledast \neg\mathsf{SIZE} \circledast \neg\mathsf{SMALL})} + \cancel{\cdots}\\
&\quad= \mathsf{POS}_{\triangle}
\end{aligned}$$

Next, with the known position $\mathsf{POS}_{\triangle}$, and the query attribute COLOUR, the information extraction process proceeds as in (2.10).

It is important to note that while the information extraction process described above (from Eq. (2.11) onwards) may seem complex, it can be accurately summarized as two iterative applications of the binding operation followed by the cleanup operation on the result of the binding operation:

1. $\mathsf{FIGURE4} \circledast \neg(\mathsf{SIZE} \circledast \mathsf{SMALL}) = \mathsf{RESULT}_{\mathsf{POS}}$
2. $\mathit{cleanup}\{\mathsf{RESULT}_{\mathsf{POS}}\} = \mathsf{POS}_{\triangle}$
3. $\mathsf{FIGURE4} \circledast \neg\mathsf{POS}_{\triangle} \circledast \neg\mathsf{COLOUR} = \mathsf{RESULT}_{\mathsf{COLOUR}}$
4. $\mathit{cleanup}\{\mathsf{RESULT}_{\mathsf{COLOUR}}\} = \mathsf{RED}$

In addition, the sequence of SPA operations needed for extracting information from a semantic pointer is solely determined by the schema (i.e., a specific combination and order of semantic pointer operators) used to encode the information as a semantic pointer. Importantly, the SPA does not place limitations on how information is represented as semantic pointer, nor does it hypothesize the specific encoding schemata used in the biological brain; and, it should be emphasized that the encoding used in the example above is merely one of many possible ways the symbolic representation of the SPA can be used to represent information.

2.2.7 Discussion

This section discussed the use of the SPA to perform symbol-like computation, demonstrating how information can be represented and manipulated in this form. In this section, three additional topics will be discussed: the idea of compression, decompression and pointers in the SPA; the issue regarding the commutativity of the SPA operators; and the caveats regarding the semantic pointer inverse.

2.2.7.1 Compression, Decompression, and Pointers

A recurring concept in the SPA is the idea of compression and decompression of information, and the use of this mechanic to create "pointers" to data. In the symbolic interpretation of the SPA, both the binding and collection operators can be considered compression operators as the results of using these operations can be represented by another semantic pointer. For example, in the following formulation:

$$\mathbf{A} = \mathbf{B} + \mathbf{C}$$

the semantic pointer "$\mathbf{A}$" (a singular semantic pointer) is considered a compressed representation as it contains the information "$\mathbf{B} + \mathbf{C}$" (the result of the collection operator applied to two semantic pointers). Conversely, semantic pointers can be decompressed using the unbinding and un-collecting operators. The compressive property of semantic pointers is important in the SPA because it allows for scalability. Having unbounded (uncompressed) representations would be detrimental to the overall effort of realizing the SPA as a physical cognitive system, seeing as most physical system have constraints on their representational capacity.

In the SPA, because these compressed representations can be decompressed to extract additional information stored within the semantic pointers, they act much in the same way pointers do in computer systems. In computer systems, a pointer is abstract value, on which an operation (dereferencing – traversing to the location in computer memory specified by the pointer's value) can be applied to extract more information. In the SPA, a "pointer" is conceptual representation that has a semantic context (hence the name "semantic pointer"), on which decompression operations can be applied to extract additional information. Importantly, as with pointers in computer systems, mathematical operations can be performed on semantic pointers without the need to first dereference the pointer values. The analogy between semantic pointers and pointers in computer systems is further discussed in Section 2.3.9.

2.2.7.2 Operator Commutativity

Both the binding and collecting operators discussed in the previous section are commutative, producing identical results independent of the order of their operands. While this property is generally useful in physical implementations of the SPA (because no extra complexity is needed to keep track of the order of the operators), there are instances where enforcing the order of the operands is useful. Consider the scenario presented in Equation (2.8). In that scenario, the commutativity of the collection operator meant that images containing different combinations of $\{\mathbf{RED}, \mathbf{BLUE}, \mathbf{SQUARE}, \mathbf{TRIANGLE}, \mathbf{BIG}, \mathbf{SMALL}\}$ would be encoded as the same semantic pointer. In the standard SPA, this issue was circumvented by adding an additional "position" tag to the semantic pointers. However, this issue can also be addressed by using non-commutative operators.

For this discussion, the following assumptions are made:

- The SPA operators can be made non-commutative by marking its operands with an order. This is denoted by a numerical subscript and angle brackets. For example,

$$\langle \mathbf{X} \rangle_1 + \langle \mathbf{Y} \rangle_2 \neq \langle \mathbf{Y} \rangle_1 + \langle \mathbf{X} \rangle_2$$

- Performing a distributive operation with marked operands only affects marked operands of the same number. For example,

$$(\langle \mathbf{X} \rangle_1 + \langle \mathbf{Y} \rangle_2) \circledast \langle \mathbf{Z} \rangle_2 = \langle \mathbf{X} \rangle_1 + \langle \mathbf{Y} \circledast \mathbf{Z} \rangle_2$$

These assumptions, when applied to the scenario presented with Equation (2.8) results in the ability to extract information in a more straightforward process. To understand how this can be done, Equation (2.9) is first redefined as follows, with the "$\mathbf{POS}_{\triangle}$" and "$\mathbf{POS}_{\square}$" objects marked with the operand number 1 and 2 respectively:

$$\begin{aligned}\mathbf{FIGURE4} &= \langle \mathbf{RED\_TRIANGLE} \rangle_1 + \langle \mathbf{BLUE\_SQUARE} \rangle_2 \\ &= \langle \mathbf{SHAPE} \circledast \mathbf{TRIANGLE} + \mathbf{COLOUR} \circledast \mathbf{RED} + \mathbf{SIZE} \circledast \mathbf{SMALL} \rangle_1 + \\ &\quad \langle \mathbf{SHAPE} \circledast \mathbf{SQUARE} + \mathbf{COLOUR} \circledast \mathbf{BLUE} + \mathbf{SIZE} \circledast \mathbf{BIG} \rangle_2\end{aligned}$$

As before, this example will query the semantic pointer for the colour of the leftmost object. In order to do so, the query semantic pointer (**COLOUR**) is first inverted, then marked with the appropriate number (1), and finally, bound with the **FIGURE4** semantic pointer.

$$\begin{aligned}
&\mathbf{FIGURE4} \circledast \langle\neg\mathbf{COLOUR}\rangle_1 \\
&\quad = (\langle\mathbf{RED\_TRIANGLE}\rangle_1 + \langle\mathbf{BLUE\_SQUARE}\rangle_2) \circledast \langle\neg\mathbf{COLOUR}\rangle_1 \\
&\quad = (\langle\mathbf{RED\_TRIANGLE} \circledast \neg\mathbf{COLOUR}\rangle_1 + \langle\mathbf{BLUE\_SQUARE}\rangle_2) \\
&\quad = \ \langle\mathbf{SHAPE} \circledast \mathbf{TRIANGLE} \circledast \neg\mathbf{COLOUR} + \mathbf{RED} + \mathbf{SIZE} \circledast \mathbf{SMALL} \circledast \neg\mathbf{COLOUR}\rangle_1 + \\
&\qquad \langle\mathbf{BLUE\_SQUARE}\rangle_2
\end{aligned}$$

$$\begin{aligned}
&\mathit{cleanup}\{\mathbf{FIGURE4} \circledast \langle\neg\mathbf{COLOUR}\rangle_1\} \\
&\quad = \ \langle\cancel{\mathbf{SHAPE} \circledast \mathbf{TRIANGLE} \circledast \neg\mathbf{COLOUR}} + \mathbf{RED} + \cancel{\mathbf{SIZE} \circledast \mathbf{SMALL} \circledast \neg\mathbf{COLOUR}}\rangle_1 + \\
&\qquad \cancel{\langle\mathbf{BLUE\_SQUARE}\rangle_2} \\
&\quad = \mathbf{RED}
\end{aligned}$$

Unfortunately, this way of enforcing non-commutativity does not support the 2-step query process described by Equation (2.11), which illustrates the limitations of this approach.

2.2.7.3 Semantic Pointer Inverses

When introducing the semantic pointer inverse, Section 2.2.5 only discussed its application to one semantic pointer. Unfortunately, when used with the collection operator, it behaves similar to the algebraic reciprocal in terms of distributivity. In essence:

$$\neg(\mathbf{A} + \mathbf{B}) \neq (\neg\mathbf{A} + \neg\mathbf{B}) \tag{2.12}$$

Because of this property, some SPA expressions cannot be fully expanded, for example ($\mathbf{A} \circledast \neg(\mathbf{B} + \mathbf{C})$). In the SPA, it is common to use a distributive form of the inverse operator – called the "approximate inverse", and denoted by the ($\sim$) symbol – to expand SPA expressions to further computation. While convenient, the use of the approximate inverse does introduce additional terms (noise) into the SPA computation. As an example, consider the expression $((\mathbf{A} + \mathbf{B}) \circledast \neg(\mathbf{A} + \mathbf{B}))$. Using the exact semantic pointer inverse, the expression evaluates to exactly $\mathbf{I}$. However, using the approximate inverse results in:

$$\begin{aligned}(\mathbf{A}+\mathbf{B}) \circledast \sim(\mathbf{A}+\mathbf{B}) &= (\mathbf{A}+\mathbf{B}) \circledast (\sim\mathbf{A} + \sim\mathbf{B}) \\ &= (\mathbf{A} \circledast \sim\mathbf{A}) + (\mathbf{A} \circledast \sim\mathbf{B}) + (\mathbf{B} \circledast \sim\mathbf{A}) + (\mathbf{B} \circledast \sim\mathbf{B}) \\ &= \mathbf{I} + (\mathbf{A} \circledast \sim\mathbf{B}) + (\mathbf{B} \circledast \sim\mathbf{A}) + \mathbf{I} \\ &= 2 \times \mathbf{I} + (\mathbf{A} \circledast \sim\mathbf{B}) + (\mathbf{B} \circledast \sim\mathbf{A})\end{aligned}$$

As the equations show, two additional terms have been introduced as a result of using the approximate inverse. The issue is compounded if the SPA computation consists of multiple steps, and may necessitate introducing *cleanup* operations between each step to remove the extraneous terms. The approximate inverse however, is not without its uses (e.g., for numerical stability), as will be discussed in Section 2.3.4.3.

2.3 The Semantic Pointer Architecture: Cognitive Computation with Vectors

The previous section discussed the symbol-like nature of the SPA, and while it is useful in conceptualizing how information can be encoded, manipulated, and extracted using semantic pointers, the symbol-like characterization of the SPA does not provide any details on how the computation is performed numerically. While it is possible to specify a system's cognitive computation purely using the symbol-like SPA representation, a numerically grounded version of the SPA provides the first step in the goal of implementing the SPA in a neural architecture.

The SPA itself does not impose a constraint on the type of numerical system to be used, but does impose the following criteria:

- The numerical representation of the semantic pointers must be vector or matrix based.
- For semantic pointers that interact, the dimensionality of the vector or matrix representation of the semantic pointers must be of equal size.
- Between the binding and collection operators, one operator must generate a semantic pointer that is semantically similar to the input semantic pointers, and one operator must generate a semantic pointer that is semantically different to the input semantic pointers. As a convention, in Spaun, the collection operator has the former property, and the binding operator has the latter property.

As long as they keep with these criteria, any numerical system can be used – assuming they can be adapted to support the representational structure, semantic relationships, and symbolic operations presumed present in the SPA. A numerical system adapted from the Plate's Holographic

Reduced Representation (HRR) [Plate, 1994] was chosen for the Spaun model, and in this section the HRR-based implementation of the SPA will be explored. Henceforth, the term "HRR-SPA" will be used to reference the HRR-based implementation of the "cognitive" symbol-like form of the SPA discussed in Section 2.2.

2.3.1 SPA Concepts, Expressions, and Similarity

As HRRs, concepts are represented by high-dimensional vectors (Spaun uses 512 dimensional vectors), where each element of the vector is chosen from a normal distribution with a mean of 0 and a variance of $1/d$, where d is the dimensionality of the vector. Spaun borrows this definition and generates a majority of its semantic pointers[1] in the same fashion.

The reasons for generating the vectors in this fashion are two-fold. First, generating the vector from a normal distribution results in an expected vector magnitude of 1, and this will be of importance in the definition of a similarity measure (see below) and in the neural implementation of the SPA (see Section 2.5.5.1). Second, computing the Fourier transform on these vectors generates Fourier coefficients that are uniformly distributed around 0 and an identical variance across all frequency components (see Figure 2.3). The importance of the Fourier transform will be discussed in Section 2.3.2. Additionally, the common variance across all frequency components will prove advantageous in the neural implementation (see Section 2.5.5.3).

In Spaun, the discrete Fourier transform (DFT) is used to compute the Fourier transform of the semantic pointer vectors. Performing the DFT on a semantic pointer results in a complex-valued vector (i.e., each element in the vector is a complex number) of the same dimensionality as the input vector. For a given semantic pointer vector $\mathbf{X} = [x_0, x_1, \ldots, x_{d-1}]$ of dimensionality d, the DFT ($\mathcal{F}$) and the inverse DFT ($\mathcal{F}^{-1}$) is defined as:

$$\text{For } \mathbf{Y} = \mathcal{F}(\mathbf{X}), \qquad y_j = \sum_{k=0}^{d-1} x_k e^{-\frac{(2\pi i)}{d} jk} \tag{2.13}$$

$$\text{For } \mathbf{X} = \mathcal{F}^{-1}(\mathbf{Y}), \qquad x_j = \frac{1}{d} \sum_{k=0}^{d-1} y_k e^{\frac{(2\pi i)}{d} jk} \tag{2.14}$$

$$\text{where } i = \sqrt{-1}$$

Figure 2.2 illustrates the effect of performing the DFT on a 10-dimensional semantic pointer, with each complex value of the Fourier vector computed, and also represented as two-dimensional vectors on the real-complex plane. Figure 2.3 compares the difference between the Fourier coefficients

[1] In the HRR-based implementation of the SPA, each semantic pointer is a vector, and as such, the term "semantic pointer" and "vector" will be used interchangeably.

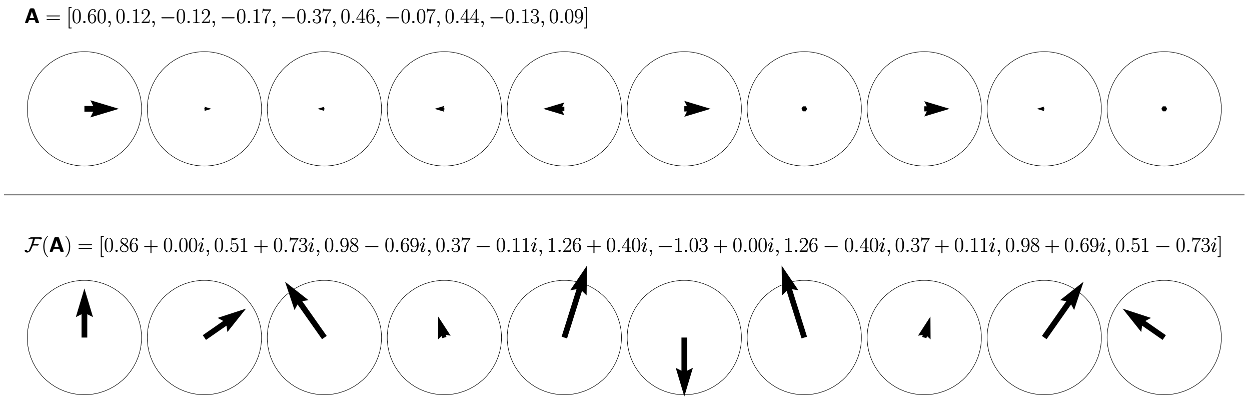

A = [0.60, 0.12, −0.12, −0.17, −0.37, 0.46, −0.07, 0.44, −0.13, 0.09]
𝓕(A) = [0.86 + 0.00i, 0.51 + 0.73i, 0.98 − 0.69i, 0.37 − 0.11i, 1.26 + 0.40i, −1.03 + 0.00i, 1.26 − 0.40i, 0.37 + 0.11i, 0.98 + 0.69i, 0.51 − 0.73i]

Figure 2.2: Depiction of a randomly generated 10-dimensional semantic pointer ($\mathbf{A}$) in vector space (top), the complex-valued vector ($\mathcal{F}(\mathbf{A})$) in Fourier space (bottom). Displayed are the numerical vector values as well as an illustration of each vector component plotted as a 2-dimensional vector in the real-complex plane.

of an HRR-SPA vector and a vector with elements chosen from a uniform distribution from the range $[0, 1)$.

It is important to note that with the HRR-SPA, the semantic pointers remain at a fixed dimensionality regardless of the number of operations performed (this is not necessarily true for SPA expressions encoded with other operators). In essence, what this implies is that all SPA expressions generated using any combination of the SPA operators discussed in the section previous have the same dimensionality. This fact is important because it allows for the neural implementation to be constructed independent of the number of operations performed within the system, and this makes the neural implementation more flexible as it will inherently support an infinite number of operations (the size of the neural network does not limit the number of operations), and it also does not require the network to be partitioned based off the number of operations to be performed.

With regards to the “semantics” of a semantic pointer, in the HRR-SPA, semantic pointers that have similar meanings are themselves similar when compared to each other. To compute the similarity of two semantic pointers, the vector dot product ($\bullet$) is used. For two given semantic pointers $\mathbf{X}$ and $\mathbf{Y}$, the similarity measure ($\mathbf{s}$) between the two vectors is defined as:

$$\mathbf{s} = \mathbf{X} \bullet \mathbf{Y} = \sum_{k=0}^{d-1} x_k y_k \tag{2.15}$$

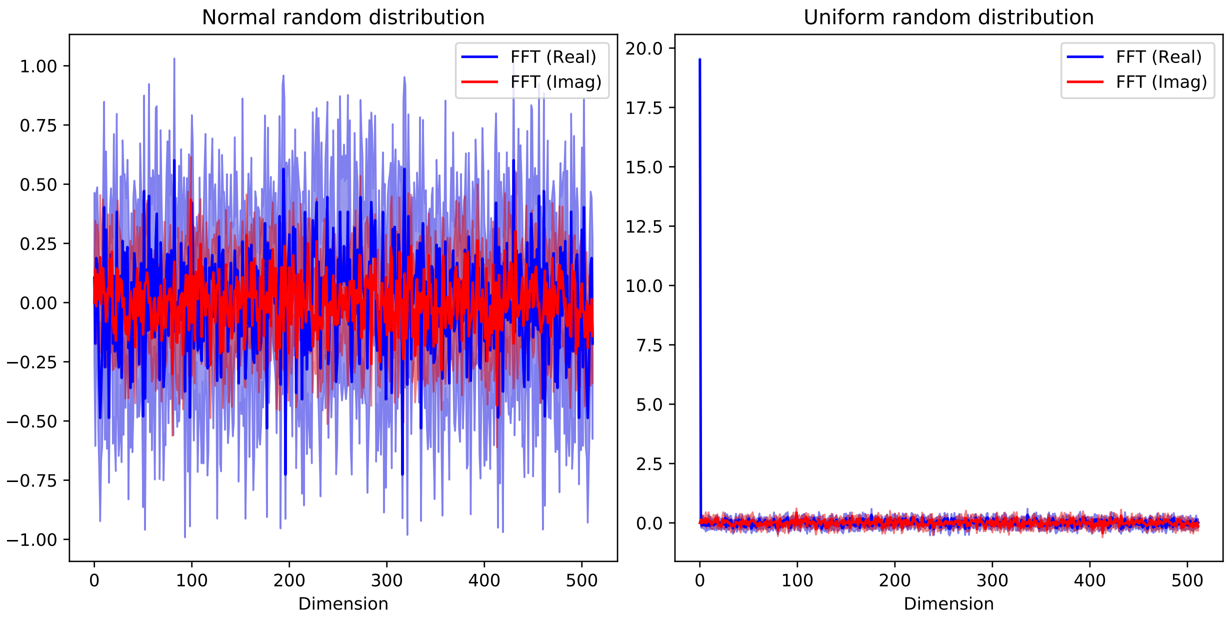


Figure 2.3: Plots showing the expected values of the Fourier coefficients produced as a result of performing the discrete Fourier transform on an HRR-SPA vector (left), and a normalized vector generated with elements chosen from a uniform distribution with a value range of $[0, 1)$ (right). For both plots, data was collected for a thousand 512-dimensional vectors, with the 95% confidence interval plotted for each of the Fourier coefficients. The values for the real-valued coefficients are shown in blue, while the values of the imaginary-valued coefficients are shown in red. From the plots, it can be seen that the HRR-SPA vectors produce Fourier coefficients that are uniformly distributed, and constrained within the range of about -1 to 1. In contrast, the uniformly-distributed vectors produce Fourier coefficients that are mostly uniformly distributed, with the exception of the first coefficient.

It should be noted that Equation (2.15) assumes that the vectors are of unit length (i.e., their magnitudes[2] equal 1), which is in keeping with the definition of how the semantic pointers are randomly generated. While it is possible to normalize the similarity equation by the vector magnitudes (i.e., computing the cosine angle between the two vectors), the division operation is relatively expensive to implement in a neural architecture (see Section 4.4) and thus in most cases where the similarity measure is required, the un-normalized version is used.

[2]The magnitude ($\|\mathbf{x}\|$) of the vector $\mathbf{x}$ is computed using the L2 norm. i.e., $\|\mathbf{x}\| = \left(\sum_{k=0}^{d-1} x_k^2\right)^{0.5}$

2.3.2 SPA Binding

The binding operation used in the Spaun implementation of the SPA is based on the HRR binding operation. With HRRs, two vectors are bound by computing the circular convolution between the two vectors. The circular convolution operation is similar to the convolution operation, with the primary difference being that the vector operands are circular (i.e., they "wrap" back on themselves). Appendix A.1 illustrates the difference between the standard convolution and the circular convolution operations. Formally, the circular convolution operation (hereafter referred to using the SPA binding symbol "$\circledast$") is defined as:

$$\begin{aligned}&\text{If } \mathbf{X} = [x_0, x_1, \ldots, x_{d-1}] \text{ and } \mathbf{Y} = [y_0, y_1, \ldots, y_{d-1}],\\ &\text{For } \mathbf{R} = \mathbf{X} \circledast \mathbf{Y},\\ &r_j = \sum_{k=0}^{d-1} x_k y_{j-k}, \qquad \text{for } j = 0 \text{ to } d-1,\\ &\text{(where the subscripts for } y \text{ are modulo-}d\text{, and } d \text{ is the dimensionality of the vectors)}\end{aligned} \tag{2.16}$$

This formal form of the circular convolution operation is rather computationally expensive, requiring d^2 multiplication operations to compute. Conveniently, the number of multiplication operations can be reduced by projecting the vectors into the Fourier domain to do the computation.

$$\begin{aligned}&\text{For } \mathbf{R} = \mathbf{X} \circledast \mathbf{Y},\\ &\mathbf{R} = \mathcal{F}^{-1}(\mathcal{F}(\mathbf{X}) \odot \mathcal{F}(\mathbf{Y})),\\ &\text{(where } \odot \text{ is the element-wise multiplication operation)}\end{aligned} \tag{2.17}$$

In the Fourier domain, the circular convolution operation amounts to the element-wise multiplication between each of the complex-valued vector coefficients, which requires a maximum of $4d$ multiplications (d multiplications for each combination of real and imaginary components) to compute.

2.3.3 SPA Collections

In HRR-SPA, the collections operation is performed by computing the superposition (element-wise addition) of the input vectors. In essence:

$$\begin{aligned}
&\text{If } \mathbf{X} = [x_0, x_1, \ldots, x_{d-1}] \text{ and } \mathbf{Y} = [y_0, y_1, \ldots, y_{d-1}],\\
&\text{For } \mathbf{R} = \mathbf{X} + \mathbf{Y},\\
&r_k = x_k + y_k, \qquad \text{for } k = 0 \text{ to } d-1
\end{aligned} \tag{2.18}$$

2.3.4 Special Semantic Pointers

Section 2.2.4 discussed 2 special semantic pointers – the identity and null semantic pointers – and Section 2.2.5 discussed the method for computing the inverse of a semantic pointer. This section will discuss how these semantic pointers are represented and calculated in the HRR-SPA.

2.3.4.1 The Identity Semantic Pointer

Equation (2.1) defines the identity semantic pointer such that binding any semantic pointer with the identity semantic pointer results in no change to original semantic pointer. In HRR-SPA, there is only one vector that satisfies this criterion, and this vector can be calculated in one of two ways. The most direct method is to consider the definition of the HRR-SPA binding operation.[3] From Equation (2.17), the binding operation with the identity semantic pointer is defined as:

$$\mathbf{X} \circledast \mathbf{I} = \mathcal{F}^{-1}(\mathcal{F}(\mathbf{X}) \odot \mathcal{F}(\mathbf{I}))$$

For the result of the element-wise multiplication of the Fourier coefficients to remain unchanged, $\mathcal{F}(\mathbf{I})$ must be a vector of ones ($[1, 1, \ldots, 1]$). Thus,

$$\begin{aligned}
\mathcal{F}(\mathbf{I}) &= [1, 1, \ldots, 1]\\
\mathcal{F}^{-1}(\mathcal{F}(\mathbf{I})) &= \mathcal{F}^{-1}([1, 1, \ldots, 1])\\
\mathbf{I} &= [1, 0, 0, \ldots, 0]
\end{aligned}$$

2.3.4.2 The Null Semantic Pointer

Equation (2.2) defines the null semantic pointer such that binding it with any semantic pointer results in the null semantic pointer. The null semantic pointer can be determined using the method

[3]The other method is to recognize that performing a convolution with an impulse vector ($[1, 0, 0, \ldots, 0]$) results in no change to the original vector.

used to compute the identity semantic pointer. With the null semantic pointer, the observation can be made that performing the element-wise multiplication of the Fourier coefficients with a vector of zeros ($[0, 0, \ldots, 0]$) provides the desired properties of the null semantic pointer. Thus,

$$\begin{aligned}\mathcal{F}(\boldsymbol{\emptyset}) &= [0, 0, \ldots, 0] \\ \mathcal{F}^{-1}(\mathcal{F}(\boldsymbol{\emptyset})) &= \mathcal{F}^{-1}([0, 0, \ldots, 0]) \\ \boldsymbol{\emptyset} &= [0, 0, 0, \ldots, 0]\end{aligned}$$

Coincidentally, the HRR-SPA null semantic pointer also meets the requirements of Equation (2.3) as $\mathbf{X} + [0, 0, \ldots, 0] = \mathbf{X}$.

2.3.4.3 The Inverse of a Semantic Pointer

Equation (2.19) defines the inverse of a semantic pointer such that binding a semantic pointer with its inverse results in the identity semantic pointer. From this, and Equation (2.17), the inverse of a semantic pointer can be computed:

$$\begin{aligned}\mathbf{X} \circledast \neg\mathbf{X} = \mathcal{F}^{-1}(\mathcal{F}(\mathbf{X}) \odot \mathcal{F}(\neg\mathbf{X})) &= \mathbf{I} \\ \mathcal{F}(\mathbf{X}) \odot \mathcal{F}(\neg\mathbf{X}) &= \mathcal{F}(\mathbf{I}) \\ \mathcal{F}(\neg\mathbf{X}) &= \mathcal{F}(\mathbf{I}) \oslash \mathcal{F}(\mathbf{X}) \\ \neg\mathbf{X} &= \mathcal{F}^{-1}(\mathcal{F}(\mathbf{I}) \oslash \mathcal{F}(\mathbf{X})), \\ &\text{(where } \oslash \text{ is the element-wise division operation)}\end{aligned} \tag{2.19}$$

As Equation (2.19) illustrates, the Fourier coefficients of the inverse semantic pointer is computed by taking the complex reciprocal[4] of the Fourier coefficients of the original semantic pointer.

However, because the Fourier coefficients of a semantic pointer have a mean of 0, it is possible to generate semantic pointers with very small valued Fourier coefficients, and since the complex reciprocal involves a division, taking the reciprocal of small Fourier coefficients results in large Fourier coefficients (potentially infinite) in the inverse semantic pointer. This is an undesirable property, both mathematically, and for the neural implementation of the inverse operation.

Fortunately, the HRR defines an inverse operation (henceforth referred to as the "approximate inverse", and given the symbol "$\sim$"). In the HRR, the approximate inverse of a vector is computed as in Equation (2.19), with the exception that instead of using the complex reciprocal, the complex

[4]For a complex number $(a + bi)$, the complex reciprocal is $(\frac{a}{a^2+b^2} - \frac{b}{a^2+b^2}i)$.

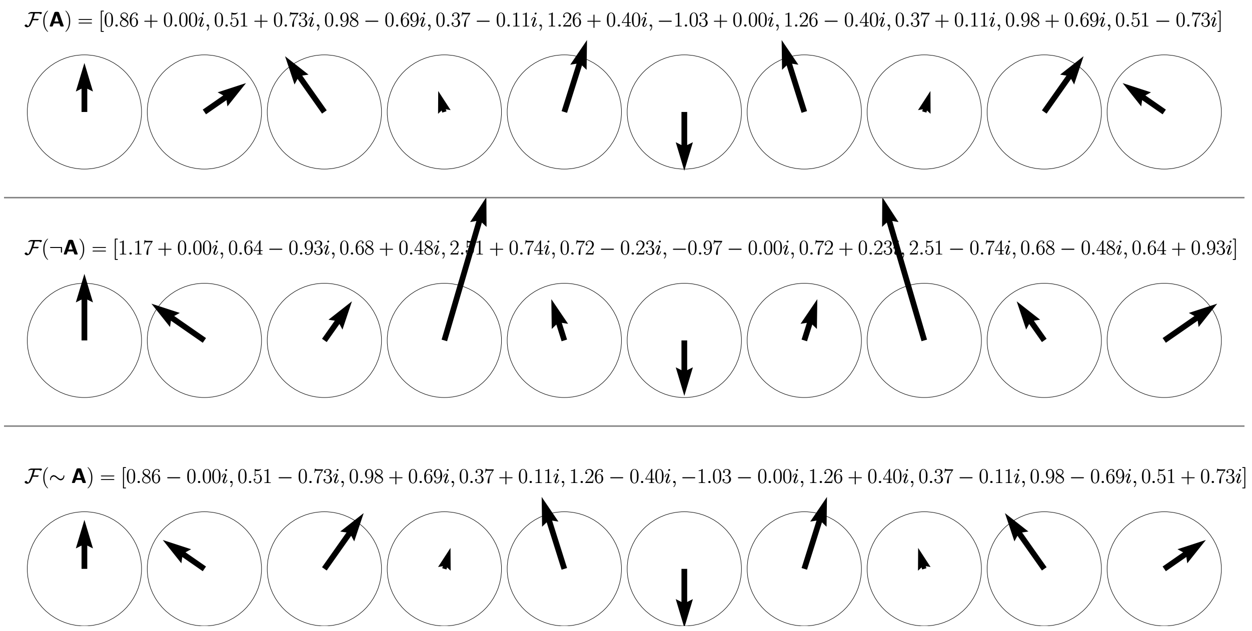


Figure 2.4: Illustration of the Fourier coefficients of one randomly generated semantic pointer ($\mathbf{A}$) (top), its exact inverse ($\neg\mathbf{A}$) (middle), and its approximate inverse ($\sim\mathbf{A}$) (bottom). The Fourier coefficients are represented as 2-dimensional vectors in the real-complex plane. See the text for details on how these Fourier coefficients are computed. Comparing the Fourier coefficients of the exact inverse and the approximate inverse reveals that the coefficients of the approximate inverse maintains the same angle as the coefficients of the exact inverse, while keeping the same magnitudes as the coefficients of the DFT.

conjugate is used.[5]

$$\begin{aligned} \mathcal{F}(\sim\mathbf{X}) &= \overline{\mathcal{F}(\mathbf{X})} \\ \sim\mathbf{X} &= \mathcal{F}^{-1}\left(\overline{\mathcal{F}(\mathbf{X})}\right), \end{aligned} \tag{2.20}$$

(where $\overline{z}$ is the complex conjugate of the complex number z)

Figure 2.4 illustrates the difference between the Fourier coefficients of a randomly generated semantic pointer, the inverse of the semantic pointer, and the approximate inverse of the semantic pointer.

[5]For a complex number $(a + bi)$, the complex conjugate is $(a - bi)$. In essence, the complex conjugate is the complex reciprocal without compensating for the magnitude of the complex number.

Interestingly, the performing the approximate inverse on a vector is equivalent to reversing the order of elements 1 to $d-1$ of the original vector (see Appendix A.2). I.e.

$$\begin{aligned} &\text{For } \mathbf{Y} = \sim\!\mathbf{X}, \\ y_k &= x_{-k}, \qquad \text{for } j = 0 \text{ to } d-1, \text{ modulo } d, \text{ or} \\ [y_0, y_1, y_2, \ldots, y_{d-1}] &= [x_0, x_{d-1}, x_{d-2}, \ldots, x_1] \end{aligned} \tag{2.21}$$

It should be noted that the use of the term "approximate inverse" and the symbol ($\sim$) here is deliberate. Since the process of reversing the order of the vector elements is a linear operation, it is distributive, and thus matches the definition of the "approximate inverse" described in Section 2.2.7. In addition, it incurs all of the disadvantages mentioned in that section.

2.3.5 Unitary Semantic Pointers

Section 2.2.2 introduced the concept of bound powers, and this is used extensively in Spaun to generate positional and numerical concept semantic pointers (see Section 3.2.5 and 3.2.6). Because the HRR-SPA binding operation involves the multiplication of Fourier coefficients, computing bound powers results in an explosive growth of the vector magnitude (see Figure 2.5).

Fortunately, the HRR defines a special class of vectors that do not suffer from this problem. Plate refers to this class of vectors as "unitary vectors", and they have the crucial property that all of their Fourier coefficients always have a magnitude[6] of 1 (i.e., they are of unit length). In HRR-SPA, unitary semantic pointers are generated using a 3-step process:

1. Generate a random semantic pointer using the method defined in Section 2.3.1.
2. Compute the DFT on the semantic pointer and normalize each Fourier coefficient of the DFT such that their magnitudes are equal to 1. I.e.

$$\begin{aligned} &\text{If } \mathbf{Y} = \mathcal{F}(\mathbf{X}), \text{ and } \mathbf{Z} = \mathcal{F}\left(Unitary(\mathbf{X})\right), \text{ then} \\ &Z_k = \frac{Y_k}{\|Y_k\|} \end{aligned}$$

3. Compute the inverse DFT on the normalized Fourier coefficients to transform the semantic pointer back from the Fourier domain.

The unitary semantic pointer itself has several useful properties:

1. A unitary semantic pointer is always of unit length (its vector magnitude is always 1)

[6]For the complex number $z = a + bi$, its magnitude $\|z\| = \sqrt{a^2 + b^2}$.

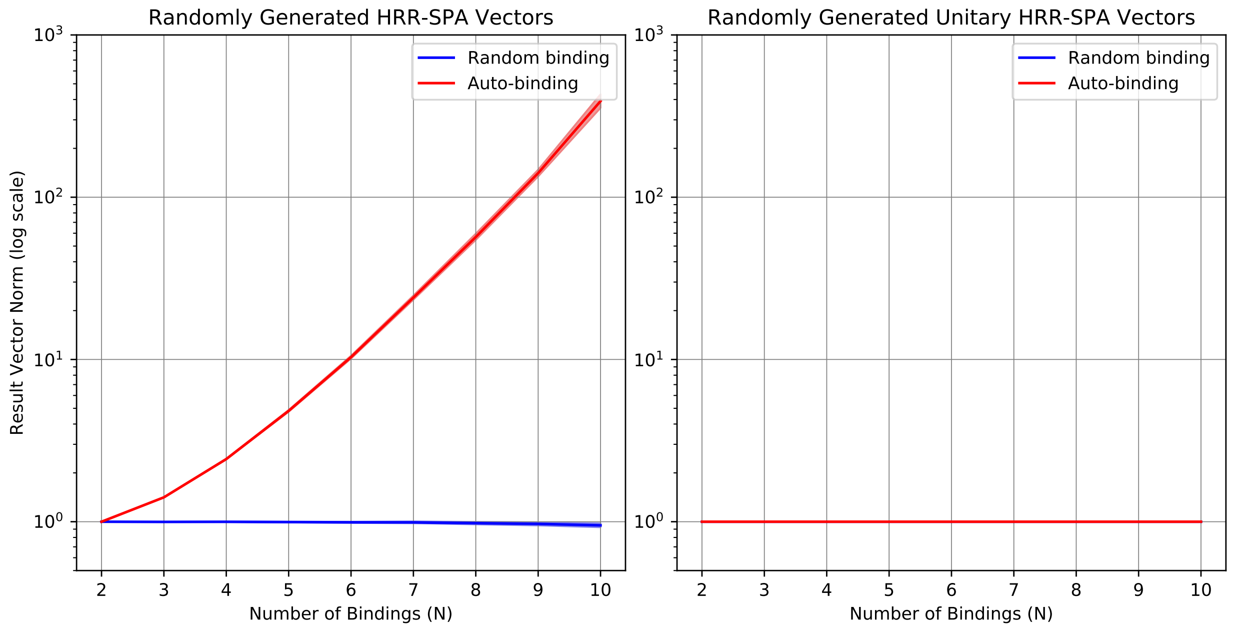


Figure 2.5: Illustration of the expected semantic pointer magnitudes for multiple applications of the binding operation. Each plot illustrates both the expected magnitudes from binding n randomly generated 512-dimensional semantic pointers (blue) and from calculating the bound n^{th} power of a single randomly generated 512-dimensional semantic pointer (red). Displayed are data from 1000 iterations with 95% confidence interval bars included. (Left) Expected semantic pointer magnitudes using randomly generated semantic pointers. (Right) Expected semantic pointer magnitudes using randomly generated *unitary* semantic pointers.

2. Since the complex reciprocal and complex conjugate of a unit length complex number are equal, it follows that the inverse and approximate inverse of a unitary semantic pointer are also equal.
3. The result of binding a unitary semantic pointer with another unitary semantic pointer is always a unitary semantic pointer.

2.3.6 Semantic Pointer Normalization

One of the primary assumptions of the HRR-SPA implementation is that the semantic pointers have an expected vector magnitude of approximately 1. While this assumption is not crucial for the purely mathematical representation of the SPA, it synergizes well with the neural implementation of the SPA (see Section 2.5.5.1). In the HRR-SPA, the semantic pointers can be normalized by dividing the vector value with its magnitude, i.e.,

$$\hat{\mathbf{A}} = \frac{\mathbf{A}}{\|\mathbf{A}\|}, \tag{2.22}$$

where $\hat{\mathbf{A}}$ is the normalized form of the semantic pointer $\mathbf{A}$. It should be noted that performing the vector normalization operation does result in the loss of information, and for a collection of semantic pointers, the expected vector magnitude of each component semantic pointer (after vector normalization) is $\frac{1}{\sqrt{n}}$, where n is the number of semantic pointers in the collection. As an example, given the collection $\mathbf{A} = \mathbf{B} + \mathbf{C} + \mathbf{D}$, the vector normalized semantic pointer $\hat{\mathbf{A}} \approx \frac{1}{\sqrt{3}}\mathbf{B} + \frac{1}{\sqrt{3}}\mathbf{C} + \frac{1}{\sqrt{3}}\mathbf{D}$.

2.3.7 Semantic Pointer Cleanup

Section 2.2.6 introduced the idea of the "cleanup" operation whereby a semantic pointer is compared to a vocabulary of semantic pointers to constrain the possible values it can take. The HRR-SPA cleanup operation proceeds using the following steps:

1. Calculate the similarity measure of $\mathbf{A}$ against all of the semantic pointers in the vocabulary.
2. From the calculated similarity measures, identify the largest one.
3. Identify the semantic pointer (from the vocabulary) that is paired with the largest similarity measure. This is the result of the cleanup operation.

It should be noted that the equivalence between the dot product operation $\mathbf{X} \bullet \mathbf{Y}$ and the matrix operation $\vec{X}\vec{Y}^T$ – where $\vec{X}$ and $\vec{Y}$ are the vector equivalents of the semantic pointers $\mathbf{X}$ and $\mathbf{Y}$, respectively, and the superscript T represents the transpose of the vector – can be used to reduce the computational complexity of the first step of the cleanup operation. This is further explored in Section 2.5.5.5.

2.3.8 SPA Operator Non-commutativity

Section 2.2.7 introduced the concept of enforcing operand order on the SPA operators. With respect to the HRR, Plate [2003] discusses three methods of implementing non-commutative variants of the circular convolution operator:

1. By applying a different vector-space permutation to each of the operands.
2. By applying a different Fourier-space permutation to each of the operands.
3. And by replacing the circular convolution operator by a randomized outer product operation.

Since the latter two methods of the list above is only applicable to the circular convolution operator, the HRR-SPA uses the first method to create non-commutative variants of the SPA operators. As in Section 2.2.7, the scenario presented by Equation (2.8) is used to illustrate the use of these random permutations in creating non-commutative SPA operators.

In Section 2.2.7, operator non-commutativity was achieved by "marking" each operand with different numerical indices. The HRR-SPA follows a similar procedure, but does so using permutation matrices to perform the operand marking. In contrast with the SPA operand marking however, the HRR-SPA requires both a permutation and inverse permutation matrix to perform the operand marking. The permutation matrix ($\mathbf{P}$) is created by randomly shuffling the rows of an identity matrix, while the inverse permutation matrix is defined such that doing the matrix multiplication between the permutation matrix and its inverse results in the identity matrix. In this case, the inverse permutation matrix is equals to the transpose of the permutation matrix ($\mathbf{P}^T$), since $\mathbf{P}\mathbf{P}^T = \mathbf{I}$.

To illustrate the application of the permutation matrices, the example described in Section 2.2.7 is re-used below. As in that section, the "$\mathsf{\textbf{POS}}_\triangle$" and "$\mathsf{\textbf{POS}}_\square$" semantic pointers are replaced with non-commutative enforcers. Here, instead of operand markings, two permutation matrices ($\mathbf{P}_\triangle$ and $\mathbf{P}_\square$) will be used to "mark" the "$\mathsf{\textbf{POS}}_\triangle$" and "$\mathsf{\textbf{POS}}_\square$" semantic pointers respectively. With this, Equation (2.9) is redefined as follows:

$$\mathsf{\textbf{FIGURE4}} = (\mathsf{\textbf{RED\_TRIANGLE}})\,\mathbf{P}_\triangle + (\mathsf{\textbf{BLUE\_SQUARE}})\,\mathbf{P}_\square$$

Once again, the query is regarding the colour of the leftmost object in the image. In order to perform the query, the query semantic pointer is inverted, permuted using the "$\mathsf{\textbf{POS}}_\triangle$" permutation matrix, and then bound with the **FIGURE4** semantic pointer:

$$\begin{aligned}
&\mathsf{\textbf{FIGURE4}} \circledast (\neg\mathsf{\textbf{COLOUR}})\,\mathbf{P}_\triangle \\
&\quad = ((\mathsf{\textbf{RED\_TRIANGLE}})\,\mathbf{P}_\triangle + (\mathsf{\textbf{BLUE\_SQUARE}})\,\mathbf{P}_\square) \circledast (\neg\mathsf{\textbf{COLOUR}})\,\mathbf{P}_\triangle \\
&\quad = [(\mathsf{\textbf{RED\_TRIANGLE}})\,\mathbf{P}_\triangle \circledast (\neg\mathsf{\textbf{COLOUR}})\,\mathbf{P}_\triangle] + [(\mathsf{\textbf{BLUE\_SQUARE}})\,\mathbf{P}_\square \circledast (\neg\mathsf{\textbf{COLOUR}})\,\mathbf{P}_\triangle] \\
&\quad = [(\mathsf{\textbf{RED\_TRIANGLE}} \circledast \neg\mathsf{\textbf{COLOUR}})\,\mathbf{P}_\triangle] + \ldots \\
&\quad = [(\mathsf{\textbf{SHAPE}} \circledast \mathsf{\textbf{TRIANGLE}} \circledast \neg\mathsf{\textbf{COLOUR}} + \mathsf{\textbf{RED}} + \mathsf{\textbf{SIZE}} \circledast \mathsf{\textbf{SMALL}} \circledast \neg\mathsf{\textbf{COLOUR}})\,\mathbf{P}_\triangle] + \ldots
\end{aligned}$$

Unlike in Section 2.2.7, since the cleanup dictionary only contains the non-permuted forms of the semantic pointers, the cleanup operation cannot be performed directly on the result above. Instead, the above result needs to be permuted with the inverse permutation matrix (in this case,

$\mathbf{P}^{\mathbf{T}}_{\triangle}$) before performing the cleanup operation:

$$\begin{aligned}
&\mathit{cleanup}\{(\textsf{\textbf{FIGURE4}} \circledast (\neg\textsf{\textbf{COLOUR}})\,\mathbf{P}_{\triangle})\,\mathbf{P}^{\mathbf{T}}_{\triangle}\} \\
&\quad = \mathit{cleanup}\{((\textsf{\textbf{SHAPE}} \circledast \textsf{\textbf{TRIANGLE}} \circledast \neg\textsf{\textbf{COLOUR}} + \textsf{\textbf{RED}} + \\
&\qquad\qquad \textsf{\textbf{SIZE}} \circledast \textsf{\textbf{SMALL}} \circledast \neg\textsf{\textbf{COLOUR}})\mathbf{P}_{\triangle})\mathbf{P}^{\mathbf{T}}_{\triangle} + \\
&\qquad\qquad ((\textsf{\textbf{BLUE\_SQUARE}})\,\mathbf{P}_{\square} \circledast (\neg\textsf{\textbf{COLOUR}})\,\mathbf{P}_{\triangle})\,\mathbf{P}^{\mathbf{T}}_{\triangle}\} \\
&\quad = \mathit{cleanup}\{(\textsf{\textbf{SHAPE}} \circledast \textsf{\textbf{TRIANGLE}} \circledast \neg\textsf{\textbf{COLOUR}} + \textsf{\textbf{RED}} + \\
&\qquad\qquad \textsf{\textbf{SIZE}} \circledast \textsf{\textbf{SMALL}} \circledast \neg\textsf{\textbf{COLOUR}})\mathbf{I} + \\
&\qquad\qquad ((\textsf{\textbf{BLUE\_SQUARE}})\,\mathbf{P}_{\square} \circledast (\neg\textsf{\textbf{COLOUR}})\,\mathbf{P}_{\triangle})\,\mathbf{P}^{\mathbf{T}}_{\triangle}\} \\
&\quad = (\cancel{\textsf{\textbf{SHAPE}} \circledast \textsf{\textbf{TRIANGLE}} \circledast \neg\textsf{\textbf{COLOUR}}} + \textsf{\textbf{RED}} + \cancel{\textsf{\textbf{SIZE}} \circledast \textsf{\textbf{SMALL}} \circledast \neg\textsf{\textbf{COLOUR}}}) + \\
&\qquad \cancel{((\textsf{\textbf{BLUE\_SQUARE}})\,\mathbf{P}_{\square} \circledast (\neg\textsf{\textbf{COLOUR}})\,\mathbf{P}_{\triangle})\,\mathbf{P}^{\mathbf{T}}_{\triangle}} \\
&\quad = \textsf{\textbf{RED}}
\end{aligned}$$

It should be noted that the limitations discussed in Section 2.2.7 also apply to this method of generating non-commutative variants of the HRR-SPA operators.

2.3.9 Discussion

This section presented a specific implementation of the SPA using the Holographic Reduced representations as a basis for generating and manipulating the numerical representation of the semantic pointers. Discussed were the techniques used to generate a HRR-SPA semantic pointer, and the use of the circular convolution and superposition operations, used respectively to implement the binding and collection SPA operators. In addition, numerically grounded examples of special semantic pointers (null, identity, inverse and unitary semantic pointers) were also examined.

Continuing the discussion regarding compression, and the idea of pointers and pointer dereferencing (see Section 2.2.7), it can be shown that this concept is also evident in the HRR-SPA.

2.3.9.1 Compression

Compression in the HRR-SPA comes in multiple forms. As discussed in Section 2.2.7, each semantic pointer can represent any combination of the SPA operators on multiple semantic pointers. Furthermore, in the HRR-SPA, regardless of the number of SPA operations performed, the semantic pointers maintain a fixed dimensionality, necessitating that all of the information provided in the creation of a semantic pointer is compressed into the predefined dimensionality. This is in contrast to other encoding techniques (e.g., tensor products [Smolensky, 1990], the binary spatter code [Kanerva, 1996]) where the dimensionality of the representation increases for each operation performed. Additionally, with the use of the normalization operation (see Section 2.3.6), information stored within a semantic pointer are not only compressed to a fixed dimensionality, but also compressed to a fixed magnitude in vector space, usually resulting in the loss of information in the process.

2.3.9.2 Dereferencing Pointers

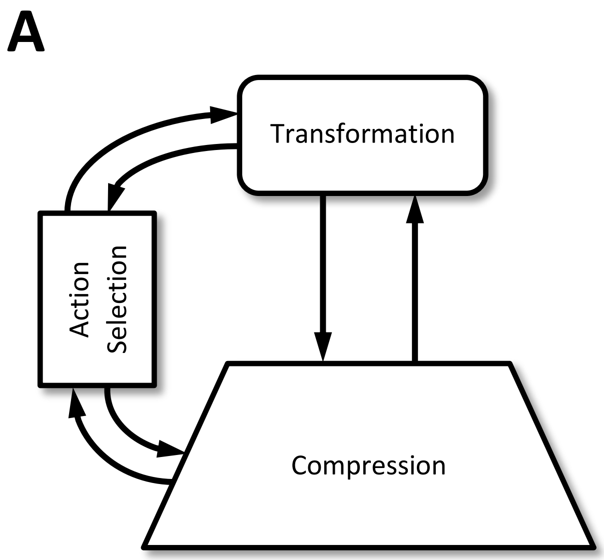
In the HRR-SPA, all of the analogies discussed in Section 2.2.7 still apply to the idea of the dereferencing of semantic pointers. Moreover, the use of the vector normalization operation on semantic pointers result in semantic pointers that represent an "averaged" amalgamation of its constituent semantic pointers which is arguably a more abstract form of representation of information when compared the symbol-like SPA semantic pointers. Because of this lossy compression, the use of the cleanup operation is necessary to successfully extract information from semantic pointers. Once again, the cleanup operation is an operation that takes an abstract value as an input, and produces known concepts as an output, furthering the analogy of how semantic pointers behave like pointers in computer systems.

2.4 The Semantic Pointer Architecture: Cognitive Computation with a Standardized Structure

The previous sections have thus far only discussed the symbol-like "cognitive" SPA in abstract terms, with no mention of how it would be implemented as a physical system. This section explores the standardized structure (referred to in [Eliasmith, 2013] as a "subsystem") which the SPA defines as the common building block of all physically realized SPA systems (see Figure 2.6B). As illustrated in Figure 2.6A, the SPA subsystem consists of three major components: the mechanism for semantic pointer compression, the mechanism to perform operations on the semantic pointers, and the mechanism to control the flow of information throughout the subsystem.

2.4.1 Semantic Pointer Compression

The SPA does not impose any restrictions on the methods used to create compressed semantic pointer representations, as long as the compressed semantic pointer representations adhere to the list of criteria listed in Section 2.3. This section briefly introduces two compressive mechanism used in Spaun: a memory-based network, and a compressive hierarchical network.



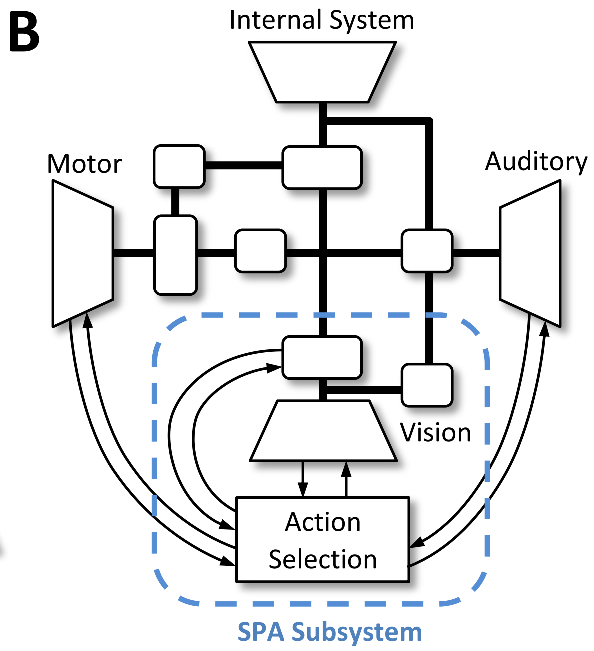


Figure 2.6: Illustration of the SPA subsystem and a generic SPA model. (**A**) The schematic of an SPA subsystem showing the three component parts: the semantic pointer compression mechanism, the semantic pointer transformation system and the action selection system. (**B**) The schematic of a generic SPA model showing the integration of the SPA subsystem as part of the overall model. The model consists of several interconnected SPA subsystems with the semantic pointer representations being transmitted over the subsystem interconnections (dark solid lines). Each subsystem consists of one compression component, and at least one transformation component. In this particular SPA model, the action selection component is shared amongst all of the SPA subsystems. (Figures adapted from [Eliasmith, 2013] with permission.)

The memory-based compressive network involves the use of a memory element, a recurrent connection, and the desired SPA operations (see Figure 2.7A). In Spaun, networks with similar structures are used within the working memory system to generate the compressed representations used as Spaun's internal "cognitive" semantic pointers. As an example of this compressive method, the semantic pointer for the list $[\mathbf{A}, \mathbf{B}, \mathbf{C}]$ – represented as the semantic pointer $\mathbf{L} = \mathbf{A} + \mathbf{B} + \mathbf{C}$ – can be constructed by configuring the network to recursively apply the HRR-SPA collection operator, and then sequentially presenting each item in the list to the network (see Figure 2.7B).

The hierarchical network method for compression proposes the use of a multi-layer topology

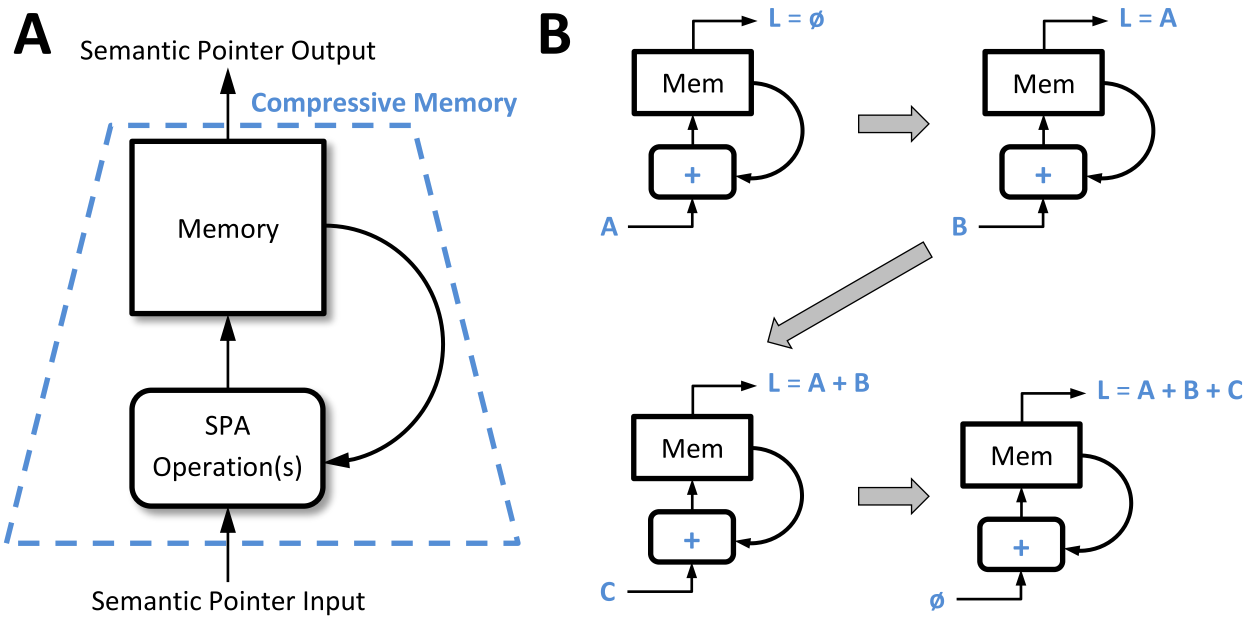


Figure 2.7: Illustration of a generic memory component for compressing semantic pointer representations. (**A**) Schematic of a generic SPA memory component. The memory component consists of a memory unit, recursively applied SPA operations, and a recurrent connection from the memory unit to the SPA operations. (**B**) Illustration of how the SPA memory component can be used to recursively compute the semantic pointer $\mathbf{L} = \mathbf{A} + \mathbf{B} + \mathbf{C}$. The process starts in with the top-left diagram, followed by the top-right, then the bottom-left, and finally the bottom-right diagram. For the memory network to construct $\mathbf{L}$, the recursive SPA operation needed is the collection operation. It should be noted that the semantic pointer value in the recurrent projection is the same as the value of the output of the SPA memory component. In the first image, the memory unit contains no value ($\emptyset$), and the input to the memory is $\mathbf{A}$. When a control signal is provided, the result of ($\emptyset + \mathbf{A}$) is fed into the memory unit, and the input to the memory is changed to $\mathbf{B}$, resulting in the second image. This process is repeated two more times with the inputs $\mathbf{C}$ and $\emptyset$, resulting in the final image where the value stored in the memory unit is the desired $\mathbf{L} = \mathbf{A} + \mathbf{B} + \mathbf{C}$.

to create compressed semantic pointer representations. Each layer takes, as input, the semantic pointer representation of the previous layer, and compresses it (commonly resulting in a change in the dimensionality of the semantic pointer) to produce the input to the next layer in the hierarchy. This method is inspired by the hierarchical topologies found in neurobiology, especially in the

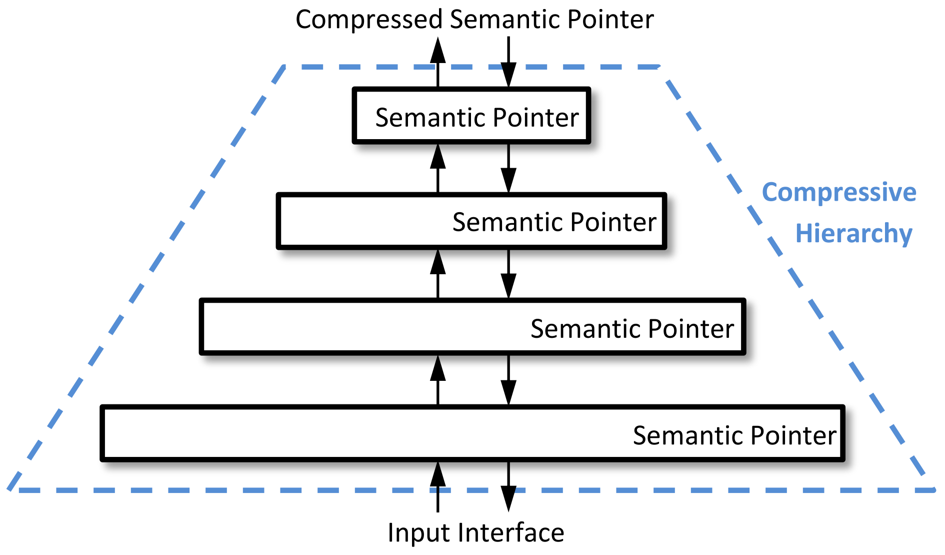


Figure 2.8: Illustration of a generic compressive hierarchy for semantic pointer generation. Shown is a four-layer hierarchical structure that receives input (e.g., raw visual input) through the "input interface" and uses the data to create a compressed semantic pointer representation at each layer. The semantic pointer representation at the top-most layer is considered the final compressed "output" semantic pointer of the hierarchy. The downward facing arrows indicate that the hierarchy can be used as a generative model, meaning that semantic pointers can be input to the top of the hierarchy and dereferenced by moving down the layers. (Figure adapted from [Eliasmith, 2013] with permission.)

sensory processing areas of the brain, the primary example being areas V1 through IT of the visual cortex in the human brain.[7] Logically, this compressive method is used within Spaun's visual system to generate Spaun's "visual" semantic pointers. Hierarchical networks also leverage the work done in the machine learning community in building statistical models that process sensory information (see Section 2.5.6.2). Figure 2.8 illustrates a generalized hierarchical network where each layer compresses the semantic pointer representation of the previous layer.

2.4.2 Semantic Pointer Transformation

The transformation mechanism of an SPA subsystem contains all of the components needed to perform the SPA operations (collecting, binding, or other brain area specific computation) desired

[7]The visual pathway of the human brain starts with the primary visual cortex (V1), then visual area V2, followed by visual area V4, and lastly the inferior temporal cortex (IT).

for the system.

2.4.3 Information Flow Control

The last component of the SPA subsystem is some mechanism to control the flow of information. The SPA network structure is meant to serve as a framework for neural systems which, by their nature have components that operate concurrently. To ensure that the appropriate pieces of information arrive at different parts of the network only when they are needed, the SPA proposes the use of an "action selection" system. The action selection system consists of an action selector that monitors the current state of the system and generates the signals necessary to configure the information flow in the rest of the system. These control signals are used by gates to either allow or inhibit the flow of information within each SPA subsystem, and also between SPA subsystems.

2.4.4 Discussion

This section described the general structure of the SPA subsystem. As will be discussed in Section 3.3, various forms of this structure are used to construct the Spaun cognitive model. The specific neural implementations of each component of the SPA subsystem are further explored in Section 2.5.6.

It should be noted that the description of the SPA subsystem thus far portrays it as a static (unchanging) system. However, this is not the case, and accommodations have been built into the SPA subsystem to allow it to be modified using externally or internally generated error signals. As illustrated in Figure 2.9, both the compression and transformation mechanisms can be used to generate error signals that can be used to modify the behaviour of each part of the SPA subsystem.

2.5 The Semantic Pointer Architecture and the Neural Engineering Framework: Cognitive Computation with Neurons

Thus far, the SPA has been used to demonstrate how cognitive computation can be done symbolically with abstract symbols, and numerically in vector space. This section starts with a basic description of the operating principles of a neuron in biology, followed by the neuron model used in Spaun. Next, the principles of the neural engineering framework (NEF) are outlined, and the section concludes with examples of how the NEF can be combined with the vector-based SPA computations described in previous sections to construct the computational SPA neural networks that serve as the building blocks of the Spaun model.

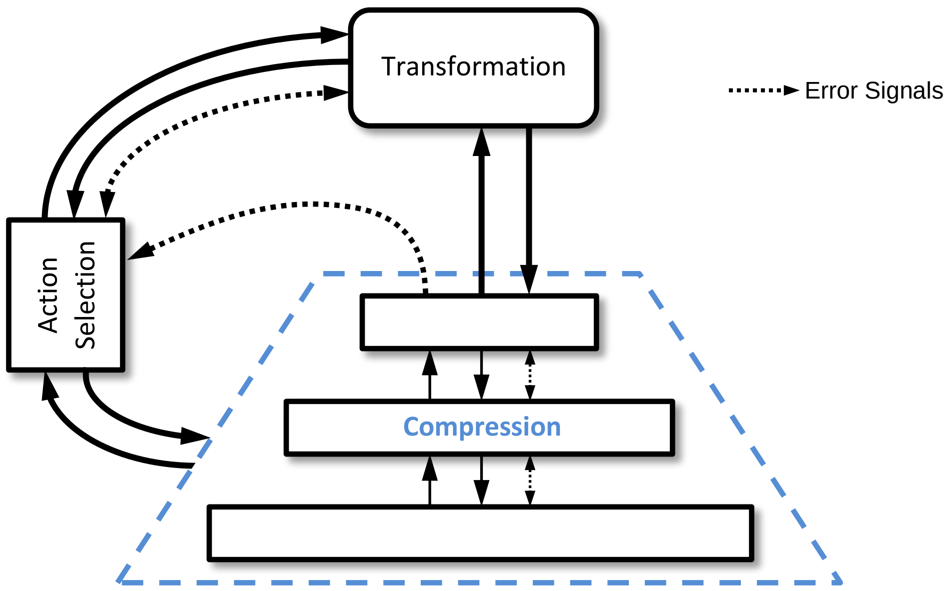


Figure 2.9: Schematic of an SPA subsystem showing error signals that can be used in conjunction with learning algorithms to modify parts of the subsystem. (Figure adapted from [Eliasmith, 2013] with permission.)

The text and figures in this section have been adapted from [Choo, 2010].

2.5.1 Neuron Basics

One of the primary motivations behind the Spaun model is to demonstrate that cognitive computation can be performed in a neural network which takes inspiration (and constraints) from the human brain. The human brain itself consists of roughly 10 to 100 billion neurons. In the pre-frontal cortex, which is believed to be important to the cognitive abilities of humans [Goldman-rakic, 1995; Henson et al., 2000], most of these neurons are large pyramidal neurons [DeFelipe and Jones, 1988; Elston, 2003], and this is the main type of neuron that the neurons in Spaun try to emulate (see Section 2.5.3); although there are also a variety of other neuron types used throughout the model, including medium spiny cells and gabaergic interneurons.

A closer look at a single neuron reveals how they function. Individually, each neuron can be considered a self-enclosed processing unit, with multiple inputs and in the case of the large pyramidal neurons, one output. The inputs to a neuron are called dendrites, and the output of the neuron is called the axon. Neurons communicate with each other in the form of electrical spikes, which are generated in the body (soma) of the neuron. These spikes then travel down the length of the axon, and to the dendrites of any neuron which the axon is connected to. The arrival of the spike at this junction (called a synapse) causes the release of neurotransmitters from the presynaptic neuron that flow across the gap between the neurons (the synaptic cleft) and bind to neuroreceptors on the efferent neuron. These neurotransmitters then cause a flow of ionic current into the dendrite that then flows down the dendrite and into the soma. The soma sums all of the input current coming from all of the dendrites, and if it exceeds a certain threshold, causes a spike to be sent down the axon of the neuron, restarting the entire spike generation process in the neurons it is connected to. Figure 2.10 shows what a typical large pyramidal cell looks like and also illustrates the process of spike generation.

2.5.2 Generating a Spike

Because neurons are physical systems, they undergo physical changes to generate spikes (i.e., the spike generation is not instantaneous), and the process of generating a spike results in a very distinctive profile in the voltage measured across the neuron's cell membrane.

As mentioned in the previous section, the arrival of a spike at a synapse causes current to flow down the dendrite and into the soma. This current flow is referred to as the postsynaptic current (PSC). When a spike arrives at a synapse, the PSC peaks, and then decays back to zero. The shape and duration of the PSC depends on the neurotransmitters being utilized at the synapse (see Figure 2.15 for an example).

The influx of current into the soma results in a buildup of electric charge within the cell body, increasing the difference in electric charge between the interior and exterior of the neuron, known as the membrane potential. The neuron's cell membrane – the wall that separates the interior and exterior of the neuron – prevents this charge from immediately equalizing. Instead, electric charge slowly leaks out of the cell through a type of ion channel. If the rate of current flow into the soma exceeds the rate at which current leaks from the cell, then current will build up in the cell, increasing the membrane potential (this is called depolarization).

When the membrane potential exceeds a certain level; called the spiking threshold; gates (ion channels) in the cell membrane open, first causing more charge to flow quickly into the cell, and then a very short time after, cause the charge to quickly exit the cell. This rapid up and down movement in the membrane potential is called an action potential, or "spike" because of its characteristic shape. After the spike is generated, the ion channels in the membrane remain open for some time, called the refractory period, to allow the membrane potential to equalize back to its resting state. During this time, no spikes will be generated regardless of the magnitude of the input current provided to the neuron. After the refractory period however, if the input current is sufficient to drive the membrane potential past the threshold level, another spike will

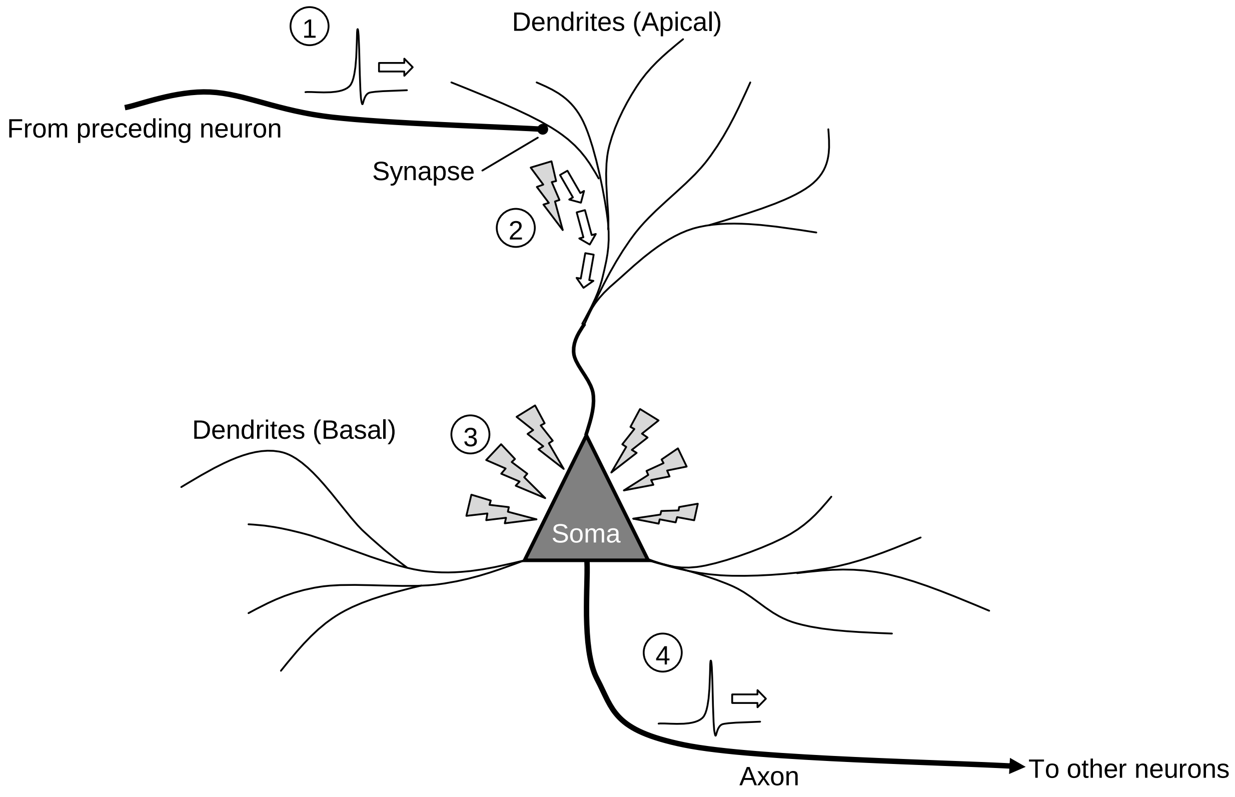


Figure 2.10: An simple schematic of a large pyramidal neuron detailing the dendrites and axon. The illustration also shows the effects of receiving an incoming spike at the neuron's dendrites.
① Spike from preceding neuron travels down its axon and arrives at the synapse.
② Dendrites to send current to the soma.
③ Membrane potential increases in the soma.
④ A spike is sent down the axon.
(Figure reproduced from [Choo, 2010] with permission.)

be generated, repeating the cycle of rapid depolarization and subsequent return to equilibrium of the membrane potential. Measuring the membrane potential throughout this process produces a graph similar to the one shown in Figure 2.11.

One must note that the initiation of a spike happens at the base of the soma, where the axon originates. The electric charge that flows into this area of the cell during the rapid upswing of the action potential will disperse within the cell body, causing the membrane potential in the area slightly downstream of the trigger area to rise and then exceed the spiking threshold. Essentially, a spike originating in the trigger area will cause another spike slightly downstream of the trigger area, and like a row of dominoes, the spike will continue to propagate in this manner, travelling like a wave down the length of the axon.

2.5.3 Simulating Neurons

As the functional mechanism of a neuron is complex, the degree to which neurons are simulated varies. Some models simulate neurons by including details of individual ion channels and their effects on the neurons membrane potential, as well as taking into account the physical size of the neurons and the effect this has on the speed of spike propagation [Carnevale and Hines, 2006]. Others abstract the behaviour of the membrane potential with a simple mathematical equation, and treat all neurons as point sources, ignoring the effects of the size of the neuron on spike propagation speed. Spaun, which is composed of millions of neurons, takes the latter approach, to reduce the computational requirements needed to run the model.

The neurons used in Spaun use the "leaky-integrate-and-fire" (LIF) neuron model. The LIF neuron has been shown to simulate the spiking behaviour of the pyramidal neurons found in the human neocortex[8] [Rauch et al., 2003].

The LIF neuron model is based off an electrical circuit known as an "RC" (resistor-capacitor) circuit. In an RC circuit, given a constant input current, the capacitor will slowly accumulate charge. When the input current is removed, the resistor will cause the accumulated charge to dissipate, and it is these two behaviours that make it ideal to simulate the membrane potential of a neuron – the capacitor emulating the accumulation of charge across the cell membrane in the soma, and the resistor emulating the slow leak current out of the soma. The combination of the resistor and capacitor values determine the "RC time constant" (τ^{RC}), which is the rate at which electric charge is accumulated and dissipated. The changes in the voltage across an RC circuit – or the change in voltage across the cell membrane in the LIF neuron model – can be

[8]Neocortex refers to the "new" parts of the brain. The term "neocortex" is used interchangeably with the term "cerebral cortex" which is on the "surface" of the brain. Older cortical structures are generally found deeper within the brain.

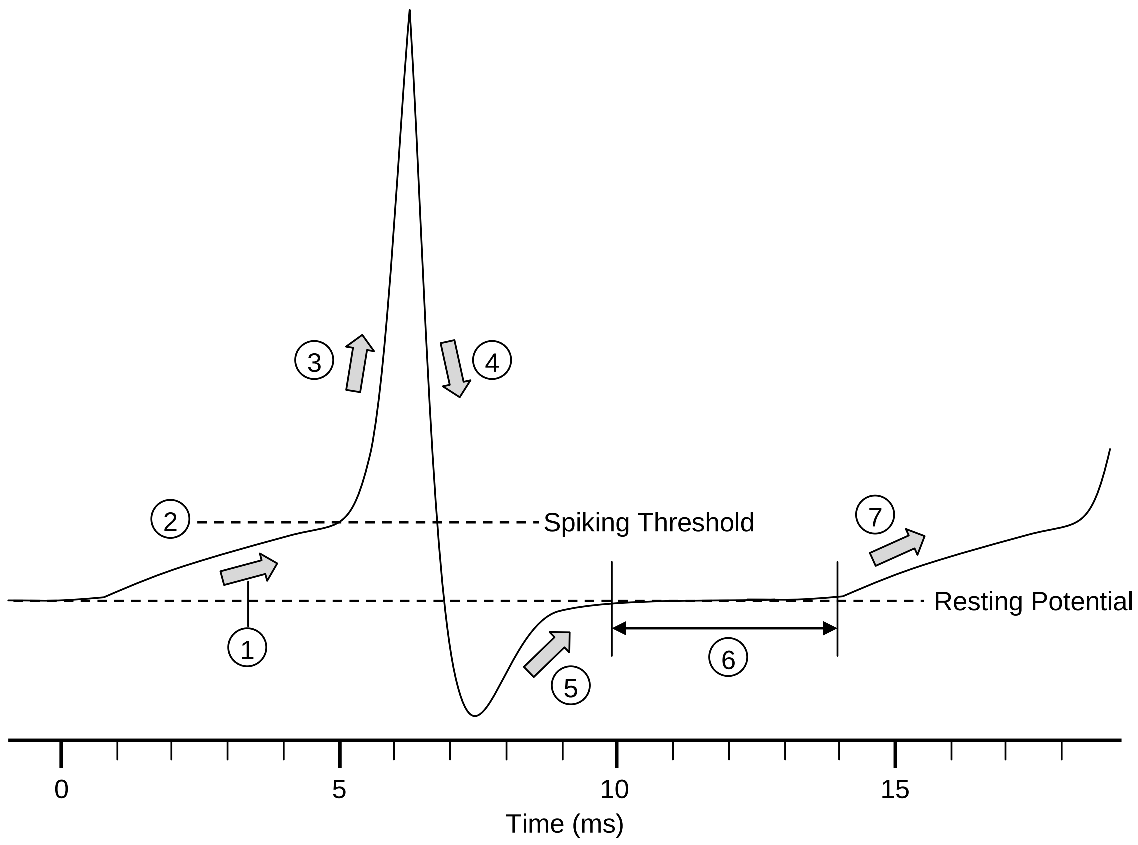


Figure 2.11: Illustration of the components of an action potential (spike) generated by a typical large pyramidal neuron. The graph shown is a plot of the membrane potential measured at the trigger zone of the neuron.
① Current from dendrites cause increase in membrane potential.
② Membrane potential exceeds spiking threshold.
③ Gates in the cell membrane open, causing a large influx of electric charge into the cell.
④ Followed by a large outflow of electric charge out of the cell.
⑤ Membrane potential returns to resting conditions.
⑥ Refractory period.
⑦ Cycle continues if neuron is being provided more input current.
(Figure reproduced from [Choo, 2010] with permission.)

calculated using Equation (2.23).

$$\begin{aligned}\tau^{RC} &= RC,\\ \frac{dV}{dt} &= -\frac{1}{\tau^{RC}}(V - J_{in}R),\end{aligned} \tag{2.23}$$

where R and C are the resistor and capacitor values of the RC circuit, $\frac{dV}{dt}$ is the rate of change of electric charge in the neuron, V is the membrane potential of the neuron at a moment in time, and J_{in} is the total amount of current flowing into the soma from the dendrites. When the membrane potential of the simulated neuron reaches the threshold level, a spike is "pasted" in, and the membrane potential is reset to its resting level. The model then fixes the membrane potential at the resting level for the duration of the refractory period (τ^{ref}), after which it is allowed to accumulate incoming electric charge again. Figure 2.12 illustrates the simulation of the LIF neuron when provided with a constant input current. The LIF neuron model is explored in greater detail in Eliasmith and Anderson (2003), where they describe the full RC circuit used, as well as discuss the advantages and disadvantages of the LIF neuron model.

2.5.4 The Neural Engineering Framework

Given the description of the behaviour of a single neuron, it is hard to imagine how they can be used to perform the complex computations needed in the SPA. However, the Neural Engineering Framework [Eliasmith and Anderson, 2003] provides a systematic approach to designing neural networks within the constraints of the underlying neurobiology.

Starting from the level of a single neuron, and working up to the level of networks of populations of neurons, this section describes how the NEF can be used to create networks to compute the arbitrary functions needed by any application, including Spaun. It should be noted that while Spaun utilizes the NEF to build computational networks out of LIF neurons, the methods described by the NEF can be applied to any type of neuron.

2.5.4.1 Representation with Single Neurons

This section starts at the very root of the problem of performing complex computations in neural networks. Here, the issue to be solved is the use of a single neuron to perform the simplest computation – the representation of an arbitrary scalar value.

As described in the previous section, the primary principle behind the neuron's operation is the manipulation of input currents to the neuron that in turn alter the electric charge within the neuron. The first step to using neurons to represent physical values is to somehow link these

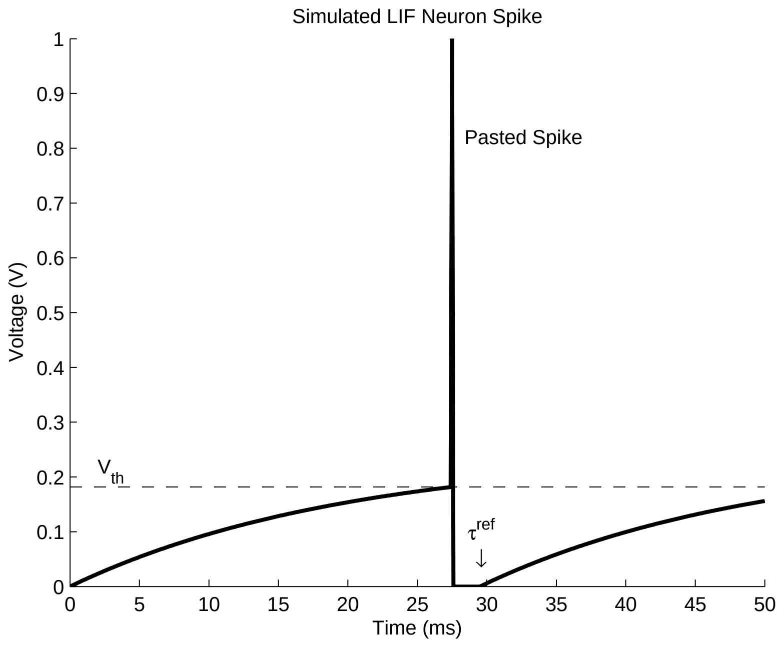


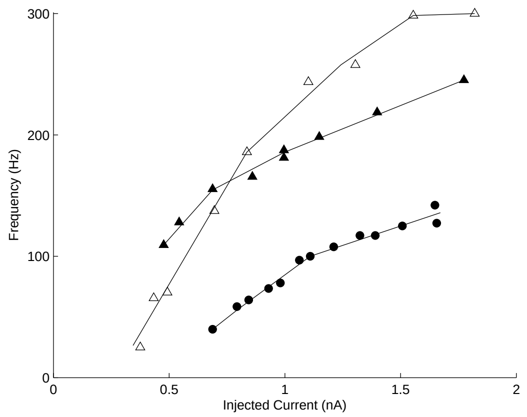
Figure 2.12: A simulation of the response of a leaky-integrate-and-fire (LIF) neuron given a constant input. The sub-threshold soma voltage curve was calculated using Equation (2.23). When the voltage exceeds the spiking threshold, V_{th}, a spike is inserted, and the voltage is reset to 0. The voltage is kept at 0 until a predefined period, τ^{ref}, has passed; after which, the voltage is allowed to increase again. (Figure reproduced from [Choo, 2010] with permission.)

input currents to the physical values that are to be represented. In the NEF, the assumption is made that there is a directly proportional relationship between the physical value x, and the total input current J to the neuron. This relationship can be written as:

$$J(x) = \alpha x + J^{bias}, \tag{2.24}$$

where $J(x)$ is the input current as a function of the variable x, α is a scaling factor that converts x into the appropriate units used by the input current, and J^{bias} is a background current that results from the background firing of all the neurons connected to the neuron we are modelling.

By measuring the steady-state fire rate of the neuron while varying this input current, the response curve of the neuron can then be plotted. Figure 2.13 compares the response curves of neocortical neurons and the response curves generated using the LIF neuron model. For the LIF model, the LIF equations (Equation (2.23)) can be used to derive the equations for the neuron



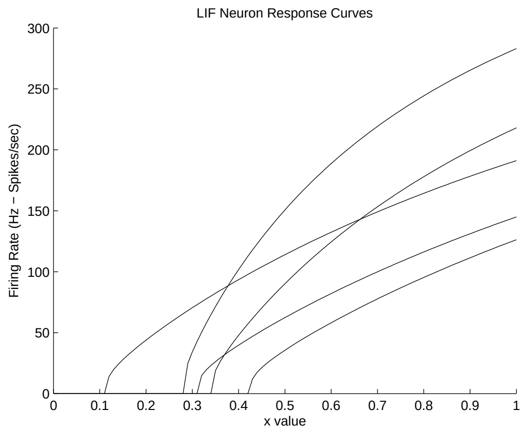


Figure 2.13: Example neuron response curves: (Left) Response curves plotted for regular-spiking neurons found in the guinea pig neocortex, from [McCormick et al., 1985]. (Right) Response curves generated by the LIF neuron model using Equation (2.25). Note that for this graph, the x-axis does not indicate the input current, but rather a value proportional to the input (injected) current. (Figures reproduced from [Choo, 2010] with permission.)

response curve, which is defined as:

$$a(x) = G[J(x)] = \begin{cases} \frac{1}{\tau^{ref} - \tau^{RC}\ln\left(1 - \frac{J^{th}}{J(x)}\right)}, & \text{if } J(x) > J^{th} \\ 0, & \text{otherwise} \end{cases} \tag{2.25}$$

where $a(x)$ is the activity of the neuron – reported in spikes per second – for the input value x; $G[\ldots]$ is the function that defines the non-linear response curve of the neuron to the input current $J(x)$; τ^{RC} and τ^{ref} are the RC and refractory time constants mentioned before; J^{th} is the threshold current above which the neuron will begin spiking, and is directly proportional to the spiking threshold level mentioned in the previous sections; and $J(x)$ is the total input current that was derived before.

The units of $a(x)$ are given in spikes per second (Hz), which is not ideal, and in the NEF, a scaling factor is used to “decode” the activity value back into the appropriate units for x. We can then write the estimated representation of x as:

$$\hat{x} = a(x)d, \tag{2.26}$$

where $\hat{x}$ represents the estimate of the value of x, and d is the scaling factor – known as a “decoder” in the NEF – used to convert the activity of the neuron back into the units and scale of x.

Even with this decoder, the use of a single neuron is insufficient for constructing a network that accurately represents the given value x. As the equations demonstrate, a linear change in x results in a non-linear change in $a(x)$, and no amount of scaling will reverse this non-linearity. In order to compensate for the non-linearity, additional neurons need to be recruited.

2.5.4.2 Representation using Populations of Neurons

The rationale behind the utilization of multiple neurons to achieve a better representation of a physical scalar value is similar to the explanation of how additional bits in floating point number (on a digital system) can be used to add precision to the value being represented.

In the NEF, the addition of neurons increases the number of non-linearities the network contains, which increases the representational capacity of the network. The NEF achieves this by performing a linear summation of scaled versions of the neural response curves, as illustrated in Figure 2.14

From Equation (2.26) it can be seen that the independent scaling of the response curves can already be achieved by manipulating the decoder values for each neuron. The reconstructed estimate of the input x for a population (network) of n neurons can then be re-written as:

$$\hat{x} = \sum_{i=1}^{n} a_i(x) d_i, \tag{2.27}$$

where a_i and d_i are the response curve and the decoder for the i^{th} neuron, respectively. One approach to obtaining the decoder values needed to accurately reconstruct the input value is to solve for the decoders using the method of least squares, which results in the following equations:

$$d = \mathbf{\Gamma}^{-1}\mathbf{\Upsilon}, \qquad \text{where} \qquad \mathbf{\Gamma} = \mathbf{A}\mathbf{A}^T \qquad \text{and} \qquad \mathbf{\Upsilon} = \mathbf{A}\mathbf{X}^T \tag{2.28}$$

In the equation above, $\mathbf{A}$ is the population's activity matrix[9], and $\mathbf{X}$ is a vector of x values that the population needs to represent. A detailed derivation of the equations used to compute the decoders can be found in [Eliasmith and Anderson, 2003, Appendix A].

It is important to note that within the NEF, the decoders are solved to optimize the representational accuracy over a specified range. In the case of Equation (2.28), the decoders are

[9]The activity matrix is a matrix that is constructed from the neuron response curves from all of the neurons in the neural network. Each row in this matrix corresponds to the neuron response curve of one neuron in the population of neurons.

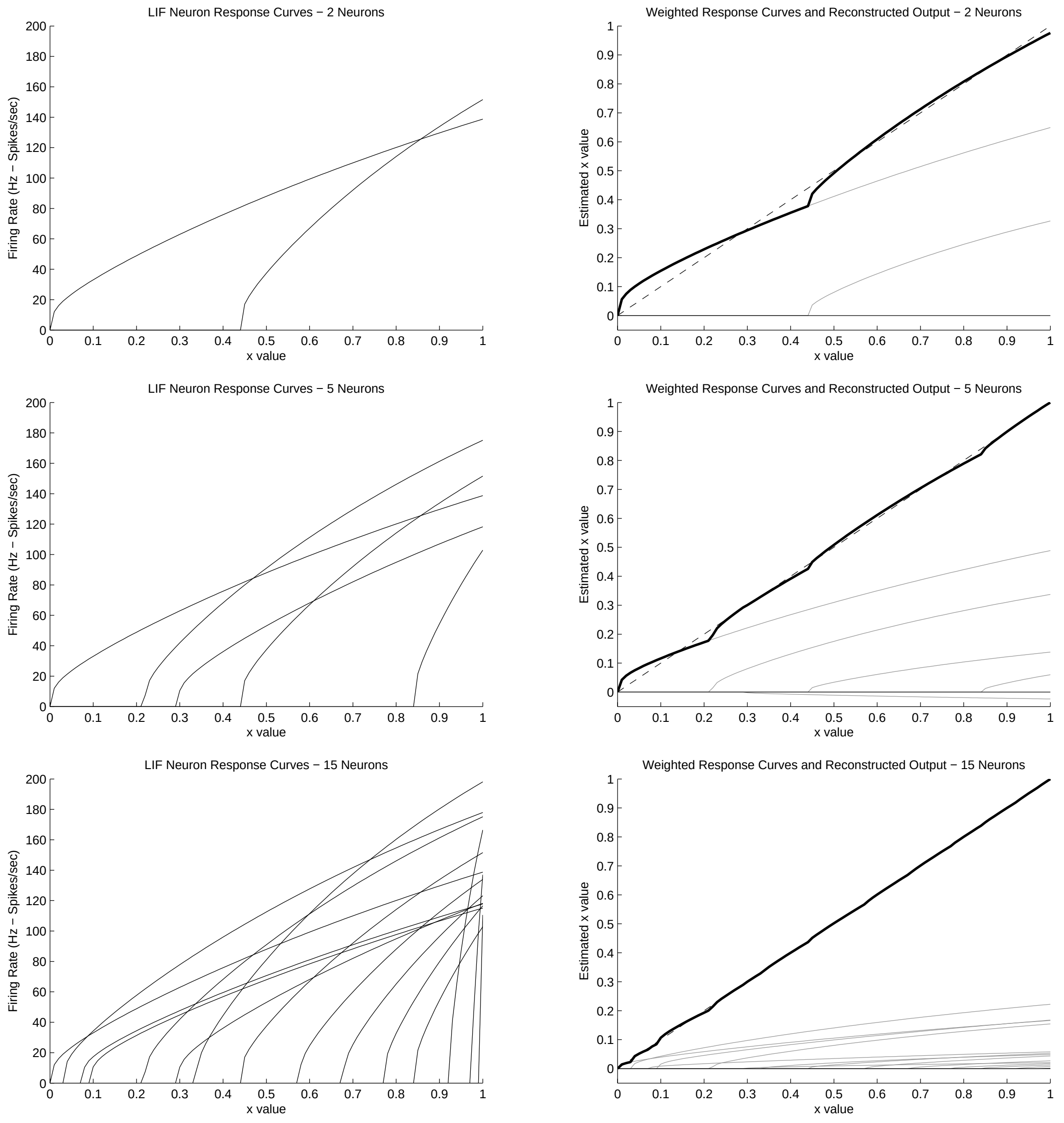


Figure 2.14: An illustration of how the response curves from multiple neurons can be appropriately scaled to estimate the input x value. In the left column, the original response curves for the neurons are shown. In the right column, the rescaled version of each response curve is shown (solid grey lines), in addition to the reconstructed x values (solid black line), and the ideal reconstruction of the x values (dashed line). The reconstructed x values are calculated by adding together the weighted response curves. Note that the response curves that lie below the zero line are response curves that have been scaled by a negative amount. From top to bottom, it is observed that as the number of neurons increases (from 2 to 5 to 15), the reconstructed estimate x improves. (Figure reproduced from [Choo, 2010] with permission.)

optimized such that the representational accuracy of the neural population is highest for the range of x values in the $\mathbf{X}$ vector.

2.5.4.3 Representation in Higher Dimensions

In 1986, Georgopoulos et al. recorded the activity of neurons from the motor cortex of rhesus monkeys while the monkeys were doing a task involving the movement of their arm in one of 8 spatial directions. They made the observation that these neurons would fire maximally when the monkey was commanded to move its arm, and that the direction of the arm's movement would affect which neuron would be most active. In spatial coordinates, each one of these "preferred directions" can be thought of as a two dimensional vector originating from the initial arm position and pointing toward the final arm position. In essence, each neuron was encoding a two-dimensional vector value as a spike rate. By knowing the preferred direction for each neuron, and the firing rate of each neuron, it would then be possible to get a reconstruction of the two-dimensional vector direction that the monkey had been commanded to move.

The NEF uses a similar concept to enable the representation of multi-dimensional vector values in neural populations. As with the observations Georgopoulos made, each randomly generated neuron in the NEF is given a randomly generated "encoder" vector with the same dimensionality of the vector value being represented. These encoders represent the preferred direction vector of each neuron. To compute the associated scalar value to be used in the neurons input current equation (Eq. (2.24)), input vector must be projected onto the preferred direction vector. In the NEF, this is done using the dot product operator, and thus, the input current equation can be modified to be:

$$J(\vec{\mathbf{x}}) = \alpha(\vec{\mathbf{e}} \bullet \vec{\mathbf{x}}) + J^{bias} \tag{2.29}$$

In the equation above, the input current J is now a function of $\vec{\mathbf{x}}$, the commanded (input) direction vector. Instead of being directly proportional to the input vector $\vec{\mathbf{x}}$, the current is now proportional to the dot product ($\bullet$) of the preferred direction vector ($\vec{\mathbf{e}}$) and $\vec{\mathbf{x}}$. Note that in the original definition of the input current equation, $J(x)$ is a scalar value. Since the dot product also produces a scalar value[10] the modified input current $J(\vec{\mathbf{x}})$ is also a scalar value. This means that it can be directly substituted into the neuron response function without needing to make any additional changes:

$$a(\vec{\mathbf{x}}) = G[J(\vec{\mathbf{x}})] = \begin{cases} \frac{1}{\tau^{ref} - \tau^{RC} \ln\left(1 - \frac{J^{threshold}}{J(\vec{\mathbf{x}})}\right)}, & \text{if } J(\vec{\mathbf{x}}) > J^{threshold} \\ 0, & \text{otherwise} \end{cases} \tag{2.30}$$

[10]In fact, the encoder can be thought of as projecting the input vector into the one-dimensional scalar space that represents the current being input to the neuron.

Reconstructing an estimate of the original input vector becomes slightly more complex because the result of the reconstruction should match the dimensionality of the input vector. Since the response curves of the neurons produce scalar values, performing the vector reconstruction requires increasing the dimensionality of the decoders to match the desired output dimensionality. Fortunately, calculating the decoders remains the same as in the scalar case,

$$\vec{\mathbf{d}} = \mathbf{\Gamma}^{-1}\mathbf{\Upsilon}, \qquad \text{where} \qquad \mathbf{\Gamma} = \mathbf{A}\mathbf{A}^T \qquad \text{and} \qquad \mathbf{\Upsilon} = \mathbf{A}\mathbf{X}^T, \tag{2.31}$$

with the exception that $\mathbf{X}$ is no longer a vector of x values that the population needs to represent, but rather a matrix of vectors that the population needs to represent. Performing the matrix calculation reveals that the dimensionality of the decoders does indeed match the dimensionality of the input vector. The reconstruction of the original input vector can then be computed as:

$$\hat{\mathbf{x}} = \sum_{i=1}^{n} a_i(\vec{\mathbf{x}})\vec{\mathbf{d}}_i \tag{2.32}$$

As in the scalar case, $\hat{\mathbf{x}}$ is the reconstruction of the input vector, a_i is the i^{th} neuron's response to the $\vec{\mathbf{x}}$, and $\vec{\mathbf{d}}_i$ is the i^{th} neuron's now multi-dimensional decoding vector.

2.5.4.4 Representation Over Time

The discussion thus far has been concerned about time-invariant values, be they scalar or multi-dimensional. However, being physical (and dynamic) systems, neurons almost exclusively operate with signals that vary over time. This implies that the framework that has been discussed so far must be extended to the temporal domain. However, the task of doing so is similar to the method used to extend the NEF from the representation of scalar to the representation of multi-dimensional values and requires minimal changes.

To perform an analysis in the temporal domain, rather than using the static input $\vec{\mathbf{x}}$, the time-varying input $\vec{\mathbf{x}}(t)$ is used. This signal can be thought of as vector that changes its direction over time. Now, as in the vector case, the input current is:

$$J(\vec{\mathbf{x}}(t)) = \alpha(\vec{\mathbf{e}} \bullet \vec{\mathbf{x}}(t)) + J^{bias} \tag{2.33}$$

Note that in this case, $J(\vec{\mathbf{x}}(t))$ changes with time because $\vec{\mathbf{x}}(t)$ is a time-varying signal. Calculating the response a of the neuron population becomes more complex because ideally, the response of the neurons should be in terms of spikes and the previous method of calculating the response only provided a static rate response rather than a response that changes over time. Using the mechanism described in Section 2.5.3 to generate the spiking behaviour, the individual spike response for each neuron in the population can be rewritten as:

$$a(\vec{\mathbf{x}}(t)) = \sum_s \delta(t - t_s), \tag{2.34}$$

where s is the number of spikes there are in the spike train, and $\delta(t - t_s)$ denotes the time of each spike. The function $\delta(t)$ is equal to 1 at $t = 0$ and 0 everywhere else.

In order to reconstruct the input signal $\vec{\mathbf{x}}(t)$, or in this case, to determine how the downstream neuron is interpreting the outgoing spike trains of the entire population, an additional mechanism is needed to "smooth" out the spike train into a time-varying signal. The NEF borrows from biology and uses the concept of applying a post-synaptic filter $(h(t))$ to the spike train to produce post-synaptic current input to the neuron.[11] Applying the post-synaptic filter $h(t)$ to the reconstruction equation results in:

$$\hat{\mathbf{x}}(t) = \left(\sum_{i=1}^{n} a_i(\vec{\mathbf{x}}(t))\vec{\mathbf{d}}_i \right) * h(t) \tag{2.35}$$

It should be noted that the decoders in the equation above are identical to the time-invariant cases. [Eliasmith and Anderson, 2003, Appendix B] provides further details as to the reasoning behind why it is possible to use the time-invariant decoders for the spiking neuron systems. The effect of applying (convolving) the post-synaptic filter with the spike train is illustrated in Figure 2.15.

As an aside, the convolution with the PSC signal $h(t)$ can also be performed in the calculation of the neuron response function a. Conceptually, this is a more accurate characterization of the neurobiology because the current flowing to the soma of the downstream neuron is the sum of all the PSC's from the dendrites, and not the sum of all the spike trains. The neuron responses can then be rewritten as:

$$a(\vec{\mathbf{x}}(t)) = \sum_s \delta(t - t_s) * h(t) = \sum_s h(t - t_s), \tag{2.36}$$

and the reconstructed estimate as:

$$\hat{\mathbf{x}}(t) = \sum_{i=1}^{n} a_i(\vec{\mathbf{x}}(t))\vec{\mathbf{d}}_i \tag{2.37}$$

[11] As mentioned in Section 2.5.2, the arrival of a spike at a synapse causes an influx of current into the dendrite that decays with an exponential time constant. With time-varying signals, this effect can be achieved by convolving a spike with the signal $h(t)$ that has the same exponentially decaying shape as the PSC

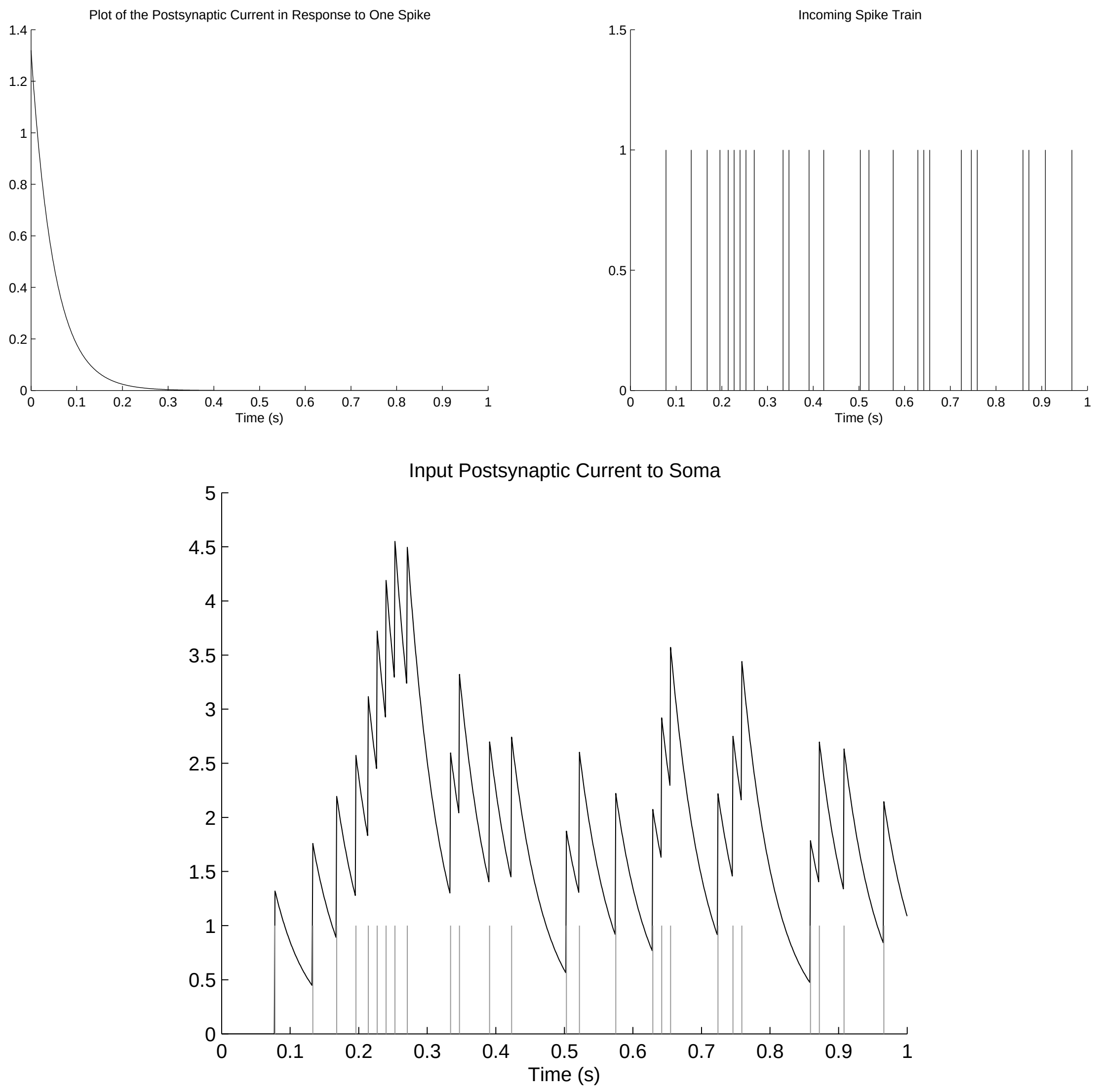


Figure 2.15: (Top Left) Plot of the shape of the PSC signal, $h(t)$, in response to one spike arriving at the synapse. (Top Right) The input spike train, $\sum_s \delta(t - t_s)$, used in this example. (Bottom) Plot of the total input current going into the soma, calculated using Equation (2.36). The input spike train is overlaid on this plot in grey. (Figure reproduced from [Choo, 2010] with permission.)

2.5.4.5 Representation of Arbitrary Transformations

Thus far, the discussion has been centered around the reconstruction of the original input signal (i.e., $\hat{x} \approx x$). While this has been valuable in deriving the formulae necessary to perform representations in neurons, it does not provide many interesting applications. This section therefore discusses how the framework can be easily extended to perform arbitrary transformations to the input signal.

Consider a weighted linear combination of two signals: $\vec{\mathbf{z}} = C_1\vec{\mathbf{x}} + C_2\vec{\mathbf{y}}$.[12] To perform this transformation, three populations are needed, one for each variable, to be connected as shown in Figure 2.16.

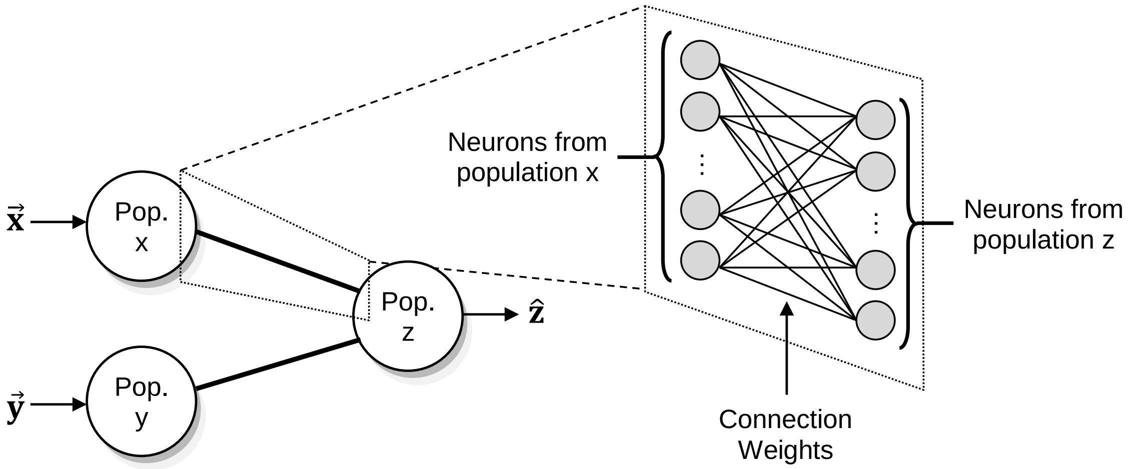


Figure 2.16: A connection diagram of the neural network needed to compute the transformation $\vec{\mathbf{z}} = C_1\vec{\mathbf{x}} + C_2\vec{\mathbf{y}}$. Each neuron in one population is fully connected with all of the neurons in the other population (shown in pop-up). (Figure reproduced from [Choo, 2010] with permission.)

In the following analysis, a superscript letter will be used to denote these populations. For example, $a^{\mathbf{x}}$ will be used to denote the neuronal response of the population responsible for the signal $\vec{\mathbf{x}}$. As with the analysis in the previous sections, the input current for one neuron in the output population – in this case the "z" population – is first derived.

$$J^{\mathbf{z}}(C_1\vec{\mathbf{x}} + C_2\vec{\mathbf{y}}) = \alpha(\vec{\mathbf{e}}^{\mathbf{z}} \bullet (C_1\vec{\mathbf{x}} + C_2\vec{\mathbf{y}})) + J^{bias} \tag{2.38}$$

Since the outputs of the "x" and "y" populations would be the reconstructions of the inputs, $\hat{\mathbf{x}}$

[12]Note that while the variables $\vec{\mathbf{x}}$, $\vec{\mathbf{y}}$, and $\vec{\mathbf{z}}$ are listed as time-invariant values here, the same analysis can be easily performed with time-varying signals. See Section 2.5.4.4

and $\hat{\mathbf{y}}$ are substituted for $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$, respectively. The equation for the input current then becomes:

$$\begin{aligned} J^{\mathbf{z}}(C_1\vec{\mathbf{x}} + C_2\vec{\mathbf{y}}) &= \alpha(\vec{\mathbf{e}}^{\mathbf{z}} \bullet (C_1\hat{\mathbf{x}} + C_2\hat{\mathbf{y}})) + J^{bias} \\ &= \alpha\left[\vec{\mathbf{e}}^{\mathbf{z}} \bullet \left(C_1 \sum_i a_i^{\mathbf{x}}(\vec{\mathbf{x}})\vec{\mathbf{d}}_i^{\mathbf{x}} + C_2 \sum_j a_j^{\mathbf{y}}(\vec{\mathbf{y}})\vec{\mathbf{d}}_j^{\mathbf{y}}\right)\right] + J^{bias} \\ &= \sum_i w_i^{\mathbf{x}} a_i^{\mathbf{x}}(\vec{\mathbf{x}}) + \sum_j w_j^{\mathbf{y}} a_j^{\mathbf{y}}(\vec{\mathbf{y}}) + J^{bias}, \end{aligned} \tag{2.39}$$

where $w_i^{\mathbf{x}} = \alpha C_1 \left(\vec{\mathbf{e}}^{\mathbf{z}} \otimes \vec{\mathbf{d}}_i^{\mathbf{x}}\right)$, and $w_j^{\mathbf{y}} = \alpha C_2 \left(\vec{\mathbf{e}}^{\mathbf{z}} \otimes \vec{\mathbf{d}}_j^{\mathbf{y}}\right)$, and $(\otimes)$ represents the outer product operation. In the general case, the scalar variables C_1 and C_2 may be substituted with the matrices $\mathbf{C}_1$ and $\mathbf{C}_2$. The weights $w_i^{\mathbf{x}}$ and $w_j^{\mathbf{y}}$ then become $w_i^{\mathbf{x}} = \alpha\left((\vec{\mathbf{e}}^{\mathbf{z}}\mathbf{C}_1) \otimes \vec{\mathbf{d}}_i^{\mathbf{x}}\right)$.

With the input current equation derived, the output $\vec{\mathbf{z}}$ can then be computed as:

$$a^{\mathbf{z}}(C_1\vec{\mathbf{x}} + C_2\vec{\mathbf{y}}) = G\left[J^{\mathbf{z}}(C_1\vec{\mathbf{x}} + C_2\vec{\mathbf{y}})\right] \tag{2.40}$$

$$\hat{\mathbf{z}} = \sum_{k=1}^{n} a_k^{\mathbf{z}}(C_1\vec{\mathbf{x}} + C_2\vec{\mathbf{y}})\vec{\mathbf{d}}_k^{\mathbf{z}} \tag{2.41}$$

Note that in Equation (2.40), the function $G[\ldots]$ is the non-linear neuron response as detailed in Equation (2.30), or for time-varying signals, a spike train, as in Equation (2.36).

2.5.4.6 Computation of Arbitrary Functions

The previous sections have described how the NEF can be used to compute linear projections and transformations of scalar and multi-dimensional values. In this section, one of the powerful advantages of the NEF method will be described – using the NEF to compute arbitrary non-linear functions in neural networks.

Surprisingly, the method by which neural populations can be used to compute arbitrary functions has already been discussed. Previous sections demonstrated how decoders can be calculated for each neuron in such a way as to allow the neural population to represent the input value x. In essence, the decoders have been computed to perform the identity function. Similarly, the same method can be used to solve for decoders that allow the neural population to compute – or more accurately, approximate – arbitrary functions. Figure 2.17 illustrates how this concept is applied to get a neural population to compute the non-linear function $f(x) = \sin(2\pi x)$. As with Figure 2.14, it can be seen that the accuracy of the approximated function improves with the number of neurons used in the network.

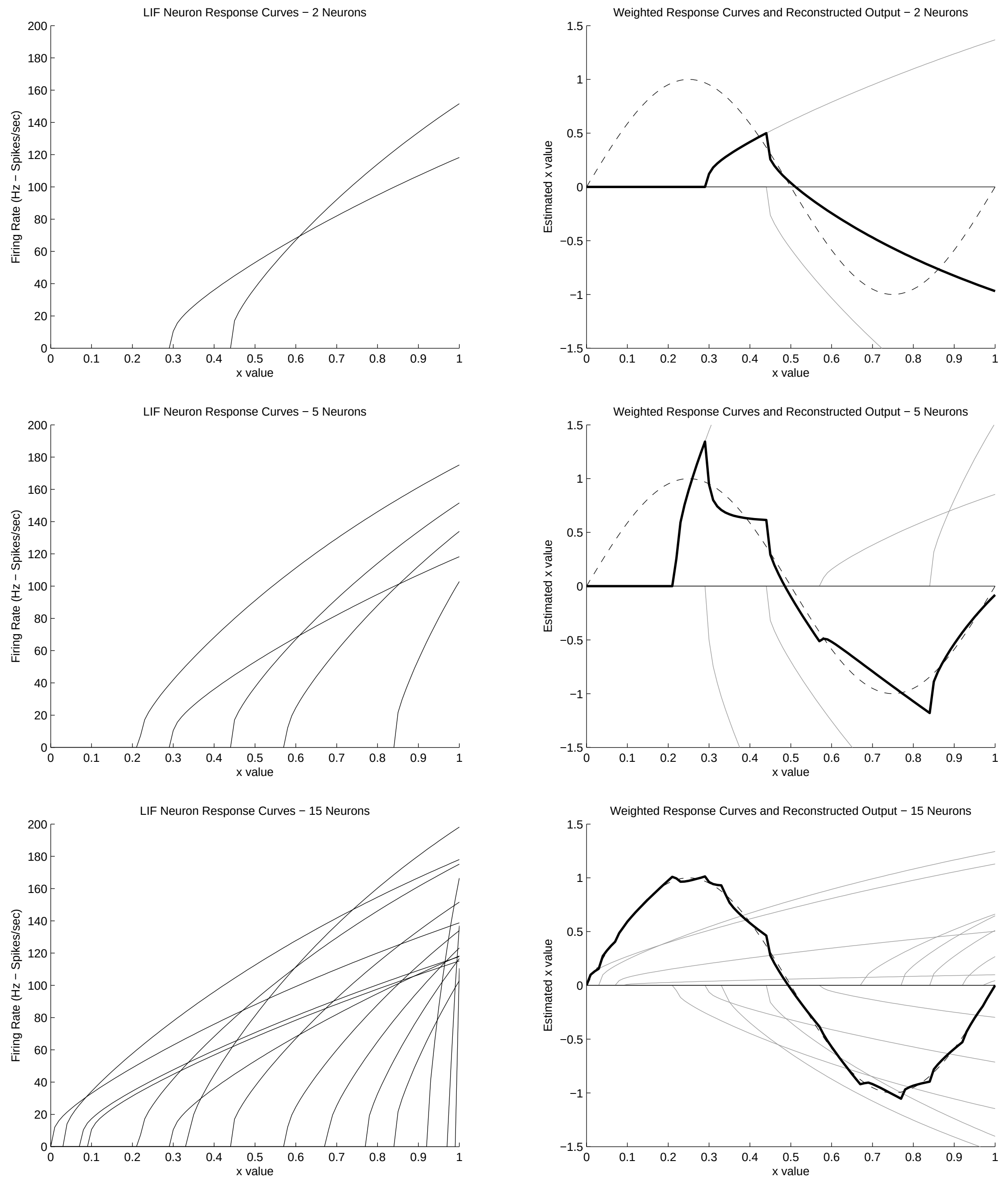


Figure 2.17: An illustration of how decoders can be used to weight the neuron response curves to reconstruct any desired function. The function used in this example is $\sin(2\pi x)$, and the ideal reconstruction of this function is denoted by the dotted line. The layout of this figure and the number of neurons used for each row are identical to Figure 2.14. (Figure reproduced from [Choo, 2010] with permission.)

Formally, the decoders for a population of neurons that computes the function $f(x)$ can be determined by modifying Equation (2.28) such that in the calculation of $\mathbf{\Upsilon}$, the values of the desired function applied to $\mathbf{X}$, $f(\mathbf{X})$, are used in place of $\mathbf{X}$:

$$d^{f(x)} = \mathbf{\Gamma}^{-1}\mathbf{\Upsilon}, \qquad \text{where} \qquad \mathbf{\Gamma} = \mathbf{A}\mathbf{A}^T \qquad \text{and} \qquad \mathbf{\Upsilon} = \mathbf{A}f(\mathbf{X})^T \tag{2.42}$$

This same method can also be extended to compute functions dependent on multiple inputs. In the next example, the neural implementation for the scalar non-linear function $z = xy$ is derived.[13] Since the desired function is a function of both x and y, the neural population needs to have information of both x and y in order to perform the computation. This is done by setting up a linear transformation (see Figure 2.16) to project the independent scalar x and y values into a 2-dimensional space, where each dimension is used to represent x and y, respectively:

$$\begin{aligned} \vec{\mathbf{m}} &= [x\ y] = [1\ 0]x + [0\ 1]y \\ &= C_1 x + C_2 y, \end{aligned} \tag{2.43}$$

where $C_1 = [1\ 0]$ and $C_2 = [0\ 1]$.

Then, to compute the multiplication function, the decoders of the "z" population are computed using the function $z = (m_1 \cdot m_2)$ – where m_1 is the first element of the vector $\vec{\mathbf{m}}$ (which contains the value of x), and m_2 is the first element of the vector $\vec{\mathbf{m}}$ (which contains the value of y) – to generate the appropriate $f(\mathbf{X})$ matrix in Equation (2.42). An alternative understanding of this method is to consider that the decoders for the neural population have been solved to compute the mapping of a 2-dimensional vector onto a 1-dimensional scalar which is derived as is the result of multiplying both elements in the vector together. As an example, part of this mapping is:

$$\begin{aligned} &[0.0, 0.0] \rightarrow 0 \\ &[0.0, 1.0] \rightarrow 0 \\ &[1.0, 1.0] \rightarrow 1 \\ &[0.5, 0.5] \rightarrow 0.25 \end{aligned}$$

2.5.4.7 Relating the NEF to Biology

The discussion about the NEF introduced abstract concepts such as neural encoders ($\mathbf{e}$) and decoders ($\mathbf{d}$), and one might get the misconception that these properties are intrinsic to each

[13]In this example, a scalar function is used only because of its simplicity. The same analysis, albeit more involved, can be performed for multi-dimensional functions.

neuron. Looking at the neurobiology, neurons receive inputs in terms of current, and produce outputs in terms of spikes, both of which are scalar values. This makes it difficult to reconcile the mechanism by which neurons represent these multi-dimensional encoder and decoder values. It is important to note that individual neurons in isolation do not have the ability to represent such objects. Rather, it is the configuration of the connection weight matrix between neurons that enables the neurons to represent such objects. The NEF merely presents a framework by which to calculate the connection weight matrix in such a way as to enable the neural population to compute a desired function.

A re-examination of the input current derived for a linear transformation of inputs to a neural population reveals that the encoders, decoders and transformation matrices can be mathematically combined to form a single connection weight matrix. Consider, as an example, the network described in Figure 2.16. The current equation for that network is:

$$\begin{aligned} J^{\mathbf{z}}(C_1\vec{\mathbf{x}} + C_2\vec{\mathbf{y}}) &= \alpha(\vec{\mathbf{e}}^{\mathbf{z}} \bullet (C_1\hat{\mathbf{x}} + C_2\hat{\mathbf{y}})) + J^{bias} \\ &= \alpha\left[\vec{\mathbf{e}}^{\mathbf{z}} \bullet \left(C_1 \sum_i a_i^{\mathbf{x}}(\vec{\mathbf{x}})\vec{\mathbf{d}}_i^{\mathbf{x}} + C_2 \sum_j a_j^{\mathbf{y}}(\vec{\mathbf{y}})\vec{\mathbf{d}}_j^{\mathbf{y}}\right)\right] + J^{bias}, \end{aligned}$$

that can be re-written as:

$$\begin{aligned} J^{\mathbf{z}}(C_1\vec{\mathbf{x}} + C_2\vec{\mathbf{y}}) &= \sum_i w_i^{\mathbf{x}} a_i^{\mathbf{x}}(\vec{\mathbf{x}}) + \sum_j w_j^{\mathbf{y}} a_j^{\mathbf{y}}(\vec{\mathbf{y}}) + J^{bias}, \\ \text{where} \qquad w_i^{\mathbf{x}} &= \alpha C_1\left(\vec{\mathbf{e}}^{\mathbf{z}} \otimes \vec{\mathbf{d}}_i^{\mathbf{x}}\right), \text{ and } w_j^{\mathbf{y}} = \alpha C_2\left(\vec{\mathbf{e}}^{\mathbf{z}} \otimes \vec{\mathbf{d}}_j^{\mathbf{y}}\right), \end{aligned} \tag{2.44}$$

which demonstrate that in addition to being abstract concepts, the encoders and decoders are combined to construct the physical connection weight matrix between two populations of neurons.

2.5.5 HRR-SPA Operators Implemented as Neural Networks

While the NEF provides a basic set of tools that can be used to implement any mathematical function in a population of spiking neurons, the exact application of these tools varies depending on the complexity of the function to be implemented. This section demonstrates how the NEF is used to realize the various SPA operators (specifically, the operators defined by the HRR-SPA) as spiking neural networks, and discusses some of the nuances to the various approaches used.

2.5.5.1 Semantic Pointer Representation

As mentioned in Section 2.3.1, the randomly generated HRR-SPA semantic pointers have an expected vector magnitude of 1. This means that the concepts introduced in the NEF – namely the multi-dimensional encoders and decoders – can be used to construct a neural network capable of representing a semantic pointer. Briefly, constructing a neural network to represent an HRR-SPA semantic pointer involves creating a multidimensional neural ensemble[14] with randomly generated d-dimensional encoders, and d-dimensional decoders optimized to represent the vector values within the unit d-dimensional hypersphere.

However, an issue does arise with the large number of dimensions used in the semantic pointer representation for Spaun. Referring back to Equation (2.31), part of the derivation of the decoders requires calculating the inverse of the $\mathbf{\Gamma}$ matrix. Since the $\mathbf{\Gamma}$ matrix is an $n \times n$ matrix (where n is the number of neurons in the population), it scales quadratically with a linear increase in the number of neurons. While this is not an issue with a small number of neurons, in Spaun, the neural ensembles typically contain $30 \times 512 = 15360$ each.[15] For such a neural ensemble, the decoder calculation would require the computation of the inverse of a matrix with $235,929,600$ elements.

Because of the computation cost of calculating the inverses of these huge matrices, the multidimensional ensembles in Spaun are implemented in a different manner. Instead of having one ensemble represent all of the dimensions in the semantic pointer, multiple neural ensembles are used to represent subsections of the semantic pointer. In particular, Spaun uses d one-dimensional ensembles to represent the d-dimensional semantic pointers. These ensembles are arranged in a collection referred to as an "ensemble array". In order to maximize the representational accuracy of the ensemble array, each ensemble in the ensemble array is optimized to represent a scalar value in the range of $\pm 3.5/\sqrt{d}$.[16] Figure 2.18 illustrates the difference between the multidimensional ensemble and the equivalent ensemble array.

[14]A neural ensemble is a population of neurons that has been constructed with the principles of the NEF to compute a specific function. The term "ensemble" and "population" are used interchangeably in this document.

[15]The number of neurons that would be used in a single multidimensional ensemble in Spaun is at minimum $m \times d$, where m is the number of neurons per dimension, and d is the number of dimensions in the semantic pointer representation. Typically, $m = 30$ and $d = 512$.

[16]From Section 2.3.1, each element in an HRR-SPA semantic pointer is chosen from a normal distribution with a mean of 0 and a variance of $1/d$. From experimental data, for dimension sizes larger than 15, in order to ensure that the neural ensembles are able to represent at least 99.9% of the possible values an element in the semantic pointer can take, the ensembles are optimized to represent scalar values in the range of $\pm 3.5\sigma = \pm 3.5/\sqrt{d}$. For dimension sizes less than or equal to 15, ensembles are optimized to represent the scalar values in the range of ± 1.

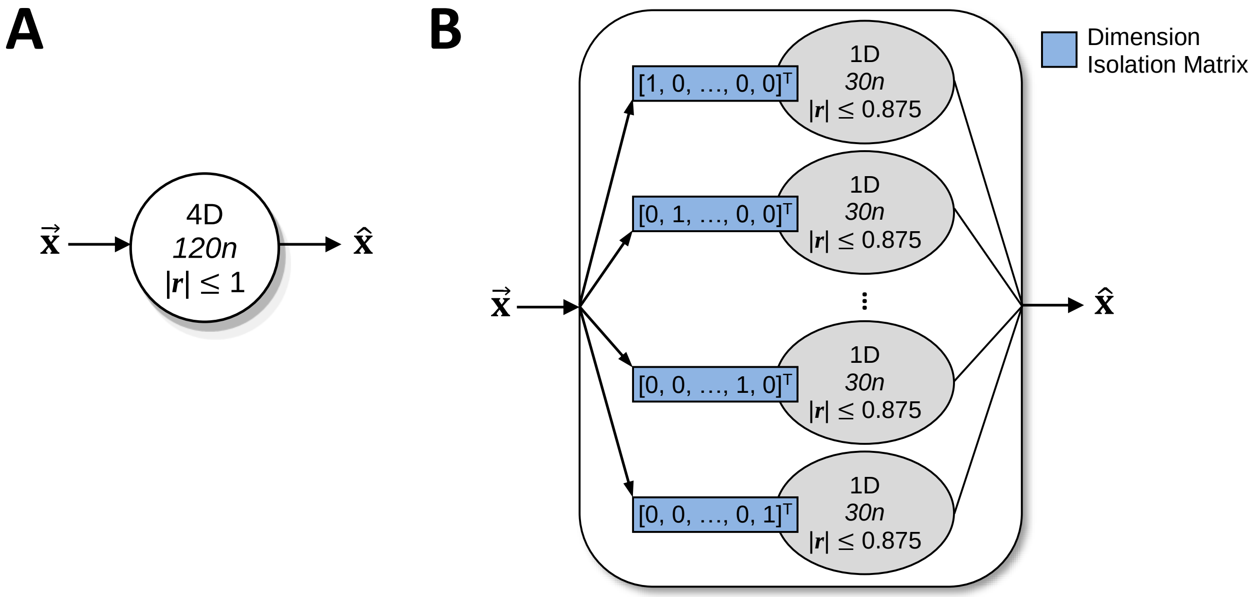


Figure 2.18: Connection diagrams illustrating the differences between a multidimensional ensemble and the equivalent ensemble array. (**A**) A single sixteen-dimensional neural ensemble with 480 neurons. The ensemble has a representational range of the 16D unit hypersphere ($|r| \leq 1$). (**B**) A sixteen-dimensional ensemble array comprised of 16 one-dimensional ensembles. Each ensemble in the ensemble array consists of 30 neurons and has a representational range of $\pm 3.5/\sqrt{16}$ ($|r| \leq 0.875$). Also illustrated are the appropriate transformation matrices (dimensional isolation matrices) necessary to achieve the correct representation for each ensemble.

2.5.5.2 SPA Collections Operator

Section 2.3.3 defined the collection operator as the element-wise vector sum of the input semantic pointers. A network like the one described in Figure 2.16 can be used to perform this operation. For such a network, the transformation weights $C_1 \ldots C_n = 1$.

2.5.5.3 SPA Binding Operator

There are multiple approaches to using the NEF to implement the SPA binding operator in a neural network. The naïve approach is to utilize a single neural ensemble and optimize the decoders to compute the circular convolution operation. However, this approach runs into the same computational complexity problems encountered with using a single neural ensemble to represent multidimensional vector values. For this reason, a different approach is needed to efficiently implement the binding operator in a neural network. Here, the approach is to decompose the binding operator into easy-to-construct base parts which are then later combined to achieve the desired circular convolution operation.

From Equation (2.17), the binding operation can be computed using the DFT, inverse DFT, and element-wise multiplication operators. Both the DFT and inverse DFT of a vector can be computed as a linear transformation, by performing the matrix multiplication between the vector and a DFT (or inverse DFT) matrix. Both the DFT and inverse DFT matrices are $d \times d$ in size, with the DFT matrix ($\mathbf{W}$) defined as:

$$\mathbf{W} = \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,d-1} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,d-1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{d-1,0} & w_{d-1,1} & \cdots & w_{d-1,d-1} \end{bmatrix}, \text{ where } w_{j,k} = e^{-\frac{(2\pi i)}{d} jk}, \tag{2.45}$$

And likewise, the inverse DFT matrix $\mathbf{V}$ is defined as:

$$\mathbf{V} = \begin{bmatrix} v_{0,0} & v_{0,1} & \cdots & v_{0,d-1} \\ v_{1,0} & v_{1,1} & \cdots & v_{1,d-1} \\ \vdots & \vdots & \ddots & \vdots \\ v_{d-1,0} & v_{d-1,1} & \cdots & v_{d-1,d-1} \end{bmatrix}, \text{ where } v_{j,k} = \frac{1}{n} e^{\frac{(2\pi i)}{d} jk} \tag{2.46}$$

Because neural ensembles can only represent physical (non-imaginary) values, the real and imaginary components of the DFT operation must be considered separately. In order to do this, the DFT and inverse DFT matrices are modified slightly:

$$\mathbf{W} = \mathbf{W}^{\mathbb{R}} + \mathbf{W}^{\mathbb{I}} i, \text{ where } w^{\mathbb{R}}_{j,k} = \cos\left(-\frac{2\pi}{n} jk\right) \text{ and } w^{\mathbb{I}}_{j,k} = \sin\left(-\frac{2\pi}{n} jk\right), \tag{2.47}$$

and,

$$\mathbf{V} = \mathbf{V}^{\mathbb{R}} + \mathbf{V}^{\mathbb{I}} i, \text{ where } v^{\mathbb{R}}_{j,k} = \frac{1}{n} \cos\left(\frac{2\pi}{n} jk\right) \text{ and } v^{\mathbb{I}}_{j,k} = \frac{1}{n} \sin\left(\frac{2\pi}{n} jk\right) \tag{2.48}$$

Note that in the equation above, the superscript $\mathbb{R}$ denotes the real-valued component of the matrices, while the superscript $\mathbb{I}$ denotes the imaginary component of the matrices.

With the DFT and inverse DFT matrices, the HRR-SPA binding operation can be written as:

$$\begin{aligned}
&\mathbf{A} \circledast \mathbf{B} = \mathcal{F}^{-1}(\mathcal{F}(\mathbf{A}) \odot \mathcal{F}(\mathbf{B})) \\
&= \mathcal{F}^{-1}(\mathbf{A}(\mathbf{W}^{\mathbb{R}} + \mathbf{W}^{\mathbb{I}} i) \odot \mathbf{B}(\mathbf{W}^{\mathbb{R}} + \mathbf{W}^{\mathbb{I}} i)) && (2.49) \\
&= \mathcal{F}^{-1}(\mathbf{AW}^{\mathbb{R}} \odot \mathbf{BW}^{\mathbb{R}} + \mathbf{AW}^{\mathbb{R}} \odot \mathbf{BW}^{\mathbb{I}} i + \mathbf{AW}^{\mathbb{I}} i \odot \mathbf{BW}^{\mathbb{R}} + \mathbf{AW}^{\mathbb{I}} i \odot \mathbf{BW}^{\mathbb{I}} i) && (2.50) \\
&= (\mathbf{AW}^{\mathbb{R}} \odot \mathbf{BW}^{\mathbb{R}} + \mathbf{AW}^{\mathbb{R}} \odot \mathbf{BW}^{\mathbb{I}} i + \mathbf{AW}^{\mathbb{I}} i \odot \mathbf{BW}^{\mathbb{R}} - \mathbf{AW}^{\mathbb{I}} \odot \mathbf{BW}^{\mathbb{I}})(\mathbf{V}^{\mathbb{R}} + \mathbf{V}^{\mathbb{I}} i) && (2.51) \\
&= (\mathbf{AW}^{\mathbb{R}} \odot \mathbf{BW}^{\mathbb{R}})\mathbf{V}^{\mathbb{R}} + (\mathbf{AW}^{\mathbb{R}} \odot \mathbf{BW}^{\mathbb{I}})\mathbf{V}^{\mathbb{R}} i + (\mathbf{AW}^{\mathbb{I}} \odot \mathbf{BW}^{\mathbb{R}})\mathbf{V}^{\mathbb{R}} i - (\mathbf{AW}^{\mathbb{I}} \odot \mathbf{BW}^{\mathbb{I}})\mathbf{V}^{\mathbb{R}} + \\
&\quad (\mathbf{AW}^{\mathbb{R}} \odot \mathbf{BW}^{\mathbb{R}})\mathbf{V}^{\mathbb{I}} i - (\mathbf{AW}^{\mathbb{R}} \odot \mathbf{BW}^{\mathbb{I}})\mathbf{V}^{\mathbb{I}} - (\mathbf{AW}^{\mathbb{I}} \odot \mathbf{BW}^{\mathbb{R}})\mathbf{V}^{\mathbb{I}} - (\mathbf{AW}^{\mathbb{I}} \odot \mathbf{BW}^{\mathbb{I}})\mathbf{V}^{\mathbb{I}} i \\
& && (2.52)
\end{aligned}$$

Note that in the equations above, the property that $i \times i = -1$ has been used to simplify the equations. In addition, because the circular convolution operation involving two real-valued vectors always produces a real-valued result, any term in the equation producing an imaginary value can be removed. The final equation for the circular convolution operation after this simplification is then:

$$\mathbf{A} \circledast \mathbf{B} = (\mathbf{AW}^{\mathbb{R}} \odot \mathbf{BW}^{\mathbb{R}})\mathbf{V}^{\mathbb{R}} - (\mathbf{AW}^{\mathbb{R}} \odot \mathbf{BW}^{\mathbb{I}})\mathbf{V}^{\mathbb{I}} - (\mathbf{AW}^{\mathbb{I}} \odot \mathbf{BW}^{\mathbb{R}})\mathbf{V}^{\mathbb{I}} - (\mathbf{AW}^{\mathbb{I}} \odot \mathbf{BW}^{\mathbb{I}})\mathbf{V}^{\mathbb{R}} \quad (2.53)$$

In this rewritten form of the circular convolution operation, it can be seen that performing the SPA binding operation requires 4 element-wise multiplication operations, 8 DFT transformation matrices, and 4 inverse DFT transformation matrices. The element-wise multiplication operation can be constructed as an ensemble array (see Figure 2.18) of multiplication operations (see Equations (2.42) and (2.43)). Combining this information with the methods described in the previous sections, the circular convolution neural network can be constructed as shown in Figure 2.19. As all of the frequency components of the DFT operation have the same variance (see Section 2.3.1), each of the ensembles in the ensemble arrays can be optimized to work with the same range of values, and no additional scaling mechanisms are needed, simplifying the network construction.

It is important to note that while the binding network seems complex, the final network is, ultimately, a collection of neurons and connection weights, indistinguishable from networks that compute any other function. In addition, the implementation described above is only one of many possible neural implementations of the HRR-SPA binding operation. Bekolay (2011) even demonstrate that a single-layer spiking neural network can be trained to compute the HRR-SPA binding operation.[17]

[17]Though possible, the binding networks used in Spaun are not learned using the methods outlined in Bekolay (2011). This is due to the long training times required and the desire to reduce Spaun's already lengthy construction times.

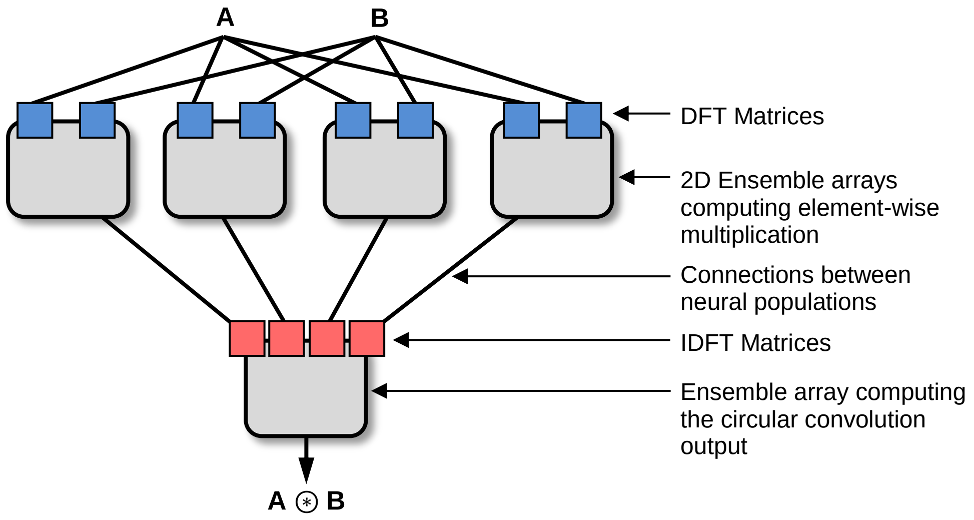

A
B
DFT Matrices
2D Ensemble arrays computing element-wise multiplication
Connections between neural populations
IDFT Matrices
Ensemble array computing the circular convolution output
A ⊛ B

Figure 2.19: An illustration of the circular convolution operation $\mathbf{A} \circledast \mathbf{B}$ implemented using a network of neural populations. The network is comprised of 5 ensemble arrays, four of them to compute the element-wise multiplication operations, and remaining one to perform the inverse DFT operation and reconstitute result of the element-wise operations back into a single d-dimensional vector. The small blue and red squares represent the transformation matrices required to perform the DFT and inverse DFT operations, respectively, while the large grey rounded-squares represent the ensemble arrays. (Figure adapted from [Choo, 2010] with permission.)

2.5.5.4 Semantic Pointer (Approximate) Inverse

Equation (2.21) defines the HRR-SPA semantic pointer (approximate) inverse as reversing the order of the vector elements for all except the first vector elements. Since this is a linear transformation, the inverse operator can be implemented using the appropriate ($\mathbf{C}$) transformation matrix in Equation (2.41). For the inverse operator, the transformation matrix ($\mathbf{L}$) is an $n \times n$ matrix with 1 in the top left corner and a reverse diagonal of 1's in the bottom right corner (see

Appendix A.2). The semantic pointer inverse matrix for an n dimensional HRR-SPA vector is:

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 \\ 0 & 0 & 0 & \dots & 1 & 0 \\ \vdots & \vdots & \vdots & .\cdot^{\cdot} & \vdots & \vdots \\ 0 & 0 & 1 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \end{bmatrix} \tag{2.54}$$

It should be noted that because the inverse operation is a linear transformation, the neural network for the SPA unbinding operation is identical to the neural network used for the binding operation, with the exception that the input vector to be inverted has a transformation matrix that is a combination of the DFT transformation and the inverse transformation. Figure 2.20 illustrates an inverse network, and an unbinding network.

2.5.5.5 Semantic Pointer Cleanup

The semantic pointer cleanup algorithm described in Section 2.3.7 can be adapted for use with the NEF to create a neural implementation [Stewart et al., 2011]. As a recap of the cleanup algorithm, the cleanup operation on the semantic pointer **A** is computed by using the following steps:

1. Calculate the similarity measure of **A** against all of the semantic pointers in the vocabulary.
2. From the calculated similarity measures, identify the largest one.
3. Identify the semantic pointer (from the vocabulary) that is paired with the largest similarity measure. This is the result of the cleanup operation.

As the methods of the NEF work seamlessly with linear matrix operations, the cleanup operation implementation can be simplified by taking advantage of the fact that the matrix multiplication of a vector with a matrix simultaneously computes the dot product of the vector with each column of the matrix. In essence, the matrix M can be constructed by vertically stacking the semantic pointers in the vocabulary into a matrix:

$$M = \begin{bmatrix} \mathbf{SP1} \\ \mathbf{SP2} \\ \vdots \\ \mathbf{SPN} \end{bmatrix} = \begin{bmatrix} sp1_0 & sp1_1 & sp1_2 & \dots & sp1_{d-1} \\ sp2_0 & sp2_1 & sp2_2 & \dots & sp2_{d-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ spN_0 & spN_1 & spN_2 & \dots & spN_{d-1} \end{bmatrix},$$

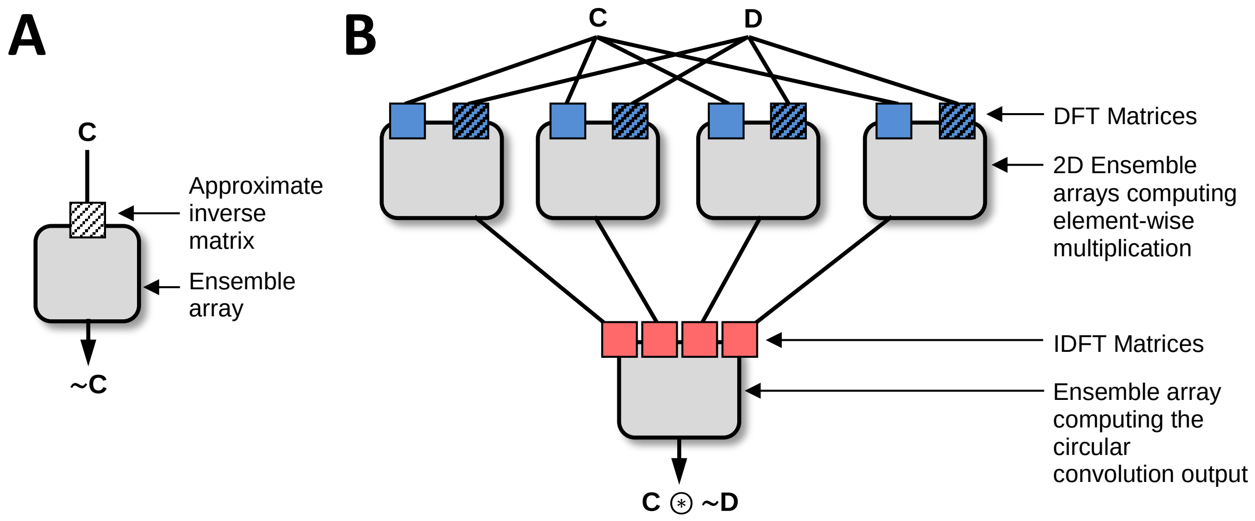

A
C
Approximate inverse matrix
Ensemble array
~C
B
C
D
DFT Matrices
2D Ensemble arrays computing element-wise multiplication
IDFT Matrices
Ensemble array computing the circular convolution output
C ⊛ ~D

Figure 2.20: Network diagrams for the HRR-SPA inverse operation and the HRR-SPA unbinding operation. (**A**) Neural network to compute the HRR-SPA (approximate) inverse operation ~**C**. The network comprises of one ensemble array, and the appropriate inverse matrix (**L**) to compute the inverse (see accompanying text for details). In the network diagram, the inverse transformation matrix is represented by the white shaded square. (**B**) Neural network to compute the HRR-SPA unbinding operation **C** ⊛ ~**D**. The small blue and red squares represent the transformation matrices required to perform the DFT and inverse DFT operations, respectively. The blue shaded squares represent the transformation matrices that are the result of combining the HRR-SPA inverse matrix with the DFT matrix (i.e., **LW**).

where **SP1**, **SP2**, etc., are the semantic pointers that make up the cleanup vocabulary, and $sp1_0, sp1_1$, etc., are the vector elements of each semantic pointer. With the M matrix, the three step cleanup operation can be re-written as the following steps[Stewart et al., 2011]:

1. Use the matrix multiplication $\vec{A}M^T$ to simultaneously compute the dot product of **A** with all of the semantic pointer vectors in the vocabulary (stacked together to form the matrix M). This operation results in a vector of similarity values ($\vec{S}$).
2. Identify the largest dot product value in $\vec{S}$ and set that to 1. All other elements are set to 0.
3. Perform the matrix multiplication $\vec{S}M$ to compute the semantic pointer vector result of the cleanup operation.

The core of the cleanup algorithm is the generation and manipulation of the N-dimensional similarity vector (where N is the number of semantic pointers in the cleanup vocabulary). Since the operations performed on each of the elements in the similarity vector are independent of each other, using the NEF, an N-dimensional ensemble array can be used to represent the similarity vector. Both the generation of the similarity vector ($\vec{A}M^T$) and the generation of the "cleaned" result ($\vec{S}M$) are matrix operations, and as demonstrated previously, can be implemented as transformation matrices on the input and output connections to the ensemble array, respectively. This leaves step 2 as the last part of the algorithm to be neurally implemented.

Step 2 of the algorithm involves isolating the maximum similarity value – also known as a "winner-take-all" (WTA) operation – and setting that value to 1. To implement the WTA mechanism in a neural network, two changes must be made to the standard randomly-generated neural ensembles of the NEF. First, the encoders for each of the ensembles in the ensemble array are aligned along a common direction. Since each ensemble in the ensemble array represents a scalar value, all of the encoders are aligned towards the positive x direction. Additionally, the gains (α) and bias currents (J^{bias}) for each neuron (see Equation (2.24)) are specifically chosen such that neuron response curves are only active beyond a preset x value threshold. Second, negatively weighted recurrent connections are added such that the output of each ensemble in the ensemble array inhibits all of the other ensembles in the ensemble array. With these negative recurrent connections, any active ensemble in the ensemble array suppresses all other ensembles in the ensemble array, thus computing a WTA function.[18]

Lastly, setting the output of the WTA result to 1 can be achieved by projecting it to another ensemble array with thresholded ensembles where the decoders of these ensembles have been optimized such that they output a fixed value (of 1) whenever the neural population is active (i.e., the neural ensembles are computing the function $f(x) = 1$). Figure 2.21 illustrates the complete HRR-SPA cleanup neural network.

It should be noted that in the description above, the vocabulary and outputs of the cleanup memory network are matching semantic pointers. This is known as an auto-associative memory, as an input that is similar to the semantic pointer **A** is cleaned-up (associated) to the semantic pointer **A**. By altering the output vocabulary matrix (M), the cleanup memory is converted from an auto-associative memory to a "regular", or hetero-associative memory. This is particularly helpful in the scenarios where a non-matching mapping between the input and output vocabularies is required by the particular SPA operation – as an example, converting between different semantic pointer conceptual spaces of differing dimensional sizes.

[18]It should be noted that another approach to computing the WTA function is to have a single N-dimensional ensemble, and solving the decoders such that they compute the WTA function. But, this approach is often less "clean" and introduces noise (multiple output semantic pointers) to the resulting output. However, this approach is more amenable if the neural ensemble is required to learn or modify the WTA function in any way – either by having to learn the WTA function from scratch, or if additional items are added to the cleanup vocabulary.

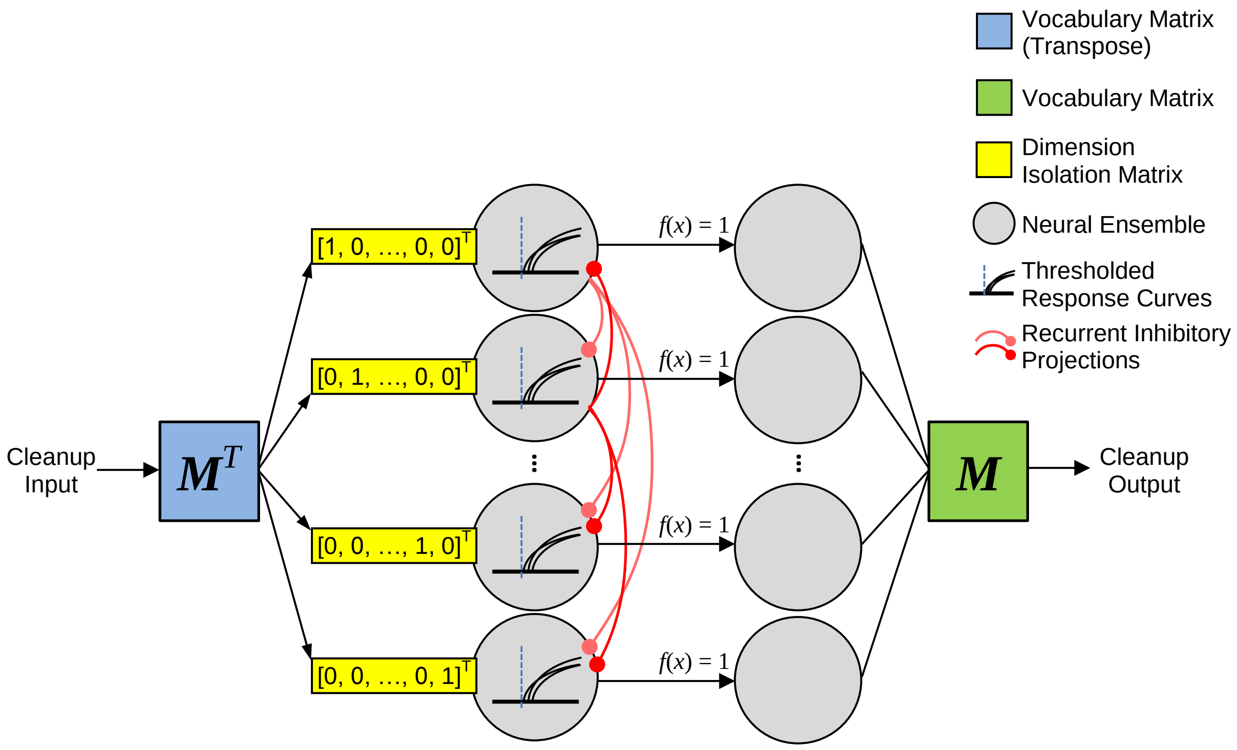


Figure 2.21: The neural network used to compute the HRR-SPA cleanup operation. To compute the similarity values, the input vector is multiplied with the transpose of the cleanup vocabulary matrix M. Next, recurrent inhibitory connections are used to perform the WTA operation on the similarity values, which have been thresholded by appropriately adjusting the response curves of each neural ensemble. The result of the WTA operation is projected onto another population that calculates the function $f(x) = 1$ for any population that is active as a result of the WTA operation, converting the similarity vector into a binary (either 0 or 1) vector. Finally, the binary similarity vector is multiplied with the cleanup vocabulary matrix to generate the final cleaned semantic pointer output. To reduce clutter, the recurrent inhibitory connections have only been shown for the first two ensembles. In reality, each ensemble has an inhibitory projection onto all other ensembles.

2.5.6 SPA Subsystem Components Implemented as Neural Networks

Section 2.4 introduced the major components of the SPA subsystem as the mechanisms for semantic pointer compression, semantic pointer transformation, and an action selection system. As the neural implementations for the components of the transformation mechanisms were discussed in the previous section, this section will explore the neural implementations of the compression and action selection mechanisms.

2.5.6.1 Semantic Pointer Compression – Memory

Mathematically, an integrator is the simplest form of memory, because it accumulates input over time. In addition, an integrator "remembers" the value that has been accumulated when the input to the integrator is removed. In the NEF discussion thus far, two of the three principles of the NEF (representation and transformation) have been discussed. To construct an neural integrator, the third principle of the NEF is used. Detailed in Chapter 8 of Eliasmith and Anderson (2003), the third principle of the NEF states that any dynamical system of the form

$$\dot{x}(t) = Ax(t) + Bu(t) \tag{2.55}$$

can be understood in block diagram network form as shown in Figure 2.22A. Converting the block diagram form into a neural implementation requires replacing the integrator (central box) with the transfer function of the neural synapse and compensating for this change in the A and B matrices, with the new matrices referred to as A' and B' respectively (see Figure 2.22B).

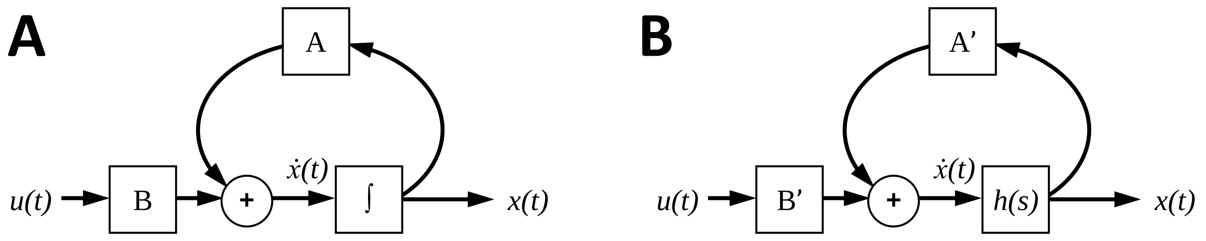


Figure 2.22: Block diagrams of a generic dynamical system, and the equivalent neural implementation. (**A**) Block diagram representation for the dynamical system $\dot{x}(t) = Ax(t) + Bu(t)$. (**B**) Neural implementation equivalent of the previous block diagram. The integrator is replaced with the transfer function of the neural synapse ($h(s)$), and the A and B matrices are replaced by A' and B', which compensate for the replacement of the integrator. The exact values of A' and B' depend on the dynamics of the synapse (i.e., $h(s)$). (Figure adapted from [Eliasmith and Anderson, 2003] with permission.)

For the exponential synapses used in the Spaun model (see Figure 2.15), the weights $\mathbf{A}' = \tau\mathbf{A}+\mathbf{I}$ and $\mathbf{B}' = \tau\mathbf{B}$, where τ is the time constant of the exponential synapse used. The dynamical systems equation for an integrator is

$$\dot{x}(t) = u(t) \tag{2.56}$$

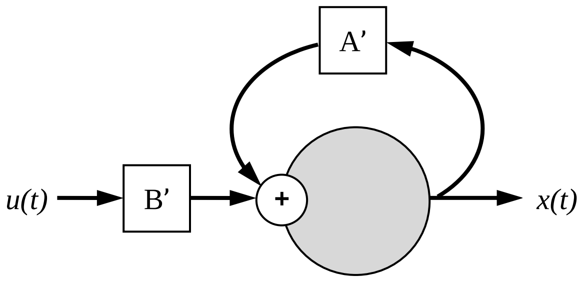


Figure 2.23: Simple schematic of how an integrator is implemented with a population of neurons. The grey circle represents a population of neurons and the square boxes represents the constant weights A' and B' applied to each signal. For an "ideal" integrator, $A' = 1$ and $B' = \tau^{PSC}$. (Figure adapted from [Choo, 2010] with permission.)

which results in a neural network where $\mathbf{A'} = 1$ and $\mathbf{B'} = \tau^{PSC}$ (see Figure 2.23).

More complex (controlled) integrators and memory systems are constructed using the same underlying concept with additional circuitry used to control the flow of information into the integrator or within the integrator via the feedback loop (see Section 2.5.6.4 for constructs to control the flow of information, and Section 3.3.8.3 for examples of more complex memory systems).

2.5.6.2 Semantic Pointer Compression – Compressive Hierarchy

The methods of the NEF are not used to directly construct hierarchical neural networks used to perform tasks like sensory input classification. This is because while the NEF can be used to approximate the complex functions necessary for these hierarchies, the machine learning community has developed more effective methods for generating these hierarchies. Rather, to construct hierarchical neural networks, the machine learning community initializes the networks with randomized connection matrices and a learning algorithm is applied to modify the weights to perform the appropriate task.

However, once a hierarchical model has been trained using the various machine learning techniques, converting them into spiking neuron models using the NEF is straightforward. The connection weights defined in the hierarchical model is used as the transformation matrix C (see Equation 2.41), and an ensemble of neurons is used to approximate the non-linearity (e.g., logistic, *tanh*, etc.) used in each node of the original hierarchical model. Figure 2.24 illustrates an example hierarchical 3-layer model using sigmoid "neurons", and the equivalent NEF hierarchical 3-layer spiking neural network.

It should be noted that the method for constructing a spiking compressive hierarchy presented above is a naïve approach using only the NEF methods of function approximation discussed so

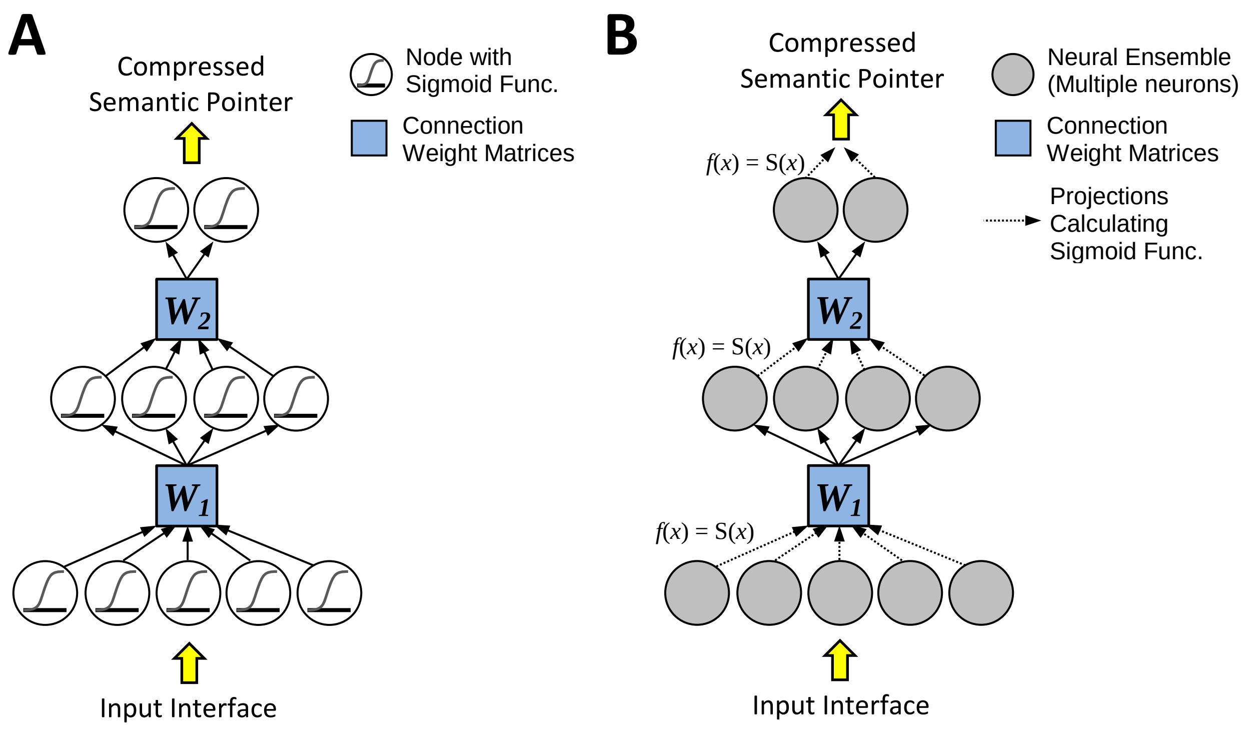

A
Compressed Semantic Pointer
Node with Sigmoid Func.
Connection Weight Matrices
W_2
W_1
Input Interface
B
Compressed Semantic Pointer
Neural Ensemble (Multiple neurons)
Connection Weight Matrices
Projections Calculating Sigmoid Func.
$f(x) = S(x)$
$f(x) = S(x)$
$f(x) = S(x)$
W_2
W_1
Input Interface

Figure 2.24: Comparison of a generic machine-learned hierarchical network and the equivalent NEF hierarchical network. (**A**) Schematic of a generic 3-layer hierarchical network. Each layer consists of nodes that compute a sigmoid function non-linearity. The output of each layer is projected onto a weight matrix (**W**), the result of which is used as the input to the next network layer. (**B**) Schematic of the hierarchical network in (**A**) converted into a spiking neural network using the NEF. Each node in the former network is replaced with a neural ensemble with enough neurons to approximate the original sigmoid non-linearity. The sigmoid function itself is computed though the network projections between layers (dotted lines). The weight matrices in both networks are identical.

far. Section 4.1.1 explores a more elegant method of directly constructing spiking hierarchical neural networks.

2.5.6.3 Information Flow Control – Action Selection

As discussed in a Section 2.4, the action selection component of the SPA subsystem monitors the current state of the system, and produces the appropriate control signals (actions) necessary to control the flow of information in the model. Since the model in question is an SPA model, semantic pointers are used to represent the various states the system can occupy. For the action selection component to decide which action to perform, it needs to compare the various semantic pointers representing the current state (the "system state") to a bank of predetermined state semantic pointers (the "condition set"), each of which are associated with a specific action (the "consequence set").

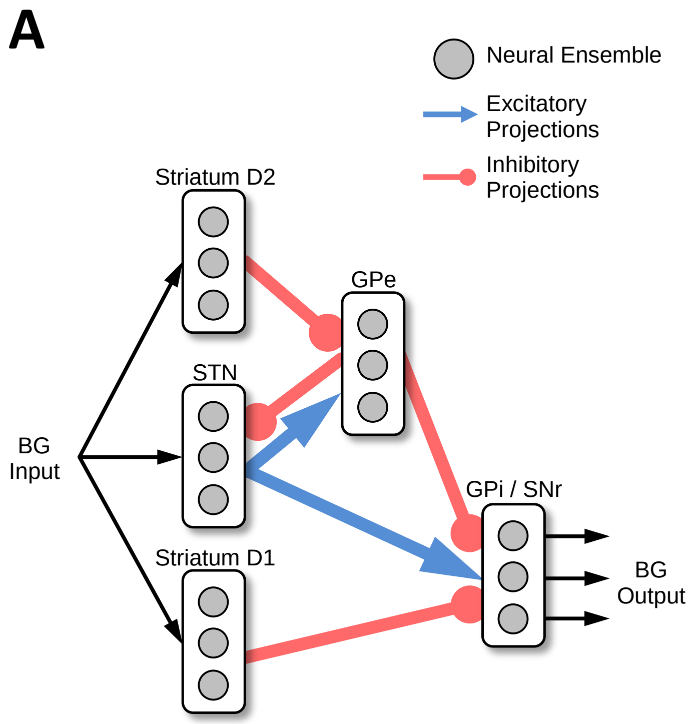
The state semantic pointer comparison operation can be implemented using the cleanup memory network discussed previously. In the context of building cognitive models, however, they are not used because for semantic pointers with close similarities, the cleanup memory network can output the wrong result, especially during the transition from one semantic pointer input to another.[19] Additionally, because the action selection component is a critical part of a model's decision making system, in order to replicate behavioural effects (e.g., spiking activity patterns, reaction times, etc.), it is necessary to take inspiration from biology for this part of the SPA subsystem.

Both neuroscientists and cognitive scientists generally believe that the basal ganglia (BG) is part of the brain's action selection system (e.g., [Kropotov and Etlinger, 1999; Redgrave et al., 1999]). In 2001, Gurney, Prescott and Redgrave developed a rate-based basal ganglia model constrained by the known neuron types and connectivity of the biological basal ganglia. Using the principles of the NEF, this model has been adapted for use as the core of the action selection component of the SPA [Stewart et al., 2010]. In terms of operation within the larger SPA model, semantic pointers of the system state are projected onto the input ensembles (the striatum D1 and D2 populations, and the subthalamic nucleus) of the basal ganglia through transformations similar to the cleanup memory network. The transformations convert the system state semantic pointer into a utility value for each condition in the condition set. As a consequence of the way the inhibitory and excitatory connections are set up in the basal ganglia network, a winner-take-all-like operation is performed on these utility values, and an inhibitory output signal is produced.[20] The inhibitory output of the basal ganglia is then inverted through a projection onto a neural ensemble. Figure 2.25 illustrates the structure and operation of a 3-condition set NEF basal ganglia model.

Section 3.3.8 provides a more detailed description of how the basal ganglia model is integrated with various working memory components and the other information flow components to form the complete action selection system.

[19]While this drawback can be mitigated in other parts of the network through the use of appropriate information flow control, the same mitigation methods cannot be applied to the action selection component (as it is an integral part of the flow control system).

[20]The basal ganglia produces inhibitory outputs, where the output with no activity is the "chosen" action.



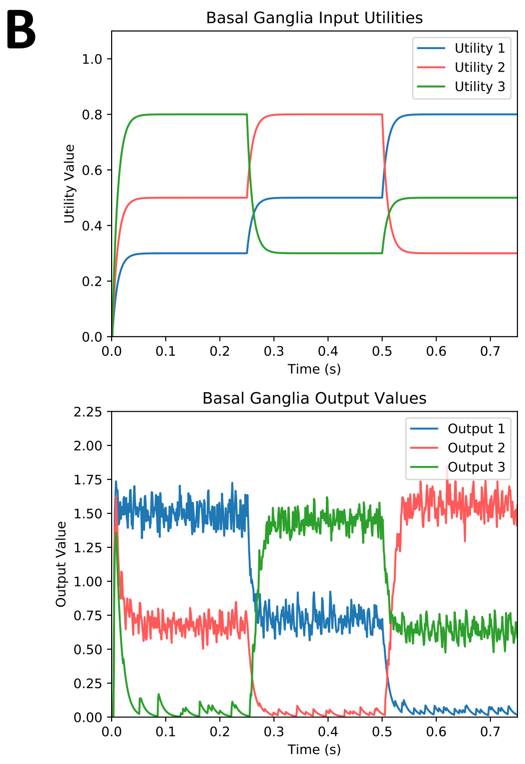


Figure 2.25: Schematic of the basal ganglia (BG) network, and plots of an example operation of the network. (**A**) Schematic of a 3-condition set basal ganglia network used in the majority of NEF models. Shown are the excitatory and inhibitory projections between the neural ensembles. Note that the projections shown are generic projections between the populations representing the different brain areas in the basal ganglia, not individual projections between each neural ensemble. (**B**) Decoded value plots of a 3-condition set NEF basal ganglia network. The top plot shows the input utility values (values fed to BG input) starting from [0.3, 0.5, 0.8], and transitioning to [0.5, 0.8, 0.3], and lastly to [0.8, 0.3, 0.5]. The bottom plot shows the output of the basal ganglia, showing it selecting "Output 3", then "Output 2", and finally "Output 1", demonstrating the network's ability to perform a WTA-like operation. Note that output of the basal ganglia is inhibitory, meaning that the "chosen" output has the lowest (almost 0) decoded value.

2.5.6.4 Information Flow Control – Gating

As described in Section 2.4.3, the goal of the gating component is to restrict the flow of information to parts of the subsystem when given the appropriate control signal from the action selection component. As neurons spike only when there is sufficient input current, mathematically, the gating behaviour can be achieved by introducing enough negative current into the neuron such that the neuron does not activate. Biologically, restricting the neuron from firing through the introduction of a control signal is known as inhibition. The negative inhibitory current can be achieved through the influx of negatively charged ions into the neuron (inhibitory post-synaptic potentials), or by opening channels in the dendrites that cause the current to flow out of the cell (shunting inhibition).

In the NEF, gating inhibition is achieved by adding an inhibitory input to each neuron. On the population level however, the inhibitory control signal cannot simply be a negative x value, since the neural populations are typically optimized to operate with both positive and negative x values. In addition, for multidimensional ensembles, a "negative x value" cannot be appropriately defined. As such, the connection weights for the inhibitory connection is modified such that given a positive inhibitory signal, a negative current is added to the input current to the neuron. In terms of the NEF, the inhibitory input is represented just like any other input to the population, and the output of the population is represented as the linear combination of the original input x and the inhibitory input q (see Section 2.5.4.5).

The connection weight matrix between the input population $\mathbf{x}$ and the output population $\mathbf{z}$ is calculated as usual – that is to say, using the decoders and encoders to calculate the weight matrix. However, the connection weight matrix for the inhibitory input population, $\mathbf{q}$, is predefined such that every connection has a weight of -1. The input current to each neuron in the output population is then:

$$J^{\mathbf{z}}(C_1\vec{\mathbf{x}} + C_2 q) = \sum_i w_i^{\mathbf{x}} a_i^{\mathbf{x}}(\vec{\mathbf{x}}) + \sum_j w_j^{\mathbf{q}} a_j^{\mathbf{q}}(q) + J^{bias}, \tag{2.57}$$

where $w_i^{\mathbf{x}} = \alpha C_1 \left(\vec{\mathbf{e}}^{\mathbf{z}} \otimes \vec{\mathbf{d}}_i^{\mathbf{x}}\right)$, and $w_j^{\mathbf{q}} = \alpha C_2(-1)$. Note that the value of the weight C_2 can be set to any arbitrary value that is greater than the maximum optimal representational range of the neuronal population. It should be noted that because the inhibitory input connection weight matrix does not include encoders or decoders, Equation (2.57) is applicable to multidimensional ensembles with no modifications necessary. Figure 2.26 shows the effect of adding an inhibitory input to a population of neurons representing the identity function ($y = x$).

As is the common theme of the NEF, the method of gating information flow through inhibition is not the only method available to neural network modellers. Gating can also be accomplished

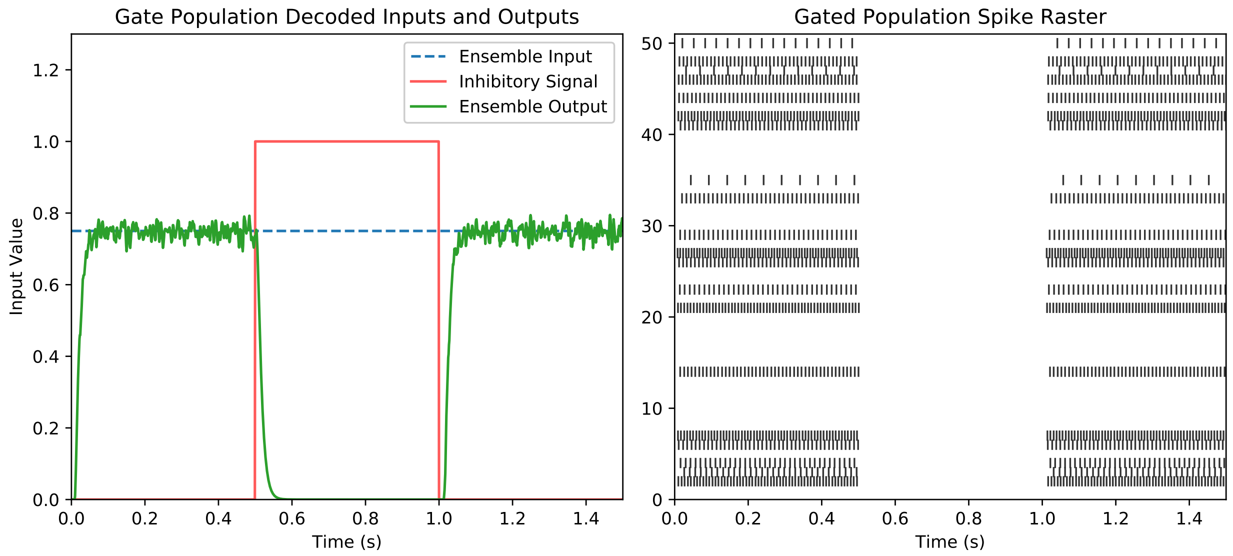


Figure 2.26: Illustration of the effect of inhibiting the output of a 50-neuron population. (Left) Plot of the decoded output of the neural population overlaid with the inhibitory input to the population, and the input signal to the population. When the inhibitory input is present, the output of the population drops to 0, and all the neurons in the population stop spiking (see right plot). (Right) Spike raster plot for each neuron in the population.

though the use of a multiplication or binding ensemble (through the multiplication with 0, or the binding with the "**∅**" semantic pointer), or through the use of non-linear multiplicative dendrites [Bobier, 2011].

2.5.6.5 Information Flow Control – Inhibited Biased Populations

The basal ganglia network described in the previous section poses a unique problem in terms of its outputs: the output values are inverted and meant to inhibit the efferent populations. Because of the inversion, a mechanism is needed to reverse the inversion before the signal is used in the rest of the system.

In the NEF, biased neural populations take on this role. Biased neural populations are created by adding a constant current input to the neurons such that when no input (i.e., $x(t) = 0$) is provided to the neural population, the output of the neural population is the desired bias value. As an example, if the desired bias value ($\mathbf{C}$) is 0.5, using equation (2.29) the input current to the neurons is calculated as:

$$J(\vec{\mathbf{x}}) = \alpha(\vec{\mathbf{e}} \bullet \vec{\mathbf{x}}) + J^{bias} + \alpha(\vec{\mathbf{e}} \bullet \mathbf{C})$$
$$J(\vec{\mathbf{x}}) = \alpha(\vec{\mathbf{e}} \bullet \vec{\mathbf{x}}) + J^{bias} + \alpha(\vec{\mathbf{e}} \bullet 0.5),$$

where $\alpha(\vec{\mathbf{e}} \bullet \mathbf{C})$ is the constant bias input current added to the neural population.

These biased neural populations can be used to invert the output of the basal ganglia by setting $\mathbf{C} = 1$, and by using the output of the basal ganglia as inhibitory inputs (see Section 2.5.6.4) to gate each biased ensemble (see Figure 2.27A). In essence, when the output of the basal ganglia is active (i.e., the action is *not* "chosen"), the biased population will be gated and the output will be 0. When the output of the basal ganglia is close to 0 (i.e., the action has been "chosen"), the biased population will stop being inhibited, allowing the predefined biased value (1) to be propagated to the rest of the system. Should another value be desired, the decoders of the biased population can be calculated to provide that value.

In most SPA models, the biased populations that invert the output of the basal ganglia are collected together in a network referred to as the thalamus, referencing the part of the biological brain that receives the majority of inhibitory outputs from the basal ganglia [Le Gros Clark, 1932]. Functionally, the thalamus is believed to play a crucial role in the routing of information between different areas of the brain (e.g., [Uno et al., 1970; Vives and Mogenson, 1985; Hwang et al., 2017]) which match the intended role for the thalamus in SPA models (see Section 3.1.4). Figure 2.27 illustrates the structure and operation of the thalamus for the 3-condition set NEF basal ganglia model from Figure 2.25.

2.5.7 Discussion

This section outlined the basic principles of the NEF, and provided examples of how they are used to implement the various HRR-SPA operators in neural networks. It should be noted that in the interest of brevity, this presentation of the NEF deliberately overlooked some of the finer details of the neural calculations; in particular, the discussion of how to compensate for the inherent noisiness of neurons (since they communicate in discreet spikes and not in real-valued signals) has been omitted. Details on the derivations of the NEF equations that compensate for this and other sources of noise can be found in [Eliasmith and Anderson, 2003].

2.5.7.1 Saturation: Side-effects of a Neural Implementation

Biologically, and looking at Equation 2.25, it can be seen that because neurons have a refractory period in which they cannot fire regardless of the magnitude of the input current, neurons have a

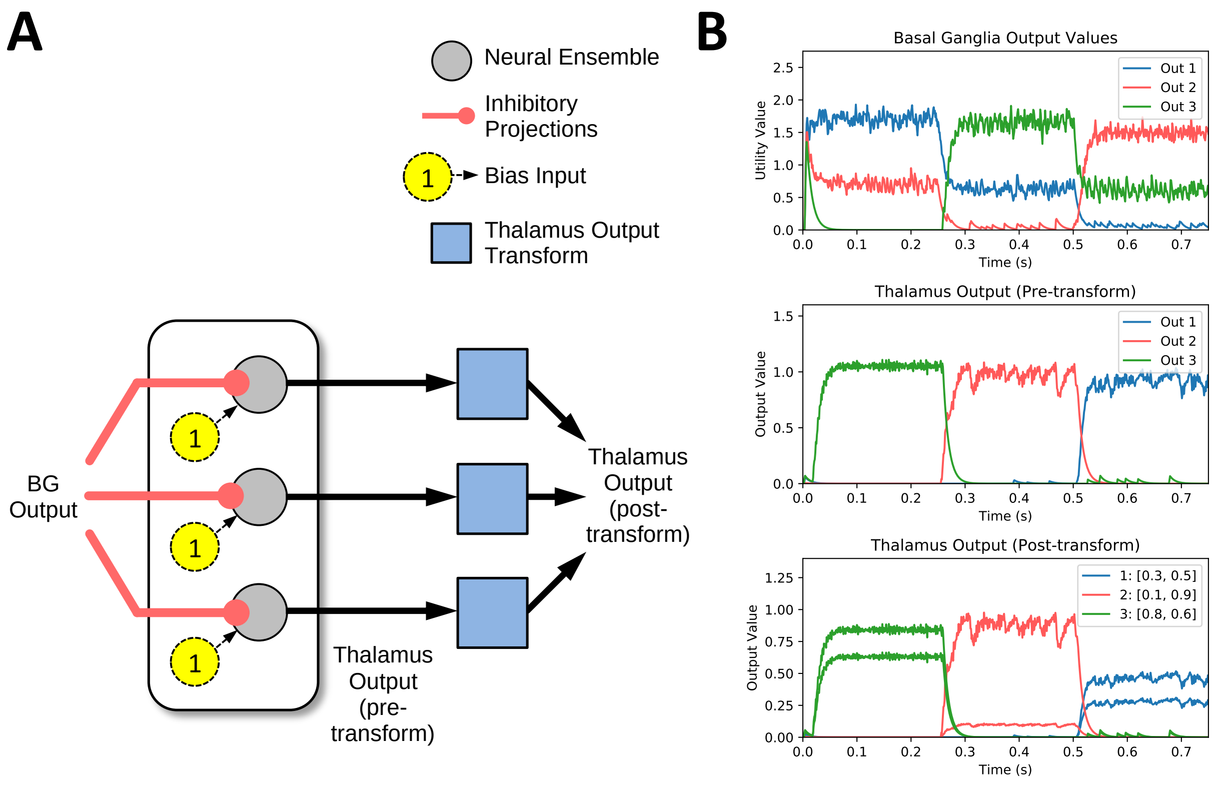


Figure 2.27: Schematic of the thalamus network, and plots of an example operation of the network. (**A**) Schematic of a 3-action set thalamus network. The inhibitory input projections are the outputs of a BG network. For the 3-action set thalamus network, a 3-condition set BG network is required, such as the one illustrated in Figure 2.25. The BG outputs inhibit the neural populations in the thalamus network using the mechanisms described in the previous section. (**B**) Decoded value plots of the 3-action set thalamus network. The top plot shows the output of the BG network that serves as the input to the thalamus network. The middle plot shows the output of the thalamus network before the output transformation is applied. Since the thalamus consists of neural populations with a bias value of 1, the output of the thalamus is 1 when the appropriate thalamus action is "chosen" (As in Figure 2.25, "Out(put) 3" is selected, followed by "Out 2", and lastly "Out 1"). The bottom plot shows the output of the thalamus network, after the output transform has been applied. While the output shown in the figure is only two dimensional, this is meant to demonstrate that the thalamus can be configured to output multidimensional semantic pointers to the rest of the system. The correct transform output values are shown in the legend.

fixed upper bound on how fast they can fire. In the NEF, this can be demonstrated by providing a neural ensemble with an input value outside the range in which the decoders have been optimized. This “saturation” effect is evident in both the scalar and multidimensional ensembles as well as in their equivalent ensemble arrays (see Figure 2.28). Because the saturation phenomenon limits the magnitude of the output vector without greatly affecting the proportional difference between each vector element, the effect of saturation is similar to the HRR-SPA normalization operation (see Section 2.3.6), and is sometimes referred to as a having a “soft-normalization” effect on the semantic pointers.

The saturation effect is also apparent in the ensemble arrays, however, the amount of saturation differs depending on how the ensemble array is configured. Because ensemble arrays represent subsets of the vector value independently, the effects of saturation depend on the dimensionality of each ensemble within the ensemble array (referred to as “sub-ensembles”). Figure 2.29 demonstrates the effect of saturation on a 512-dimensional ensemble array constructed with 1D, 8D, 16D, and 64D sub-ensembles.

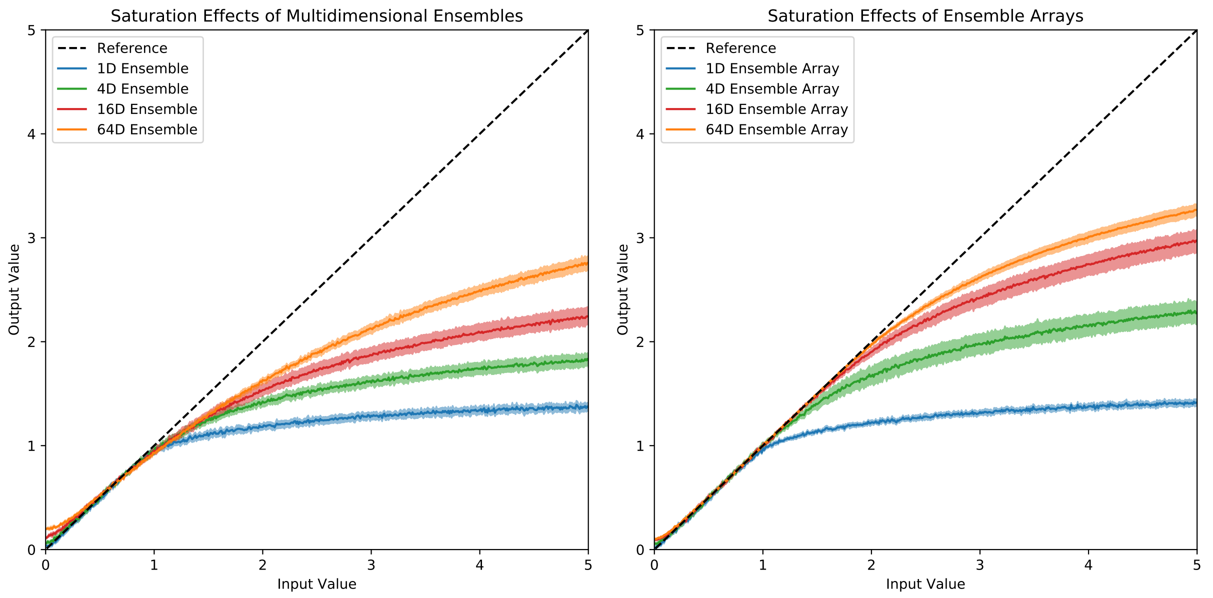


Figure 2.28: Demonstration of the neural saturation effects of multidimensional neural ensembles and ensemble arrays. For each plot, the neural ensembles / ensemble arrays are scaled linearly with the dimensionality, with the single-dimensional ensemble comprised of 30 neurons. Additionally, the 95% confidence ranges for 10 randomly generated trials are displayed. (Left) Plots of the input and output values for several multidimensional ensembles. Without the effects of saturation, the input and output values will match (reference line). As the plots demonstrate, the output of the ensemble saturates regardless of the dimensionality of the ensemble. However, the effect of the saturation reduces as the dimensionality increases. It is hypothesized that the increased number of neurons allows for larger vectors to be represented before saturating the ensemble. (Right) Saturation plots for equivalent ensemble arrays (composed of one-dimensional sub-ensembles). Each sub-ensemble in the ensemble array is optimized to the value ranges as per the discussion in Section 2.5.5.1. The ensemble arrays display similar saturation effects, however since each dimension can be represented independently, the saturation effect is less pronounced as the dimensionality increases (compared to the multidimensional ensemble counterpart).

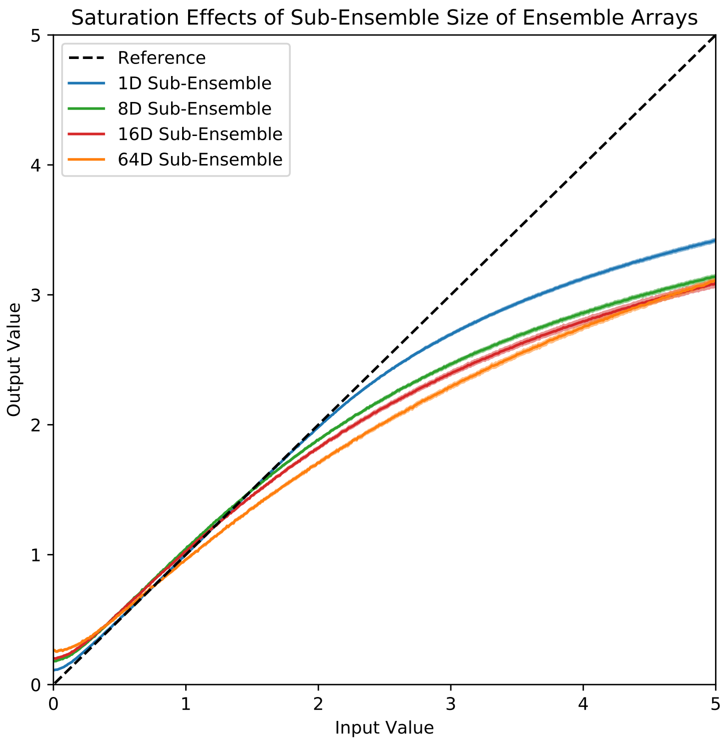


Figure 2.29: Demonstration of the neural saturation effects on a fixed dimensionality ensemble array configured with sub-ensembles of 1 dimension, 8 dimensions, 16 dimensions, and 64 dimensions. As the plots illustrate, the saturation effect is present regardless of the dimensionality of the sub-ensembles, with larger sub-ensemble dimensionality exhibiting more saturation.

Chapter 3

Spaun

Spaun [Eliasmith et al., 2012] is, the world's largest functional brain model, consisting of roughly 2.5 million neurons, and able to perform eight tasks of varying cognitive difficulties. To understand the tasks that Spaun was designed to perform, as well as the architecture of its design and the role the SPA plays in this architecture, it is helpful to review the history of Spaun's history.

Spaun was conceived as a proof-of-concept neural network, constructed using the methods described by the NEF and the SPA, and importantly, it had to be entirely self-contained, mimicking the biological condition whereby all inputs received by the system would be sensory information, and all outputs produced would be some form of motor action. Prior to the construction of Spaun, various models of different cognitive functions (e.g., visual processing, working memory, action planning) had been developed, and it was hypothesized that the NEF and the SPA could serve as the tools to integrate these various models together into a truly self-contained system.

This chapter briefly describes each of the "precursor" networks that served as the building blocks of Spaun, followed by the description of each of Spaun's eight task and how the SPA was used to accomplish each of these tasks, a description of the different modules of Spaun.

3.1 Precursor Networks

The initial iteration of Spaun integrated six existing stand-alone neural networks. Importantly, these networks were chosen because each contributed to a different aspect of the computation needed to perform the set of Spaun's tasks.[21] These six neural network models were:

[21]It should be noted that all of the models were also developed by various members of the Computational Neuroscience Research Group at the University of Waterloo, the same lab which developed Spaun.

- An MNIST classification vision neural network [Tang and Eliasmith, 2010]
- The NOCH motor control framework [DeWolf, 2010]
- The OSE serial working memory model [Choo, 2010]
- A basal ganglia (BG)-based symbolic reasoning system [Stewart et al., 2010]
- A general inductive reasoning neural network [Rasmussen, 2010]
- An adaptive spiking BG model [Stewart et al., 2012]

3.1.1 MNIST Visual Network

The MNIST (Modified National Institute of Standards and Technology) visual network is a sparse deep belief network constructed using traditional machine learning techniques to perform digit classification using the images contained in the MNIST dataset. The network consists of four hidden layers of 1000, 500, 300 and 50 nodes, with each node computing a sigmoid non-linearity (Figure 2.8 illustrates how the network is structurally constructed). Stimuli are provided to the network as a 28×28 pixel image "flattened" into a 768 dimensional vector. The output of the final layer is a 50-dimensional vector that is used as an input to a classifier that classifies the input into one of the ten image classes (one class for each digit from 0 to 9). The connection weights between each layer were computed by training each layer using a greedy unsupervised learning algorithm on Restricted Boltzman Machine (RBM)-based auto-encoders [Tang and Eliasmith, 2010] using images from the MNIST dataset.

3.1.2 NOCH Motor Control Framework

The NOCH (Neural Optimal Control Hierarchy) framework is a control-theoretic approach to modelling the different areas of the brain involved in motor control. It proposes that various motor-related brain areas compute specific functions that are hierarchically combined to perform the overall task of motor control (see Figure 3.1). It takes, as an input, a trajectory in XY space and uses a hierarchical network architecture to compute the control signals required to achieve the desired trajectory, by projecting the XY control signals into a lower-level (but higher dimensional) joint-angle space, and finally, to compute the necessary motor (muscle) commands from the joint-angle space control signals [DeWolf, 2010].

3.1.3 OSE Serial Working Memory Model

The OSE model is a semantic pointer-based model of serial working memory that is able to reproduce several recall effects (primacy and recency for both immediate and delayed recall)

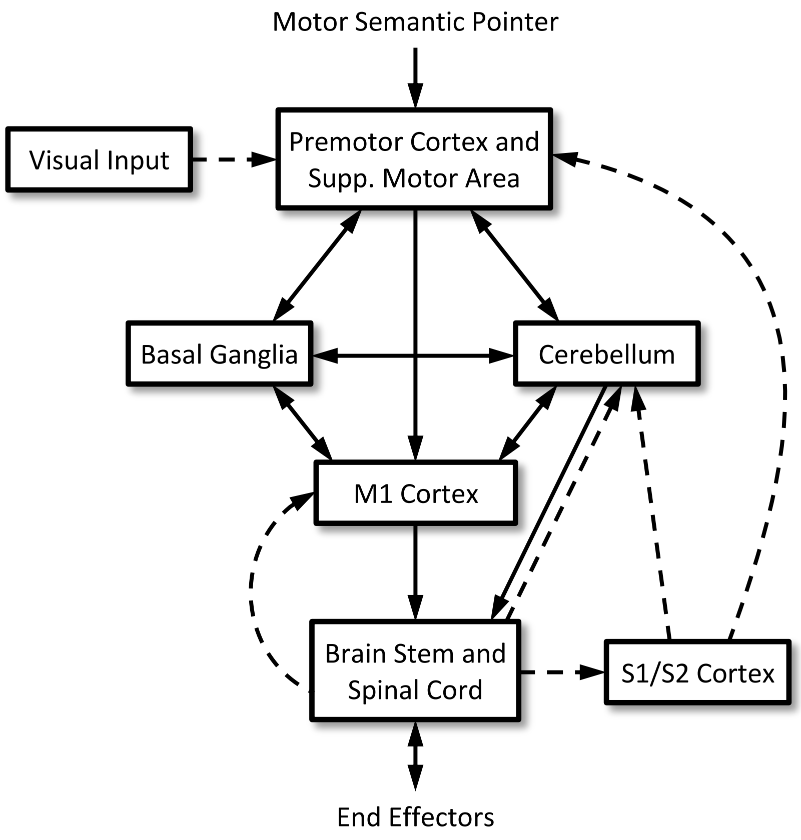


Figure 3.1: Schematic of the NOCH motor control framework. The solid arrows indicate control signals generated within and projected to other parts of the NOCH framework. The dashed arrows indicate feedback signals provided by external sensory stimuli and projected back up through the motor hierarchy. (Figure adapted from [DeWolf, 2010] with permission)

observed in human participants. In the OSE model, information is encoded using positional vectors. As an example, the list [**ONE**, **TWO**, **THREE**] is encoded as:

$$\mathbf{LIST} = \mathbf{POS1} \circledast \mathbf{ONE} + \mathbf{POS2} \circledast \mathbf{TWO} + \mathbf{POS3} \circledast \mathbf{THREE}$$

where **ONE**, **TWO**, and **THREE** are randomly generated HRR vectors representing the individual items in the list, and **POS1**, **POS2**, and **POS3** are randomly generated HRR vectors representing the positional information of each item in the list.

Architecturally, the list information is stored within two groups of multi-dimensional recurrent attractor neural networks (see Figure 3.2A). The recurrent attractor networks are based on work done in [Singh and Eliasmith, 2006]. The two neural attractor networks reproduce the functionality of the working memory system and the hippocampus [Choo, 2010]. It should be noted that it is hypothesized that the compression of semantic pointers due to neural saturation (see Section 2.5.7.1) is responsible for model's ability to faithfully reproduce the memory recall curves observed in human experiments (see Figure 3.2B).

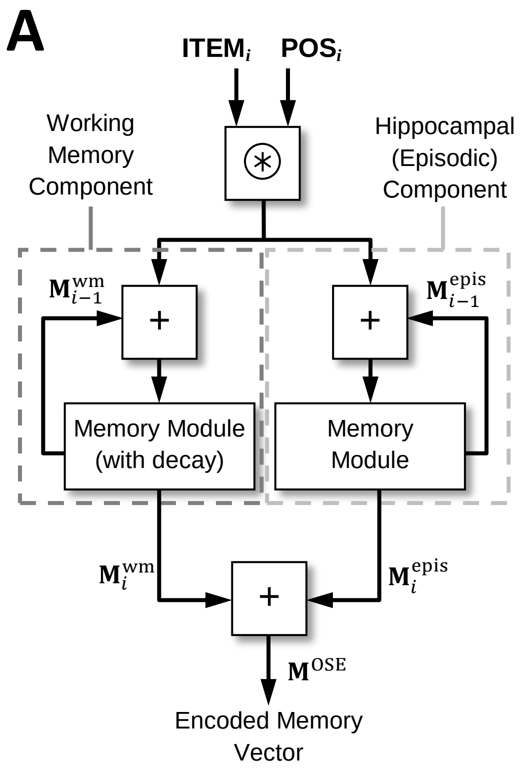
3.1.4 The (Symbol-like) Reasoning System

The reasoning system [Stewart et al., 2010] is an extension of the NEF implementation of the SPA action selection system described in Section 2.5.6.3. It consists of three components: a generalized spiking neuron model of cortex, a spiking basal ganglia network, and a spiking model of the thalamus. The cortex network consists of ensembles and memory networks that represent or store information as semantic pointers. As in Section 2.5.6.3, the basal ganglia network compares all of its input values and determines the one with the highest utility. Finally, the thalamus network takes the output of the basal ganglia and projects it back into semantic pointer space. These three elements are connected in a loop such that the information stored in cortex influences the output of the basal ganglia, thus changing the output of the thalamus. The output of the thalamus then modifies the values stored in cortex, completing the loop (see Figure 3.3). Importantly, the reasoning system presents a method by which this loop – called the cortico-basal ganglia-thalamic loop – can be used to perform complex cognitive reasoning tasks.

The recursive structure of the (cortico-basal ganglia-thalamic) loop allows for the reasoning system to be constructed as a set of condition-consequence pairs. As will be discussed in the sections to follow, in a neural system, the conditions are implemented as projections from the cortex to the basal ganglia. Likewise, the consequences are implemented as projections from the thalamus back to the cortex. By combining the conditions and consequences into the condition-consequence pairs, the reasoning system completes the (cortico-basal ganglia-thalamic) loop.

Conceptually, conditions denote the comparison of the system state to a set of predetermined requirements, and in doing so, generate the utility values necessary for the basal ganglia's operation. Similarly, consequences denote the "actions" (information flow) performed as a result of the winner-take-all-like mechanism of the basal ganglia-thalamus combination.

Together, the condition-consequence pairs allow structures similar to decision trees, state machines, and production rule systems (e.g., [Anderson, 1996; Laird, 2012]) to be constructed in a network of spiking neurons. The remainder of this section describes the neural implementation of different conditional types and different consequence types, and how the entire loop can be employed to perform a series of reasoning tasks (sequence repetition and question answering).



A
ITEM$_i$
POS$_i$
Working Memory Component
Hippocampal (Episodic) Component
$\circledast$
$\mathbf{M}_{i-1}^{\text{wm}}$
$\mathbf{M}_{i-1}^{\text{epis}}$
+
+
Memory Module (with decay)
Memory Module
$\mathbf{M}_{i}^{\text{wm}}$
$\mathbf{M}_{i}^{\text{epis}}$
+
$\mathbf{M}^{\text{OSE}}$
Encoded Memory Vector

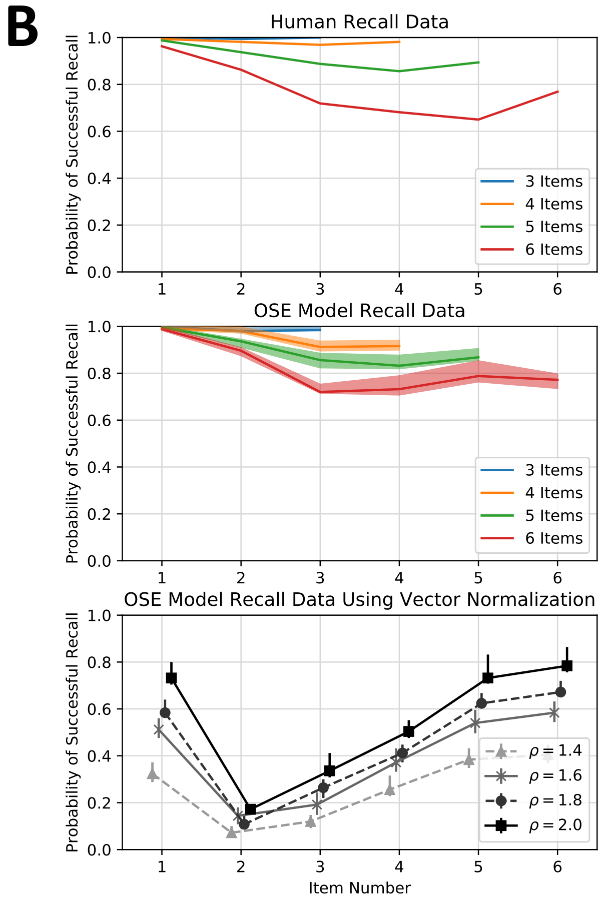


Figure 3.2: (**A**) Schematic of the OSE memory model, comprising of two recurrent attractor networks – a “working memory” component and a “hippocampal” component. (**B**) Results obtained with the OSE model compared to human recall data. (Top) Human serial recall data for lists varying in length from 3 to 6 items. (Middle) OSE model serial recall data for lists varying in length from 3 to 6 items, illustrating that the OSE model is capable of capturing the overall recall trends (primacy and recency) observed in the human recall data. (Bottom) Recall accuracy plots for 6-item lists using the OSE model modified to use vector normalization (L2-norm) rather than depending on neural saturation for normalization. It can be seen that using this method of normalization, the OSE model is unable to match the human recall data. This is true across a range of values for one of the OSE model's free parameters (ρ), which affects the feedback weighting in the hippocampal component of the model. (Figures adapted from [Choo, 2010] with permission)

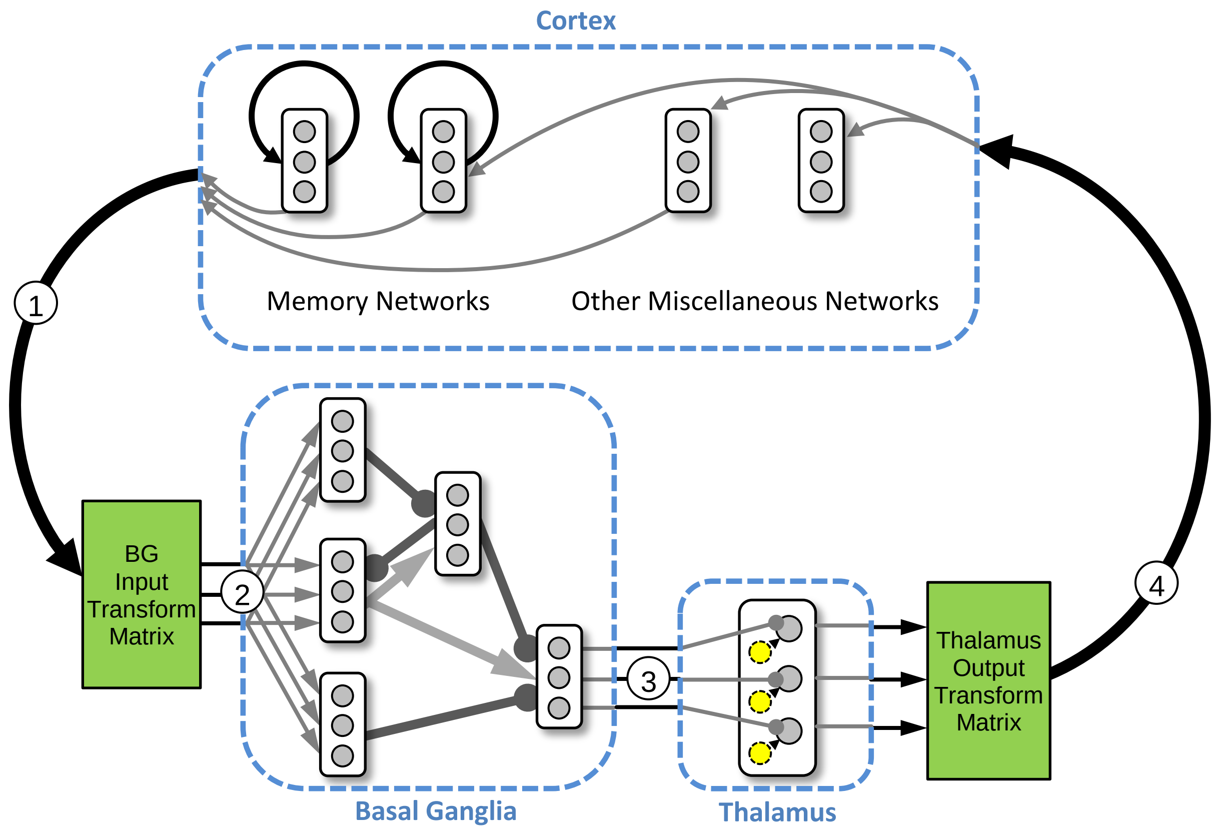

Cortex
Memory Networks
Other Miscellaneous Networks
1
BG Input Transform Matrix
2
Basal Ganglia
3
Thalamus
Thalamus Output Transform Matrix
4

Figure 3.3: Schematic of the reasoning system, illustrating the core cortico-basal ganglia-thalamic loop. Information stored and represented in the cortical areas (①) are projected into the basal ganglia network through an input transformation matrix to generate the basal ganglia input utility values (②). These utility values affect the outputs of the basal ganglia, which serve as inhibitory inputs to the thalamus (③). This in turn affects the output of the thalamus, which through the thalamic output transformation matrix modifies the information stored in cortex (④), completing the cortico-basal ganglia-thalamic loop.

It should be noted that while the reasoning system described in [Stewart et al., 2010] outlines the basic foundation of the reasoning system, the notation and definitions used in the succeeding sections have been expanded, formalized and formulated to facilitate specifically the discussion of Spaun's action selection system (see Section 3.3.8). In addition, the implementation of several networks, in particular the consequence networks (described below), have been modified to improve the ease of integration into the Spaun model.

3.1.4.1 Notation

In this thesis, the following notation is used to denote the different components of the condition-consequence pairs:

- Lower-case italicized names are used to represent the locations holding state information (i.e., the neural ensembles representing the state semantic pointers). As an example, the symbol "*vision*" might be used to represent state information originating in the visual system.
- Condition statements are represented using algebraic and HRR-SPA operators (in particular, the dot product operator ($\bullet$)). For example, a condition statement might be represented as: $0.5 \times (\mathit{vision} \bullet \mathsf{\mathbf{BLUE}}) + 0.5 \times (\mathit{vision} \bullet \mathsf{\mathbf{RED}})$
- Consequence statements are also represented using algebraic and HRR-SPA operators. In addition, because consequences result in information being transferred back into cortex, the "$\Rightarrow$" symbol is used to signify this transfer. As an example, a consequence that results in the semantic pointer "**BLUE**" being transferred into visual memory (*vis_mem*) is written as: $\mathsf{\mathbf{BLUE}} \Rightarrow \mathit{vis\_mem}$. Consequences that result in multiple consecutive transfers to different areas of cortex are written as comma-separated list of each transfer, for example: $\mathsf{\mathbf{BLUE}} \Rightarrow \mathit{vis\_mem}, \mathsf{\mathbf{WALK}} \Rightarrow \mathit{motor}$
- Conditions and consequences are paired up using the "$\longmapsto$" symbol. For example: $\mathit{vision} \bullet \mathsf{\mathbf{RED}} \longmapsto \mathsf{\mathbf{RED}} \Rightarrow \mathit{vis\_mem}$

3.1.4.2 Conditions

There are two types of conditions in the reasoning system: static conditions, and dynamic conditions. As previously stated, the goal of both condition types are to generate the utility values necessary for the basal ganglia network's operation. The difference between the two types of conditions are the methods by which the utility values are computed.

To generate the basal ganglia utility values, static conditions involve the comparison of the state of the system (represented by spiking neural ensembles) to pre-specified semantic pointer values that do not change as the system is running. In the HRR-SPA, this comparison is performed using the dot product operator:

$$\mathit{state} \bullet \mathsf{\mathbf{STATIC\_SP}}$$

Because one operand of the dot product operation is unchanging, the spiking NEF implementation of comparison can be computed by using a transformation matrix $\mathbf{M}$ composed of the necessary pre-specified semantic pointer values, similar to the methods used to construct the cleanup memory networks.

While static conditions make comparisons to unchanging semantic pointer values, dynamic conditions compare the state of the system to a semantic pointer stored in other areas of cortex (i.e., in a neural ensemble), which can change values during the course of operation of the system. These other areas of cortex can also be used as state information, and as such, dynamic conditions are written as:

$$state1 \bullet state2$$

Because both operands of the dot product operation are variable with the dynamic condition, the dot product operation cannot be computed as a linear transform. Instead, a dot product network[22] is added between the cortical state ensembles and the input of the basal ganglia network. It should be noted that because *each* dynamic condition requires its own dot product network, the NEF implementation of the dynamic conditions is expensive in terms of neuron counts. Figure 3.4 illustrates the implementation of a set of static conditions, and a set of dynamic conditions.

In addition to static and dynamic conditions, individual conditions can be constructed as compound conditions, where multiple conditions can be combined to exhibit behaviours like the binary operators AND, OR, and NOT. In the examples that follow, static conditions are used to simplify the conditional constructs. Additionally, it is assumed that the maximum utility value a condition can have is 1, as this is consistent with the definition of the dot product operator, as well as the understanding that, by default, scalar neural ensembles constructed using the NEF are optimized to represent values in the range $(-1, 1)$.[23]

Compound OR Conditions: Compound OR conditions can be constructed simply by adding multiple conditions together. Assuming that the maximum utility value of the conditions is 1, as long as one of the conditions is true, the overall conditional utility value should be greater than 1. Figure 3.5 illustrates a 3 set setup of compound OR conditionals and an example operation.

This approach works in the majority of cases; however, there are some caveats that must be considered. First, conditions can take negative values, which means that the combined conditional utility might not be greater than 1, reducing the effectiveness of the "OR" behaviour (see Figure 3.5B, segment ④). Additionally, the recurrent and neural nature of the basal ganglia network means that extremely large utility values – which are possible if all of the conditions

[22]In the NEF, a dot product network can be constructed using an ensemble array of multiplication sub-ensembles. The output of the multiplication sub-ensembles are summed to produce the dot product value.

[23]The convention that 1 is the maximum utility value for the basal ganglia network is not strict. This convention can be violated as long as the appropriate changes are made to the neural ensembles within the basal ganglia, and that every condition follow the same convention.

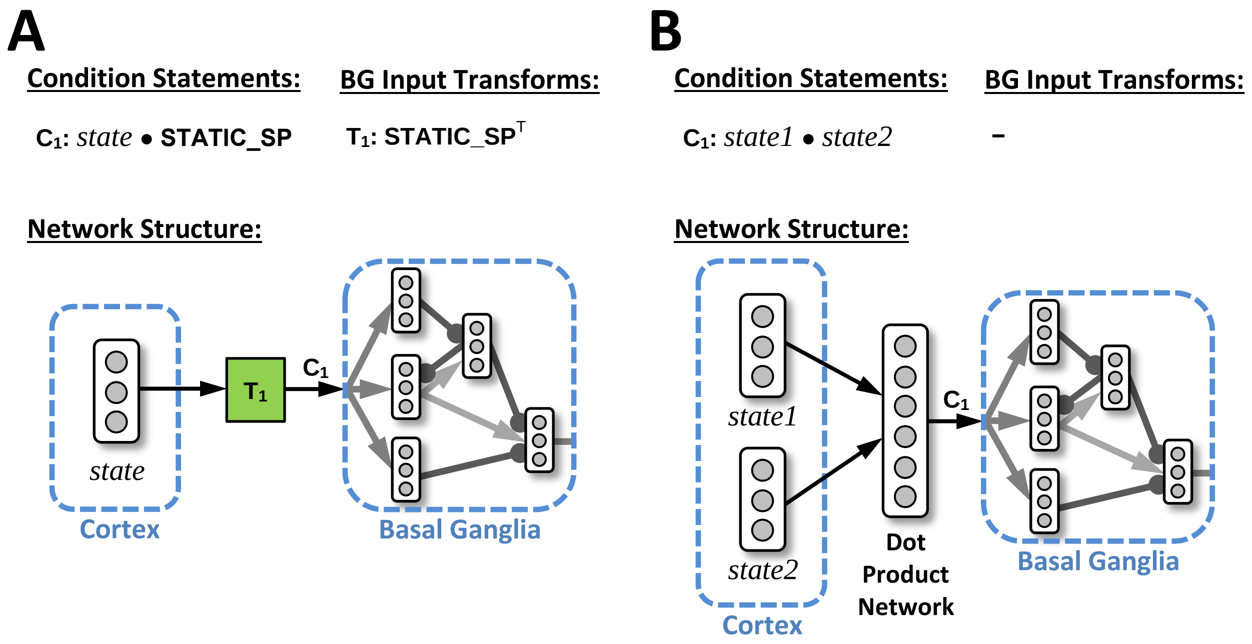


Figure 3.4: Schematic of example implementations of the static and the dynamic conditions of the reasoning system. (**A**) Schematic of the implementation of the static condition ($state \bullet$ **STATIC_SP**). The basal ganglia input utility value is determined by computing the dot product between the output of the *state* ensemble, and the static semantic pointer **STATIC_SP**. This is implemented using a transformation matrix with the value **STATIC_SP**^{T}, similar to the dot product operation in the cleanup memory networks. (**B**) Schematic of the implementation of the dynamic condition ($state1 \bullet state2$). The basal ganglia input utility value is determined by computing the dot product between the output of the $state1$ and $state2$ ensembles, through the use of a neural dot product network.

in the compound *OR* condition evaluate to true – can cause the basal ganglia network to exhibit unexpected behaviours (e.g., choosing the wrong utility value, or sticking to the previously chosen utility value). Both of these caveats can be mitigated by adding additional networks to "pre-process" the utility values of the constituent conditions.

Compound AND Conditions: Like the compound *OR* conditions, the compound *AND* conditions can be constructed by using the addition operator. However, each constituent condition is scaled such that the compound condition only evaluates to 1 when all of the constituent conditions are met. Figure 3.6 illustrates a 2 set setup of compound *AND* conditionals and an example operation.

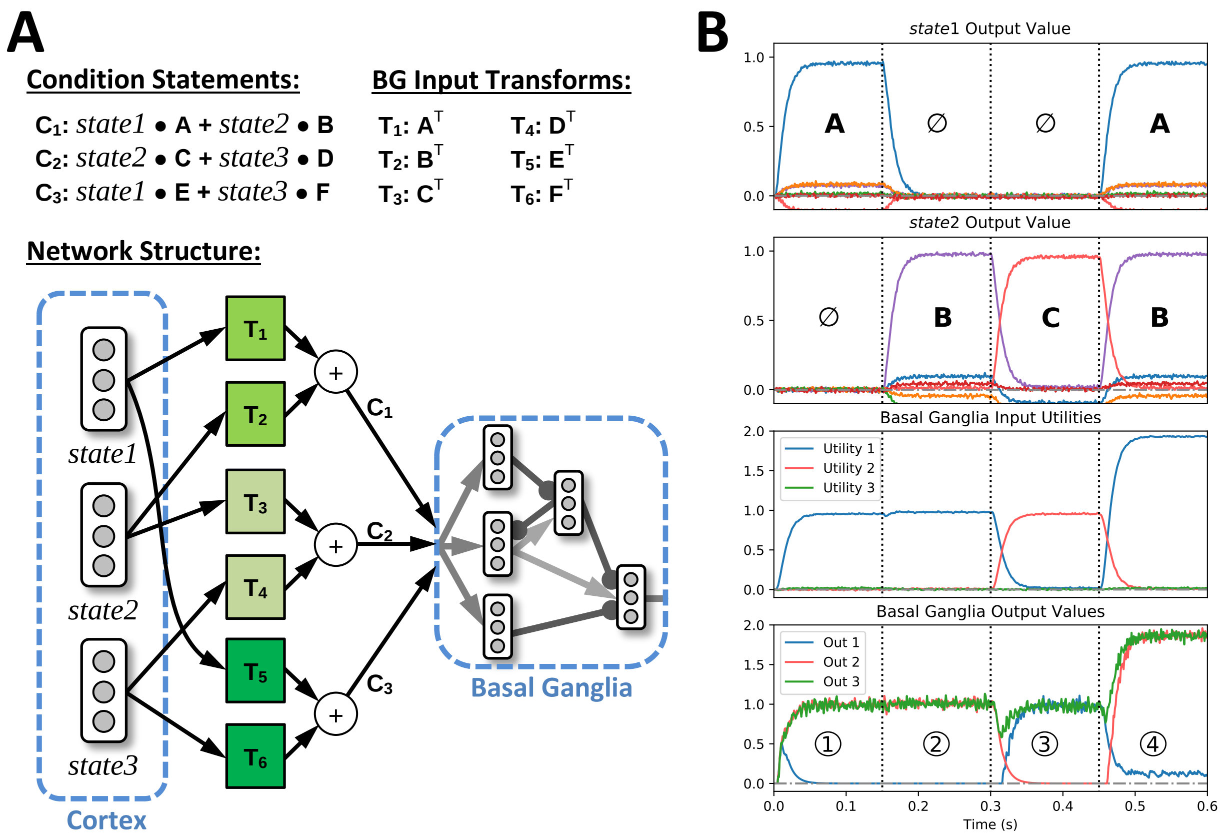


Figure 3.5: Schematic and operation of example OR conditionals in the reasoning system. (**A**) Schematic of a 3-set setup of OR conditionals, where the first condition only activates when *state*1 is **A** OR when *state*2 is **B**, the second condition only activates when *state*2 is **C** OR when *state*3 is **D**, and the third condition only activates when *state*1 is **E** OR when *state*2 is **F**. Listed are the equivalent condition statements, and the necessary basal ganglia input transforms needed to compute the appropriate basal ganglia utility values. Additionally, the network structure equivalent of the 3-set OR conditionals is illustrated. (**B**) Example operation of the OR conditionals. The plots are divided into four segments, separated by the dashed lines. For compactness, only the outputs of *state*1 and *state*2 are displayed, with the output of *state*3 fixed to be $\emptyset$. [①] In this segment, it is observed that satisfying part of the first conditional (i.e., *state*1 is **A**) results in the first conditional being activated ("Out 1")(recall that the basal ganglia output is inhibitory). [②] In this segment, it is observed that satisfying the second half of the first conditional (i.e., *state*2 is **B**) results in the first conditional being activated ("Out 1"). [③] In this segment, the value of *state*2 changes to **C**, satisfying the second conditional, resulting in the appropriate change in the activated conditional ("Out 2"). [④] In this final segment, both parts of the first conditional are satisfied, resulting in a utility value ("Utility 1") that is outside the optimized range of the basal ganglia network (which is typically from 0.5 to 1.5). This causes an unexpected output of the basal ganglia network – in this case, rather than being activated, the first condition ("Out 1") is slightly inhibited, as evidenced by a positive deviation of the output value from the zero line. As a reminder, the output of the BG network is inhibitory, and thus, if all of the outputs are positively valued (as is the case in this segment), none of the conditionals are active.

As with the compound *OR* conditional, the compound *AND* conditional comes with caveats, primarily due to the non-binary nature of the real-valued utility values. Figure 3.6B, segments ④ and ⑤ illustrate two consequences of the real-valued nature of the utility value. In the first example, a compound *AND* conditional has an overall utility value equal than 1 despite only having half of the conditions met. In the second example, because none of the conditions evaluate to a higher utility, a compound *AND* conditional is "chosen" when only a portion of its conditions are met, even when the overall utility of the compound condition evaluates to less than 1. In contrast, in a purely binary implementation, because none of the conditionals are fully satisfied, none of the conditions would have been chosen.

NOT Conditions: In the reasoning system, the *NOT* condition (i.e., the condition should be "chosen" when the state is any value *other* than the specified semantic pointer) is implemented using a biased subtraction. In essence, if a negation of the conditional value C is desired, the system is configured such that the utility calculation $1 - C$ (see Figure 3.7). As with the *OR* and *AND* conditionals, the *NOT* condition is not without caveats, particularly in ensuring the output of the utility calculation is well defined. As an example, unexpected results can occur when the utility value of the conditional C is between 0 and 1, because the $1 - C$ computation may result in a utility value high enough for the condition to be activated.

Additionally, combining the *NOT* conditional with other compound conditions can be unintuitive. As an example, compounding two conditions together to form an *AND* conditional typically involves applying weights to each condition. To illustrate, the conditional statement for "$state1$ is $\mathbf{A}$ *AND* $state2$ is $\mathbf{B}$" is:

$$(0.5 \times state1 \bullet \mathbf{A} + 0.5 \times state2 \bullet \mathbf{B}).$$

Likewise, the compound condition "$state1$ is $\mathbf{A}$ *AND* $state2$ is *NOT* $\mathbf{B}$" can be stated as:

$$(0.5 \times state1 \bullet \mathbf{A} + 0.5 \times (1 - state2 \bullet \mathbf{B})).$$

However, a "cleaner" definition of the compound *NOT* condition (see Appendix A.3) is:

$$(state1 \bullet \mathbf{A} - state2 \bullet \mathbf{B}).$$

Segments ③ through ⑥ of Figure 3.7B also illustrate two other examples of compounding the *NOT* with the *AND* and *OR* conditionals.

Default Conditions: The reasoning system also allows for the implementation of "default" conditions, which are conditions that are "chosen" when all other conditions are not fully met.

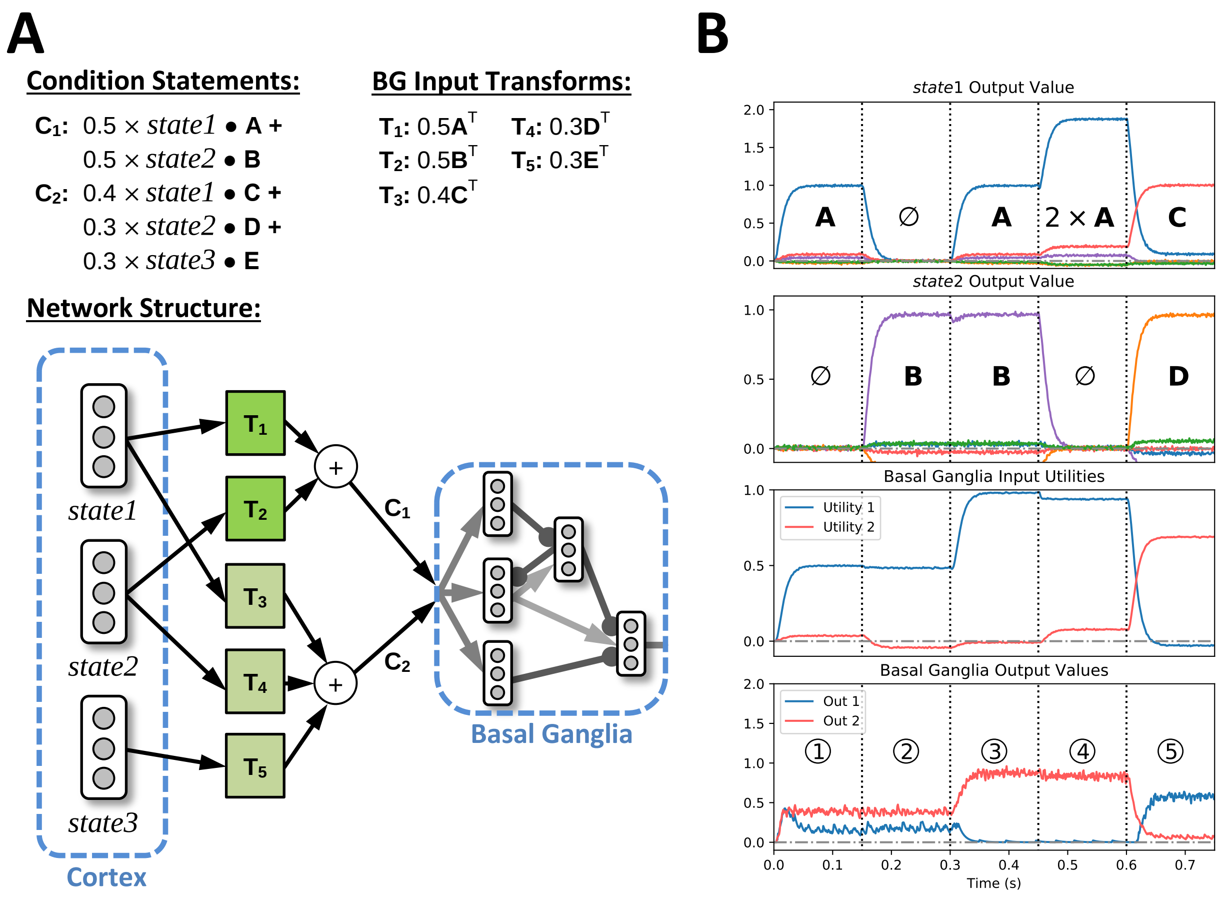


Figure 3.6: Schematic and operation of example AND conditionals in the reasoning system. (**A**) Schematic of a 2-set setup of AND conditionals, where the first condition only activates when *state*1 is **A** AND when *state*2 is **B**, and the second condition only activates when *state*1 is **C** AND when *state*2 is **D** AND when *state*2 is **E**. Listed are the equivalent condition statements, and the necessary basal ganglia input transforms needed to compute the appropriate basal ganglia utility values. Additionally, the network structure equivalent of the 2-set AND conditionals is illustrated. (**B**) Example operation of the AND conditionals. The plots are divided into five segments, separated by the dashed lines. For compactness, only the outputs of *state*1 and *state*2 are displayed, with the output of *state*3 fixed to be $\emptyset$. Additionally, the basal ganglia network is configured such that only utility values greater than 0.5 will activate it. [① & ②] In these segments, it is observed that satisfying only half of the first AND conditional (*state*1 is **A**, followed by *state*2 is **B**) does not result in any conditions being activated. [③] In this segment, the first AND conditional is fully satisfied, resulting in a utility value of 1 ("Utility 1"), and an activated condition ("Out 1"). [④] Here, the input to the first condition is "*state*1 is 2 × **A**", which technically only satisfies half of the conditional. However, because the computed utility ("Utility 1") has a value of 1 (due to the non-binary nature of the computation), the first condition remains activated ("Out 1"). [⑤] In the final segment, two of the three conditions in the second AND conditional is satisfied (*state*1 is **C** AND *state*2 is **D**), resulting in a utility value of 0.7. However, because the utility value of this condition is large enough to trigger the basal ganglia network, it is erroneously activated ("Out 2").

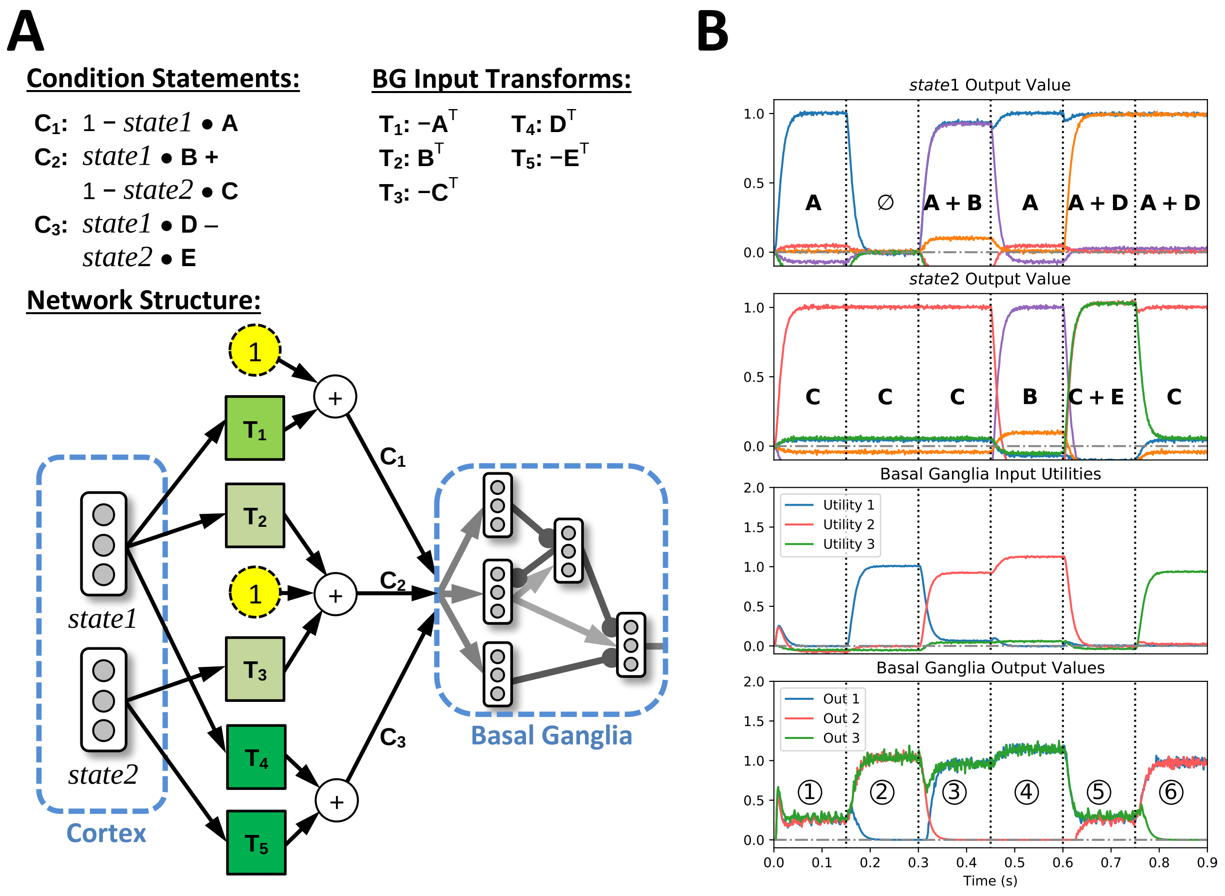


Figure 3.7: Schematic and operation of example *NOT* conditionals in the reasoning system. (**A**) Schematic of a 3-set setup of compound *NOT* conditionals. The first condition activates whenever *state*1 is *NOT* **A**, the second condition activates when *state*1 is **B** *OR* when *state*2 is *NOT* **C**, and the third condition activates when *state*1 is **D** *AND* when *state*2 is *NOT* **E**. Listed are the equivalent condition statements, the necessary basal ganglia input transforms needed to compute the appropriate basal ganglia utility values, and the network implementation. The constant "1" inputs to the conditionals are provided by bias populations, similar those in the thalamus network. (**B**) Example operation of the *NOT* conditionals. The plots are divided into six segments, separated by the dashed lines. [① & ②] These segments demonstrate the first (**C**$_1$: singular *NOT*) conditional. In segment ①, none of the conditions are satisfied, thus none of them are activated. In segment ②, *state*1 is **∅**, satisfying the first condition ("Out 1"), resulting in it being activated. [③ & ④] These segments demonstrate the second (**C**$_2$: compound *NOT* and *OR*) conditional. In segment ③, the first half of the second condition (*state*1 is **B**) is met, thus the second condition ("Out 2") is active. Similarly, in segment ④, the second half of the second condition (*state*2 is **B**, which is *NOT* **C**) is met, thus "Out 2" remains active. [⑤ & ⑥] These segments demonstrate the third (**C**$_3$: compound *NOT* and *AND*) conditional. In segment ⑤, only half of the third conditional is met (*state*1 is **D**), thus resulting in no active conditionals. In segment ⑥, both halves of the third condition are met (*state*1 is **D** *AND* *state*2 is *NOT* **E**). Consequently, the third conditional is activated ("Out 3").

Default conditions are implemented as conditions consisting of only scalar values. The scalar value is chosen such that is it between the potential maximum utility value of a partially met condition and the utility value of a fully met condition (typically 1). Figure 3.8 illustrates an example implementation of a default condition, and its interaction with other compound conditions.

3.1.4.3 Consequences

While conditions transform the state of the system into utility values for the basal ganglia network, consequences perform the inverse role of transforming the output of the thalamic network into usable values projected back into cortex.

As with the conditions, consequences also come in multiple types: static consequences, dynamic consequences, and hybrid consequences. Static consequences result in a constant value output to the cortex, while dynamic consequences result in a value being transferred from one area of cortex to another, regardless of the actual value being transferred. Hybrid consequences combine the effect of the static and the dynamic consequence types, and result in the cortex being fed with a value that is a result of a value in cortex combined with a static value (typically these values are combined with the SPA binding operation).

Regardless of type, consequences are implemented as transformations on the output of the thalamus network (see Figure 2.27 for the spiking neuron implementation of the thalamus network). Static consequences are implemented as direct transforms that convert the activated "1" output of the thalamus into a semantic pointer as part of the projection from the thalamus to an area of cortex. If the consequence is not activated, the "0" output of the thalamus is converted into the null semantic pointer. Figure 3.9 illustrates the implementation and operation of the static consequences.

Dynamic consequences are implemented as gates positioned between two desired areas of cortex. The output of the thalamus is inverted and used as the inhibitory control signal for the gate. With this implementation, when the dynamic consequence is active, the gate is dis-inhibited, allowing information to flow from one area of cortex to another. When the dynamic consequence is inactive, the gate is inhibited, stopping the information flow. Dynamic consequences can also modify the value of the information being transferred from one cortical area to another. Typically, it is desirable to have the value stored in one area of cortex be bound to a static semantic pointer before projecting it into another area of cortex. These types of dynamic consequences (henceforth referred to as "hybrid" consequences, as they are a cross between the static consequence and the "standard" dynamic consequence) are implemented in the same way as previously described, with the exception of an additional transform matrix to perform the desired binding (or other) SPA operations. Figure 3.10 illustrates the implementation of the dynamic and hybrid consequences.

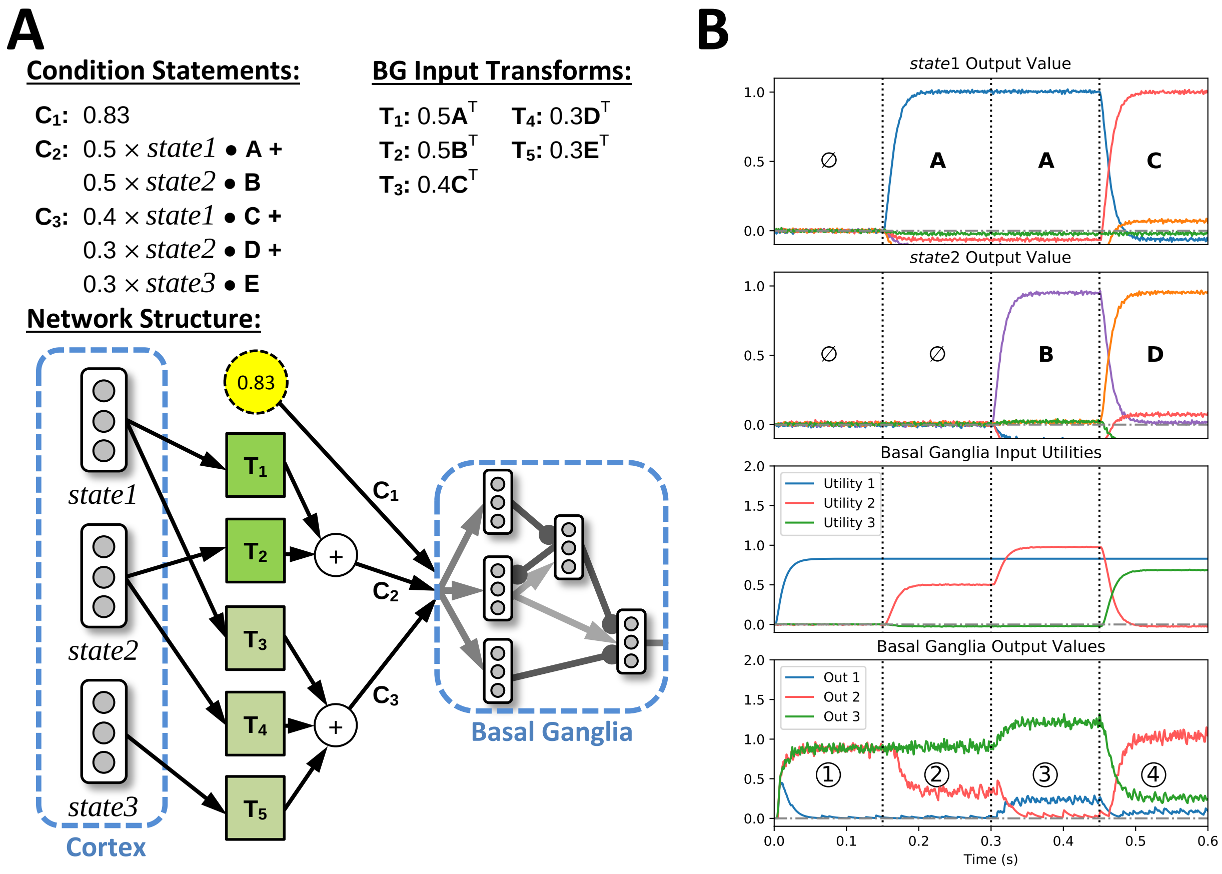
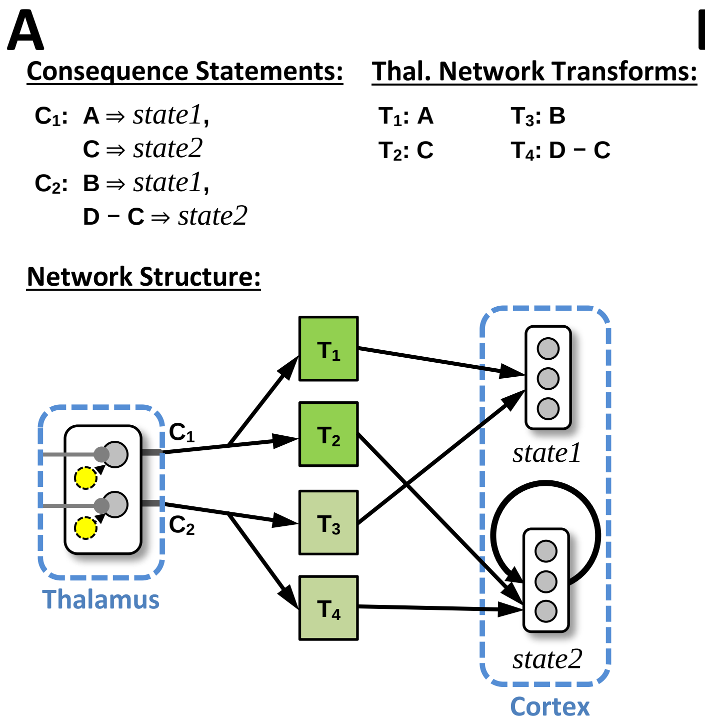


Figure 3.8: Schematic and operation of an example default conditional in the reasoning system. (**A**) Schematic of a 3-set setup of compound conditionals. The first condition is a default conditional that feeds a constant utility of 0.83 to the basal ganglia network. The second and third conditionals are identical to the AND conditionals used in Figure 3.6. (**B**) Example operation of the default conditional. For compactness, only the outputs of $state1$ and $state2$ are displayed, with the output of $state3$ fixed to be $\emptyset$. [①] This segment demonstrates the default conditional operating with no competition from other conditionals. As observed, the default condition is correctly activated ("Out 1"). [② & ③] These segments demonstrate the default conditional interacting with the second conditional. In segment ②, only half of the AND conditional is met, and the resulting utility value ("Utility 2") is lower than the default conditional utility ("Utility 1"), resulting in the correct behaviour of the default condition ("Out 1") being activated. In segment ③, the AND condition is fully met, resulting in a utility higher than the default conditional utility, thus allowing the second condition to override the default condition and be activated ("Out 2"). [④] This segment demonstrates the default conditional interacting with the third conditional. From Figure 3.6B segment ⑤, it can be deduced that for these inputs (i.e., $state1$ is **C** AND $state2$ is **D**, identical to those used in Figure 3.6), for a network without the default conditional, this condition ($\mathbf{C}_3$) would be incorrectly activated (when the conditional is only partially satisfied). In this network, with the addition of the default condition, which has a utility value of 0.83 – higher than the two-thirds activation of the third conditional, but lower than the full activation of any condition – it is observed that for these inputs, the third condition ("Out 3") is no longer active and the default condition ("Out 1") is correctly activated.



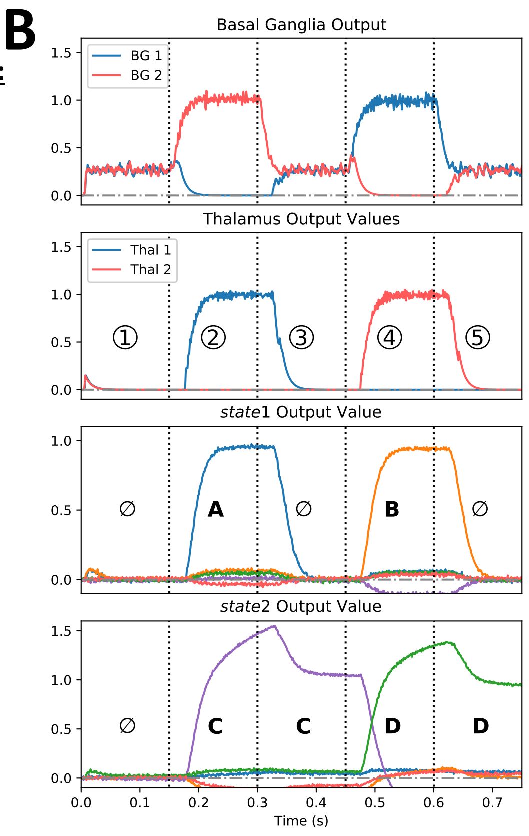


Figure 3.9: Schematic and operation of example static consequences in the reasoning system. (**A**) Schematic of a 2-set setup of static consequences. The consequences are set up to modify the values stored in the cortical ensemble *state*1 and the cortical memory network *state*2. When active, the first consequence projects **A** into *state*1, and projects **C** into *state*2. When active, the second consequence projects **B** into *state*1, and projects (**D** - **C**) into *state*2. Note that the addition of $-$**C** to the value projected to *state*2 is to remove the potential value of **C** stored in the *state*2 memory network. Shown are the equivalent consequence statements, the implemented network structure, and the necessary thalamic output transform matrices. (**B**) Example operation of the two consequence statements. [①] Here, the output of the basal ganglia (Top graph) indicates that the none of the consequences are active, resulting in no change to neither *state*1 nor *state*2. [Segments ② & ③] These segments demonstrate the operation of the first consequence statement. In segment ② the basal ganglia output indicate consequence 1 is activate ("Thal 1"), resulting in the change of the value of *state*1 to **A**, and the value of *state*2 to **C**. In segment ③, the basal ganglia output once again indicate no consequences are active, resulting in the output of the thalamus to drop to 0. Since *state*1 has no memory, it reflects this change, and its value changes to $\emptyset$. *state*2, however, is a memory network, and thus maintains its value (**C**) even when the thalamus is producing no output. [④ and ⑤] These segments demonstrate the operation of the second consequence statement. The order of operations is similar to the first consequence statement: In segment ④, the second consequence is active, resulting in the appropriate changes in the values of *state*1 and *state*2, and in segment ⑤, all consequences are inactive, causing *state*1 to loose its value, and *state*2 to remember its last input.

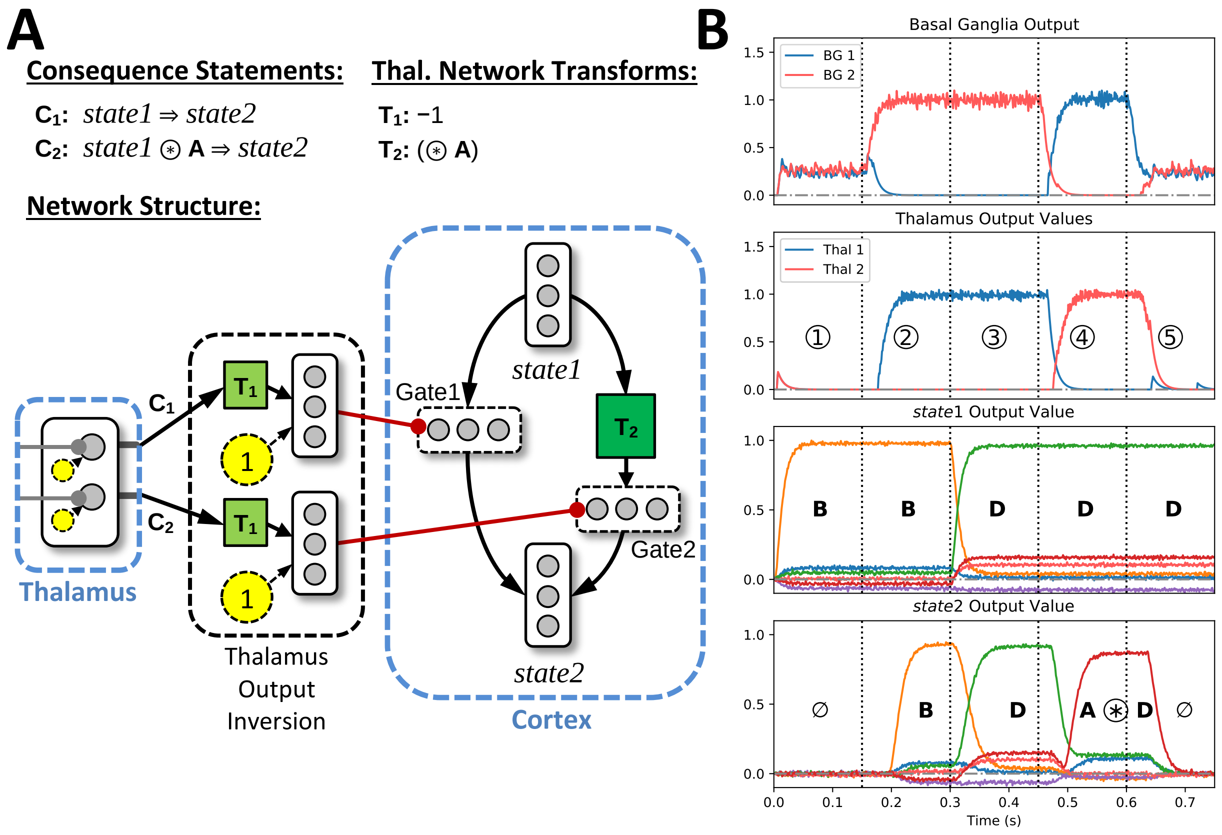


Figure 3.10: Schematic and operation of example dynamic and hybrid consequences in the reasoning system. (**A**) Schematic of a 2-set setup of on dynamic consequence ($\mathbf{C}_1$) and one hybrid consequence ($\mathbf{C}_2$). Both consequences are set up to modify the value stored in the cortical ensemble *state*2. When active, the first consequence projects the value in *state*1 into *state*2 (by dis-inhibiting the "Gate1" ensemble). When active, the second consequence projects the value of *state*1 ⊛ **A** into *state*2 (by dis-inhibiting the "Gate2" ensemble). Shown are the equivalent consequence statements, the implemented network structure, and the necessary thalamic output transform matrices. (**B**) Example operation of the dynamic and hybrid consequence statements. In the example operations of the consequence statements, the values in *state*1 are not modified by the consequences, rather they are modified by the experimental (external) setup. [① & ⑤] In these segments, the output of the basal ganglia (Top graph) indicates that the none of the consequences are active, resulting in no value being projected to *state*2. [② & ③] These segments demonstrate the dynamic consequence. In segment ②, when the consequence is activated ("BG 1" and "Thal 1"), the value represented in *state*1 is projected into *state*2. In segment ③, the consequence remains active, however, the value represented in *state*1 is modified to **D**, and the change is reflected in *state*2 demonstrating that the information channel between *state*1 and *state*2 is still open. [④] Here, the hybrid consequence is activated. In this consequence, instead of projecting the value represented in *state*1 directly into *state*2, it is passed through a transformation matrix that computes the binding operation with **A**. As a result, when the hybrid consequence is activated, the value in *state*2 becomes **A** ⊛ **D** (where **D** was the value represented in *state*1).

Thus far, the consequences described involve the transfer of information where computation performed on the "source" of the information (i.e., the LHS of the $\Rightarrow$ in the consequence statements) consists of at most one "dynamic" (i.e., a non-static semantic pointer value represented in a cortical ensemble) operand. The reasoning system also allows for "sources" to consist of complex SPA operations performed on multiple dynamic operands. The implementation of these "compound" consequences is similar to that of the dynamic consequences. However, instead of using a gate ensemble between the desired areas of cortex, a neural network that has been configured to performed the desired SPA operations is placed between the cortical areas, and the (inverted) output of the thalamus is used to gate the output of this network (see Figure 3.11).

3.1.4.4 Example Reasoning Task Implementation: Sequence Repetition

This section demonstrates how the reasoning system can be configured to perform the task of sequence repetition [Stewart et al., 2010]. Here, the goal of the system is to continuously cycle through a pre-determined sequence of state values, demonstrating that the reasoning system can be configured to create cognitive networks that are entirely self-driven and self-sustaining.

To configure the reasoning system for the sequence repetition task, a condition-consequence pair is constructed for each desired state value. The condition of the condition-consequence pair is set as a specific state semantic pointer, and the consequence is set to be the desired next sequentially ordered state semantic pointer. In this example, the state values **A**, **B**, and **C** are used, with the desired repetition sequence being **A** followed by **B** then **C** and back again to **A**. It is important to note that the system should be able to complete the sequence repetition given any starting state. The full set of condition-consequence pairs for this task are:

$$\begin{aligned} state \bullet \mathbf{A} &\longmapsto \mathbf{B} \Rightarrow state \\ state \bullet \mathbf{B} &\longmapsto \mathbf{C} \Rightarrow state \\ state \bullet \mathbf{C} &\longmapsto \mathbf{A} \Rightarrow state \end{aligned}$$

Figure 3.12 illustrates the state diagram for a 3 state sequence, as well as the implementation and operation of the spiking network equivalent implemented as a reasoning system.

3.1.4.5 Example Reasoning Task Implementation: Question Answering

This section demonstrates how the reasoning system can be configured to perform a rudimentary question and answering task [Stewart et al., 2010]. This task is more complex than the sequence repetition task, requiring the model to remember information it has been provided, as well as to use that information to correctly answer queries when prompted.

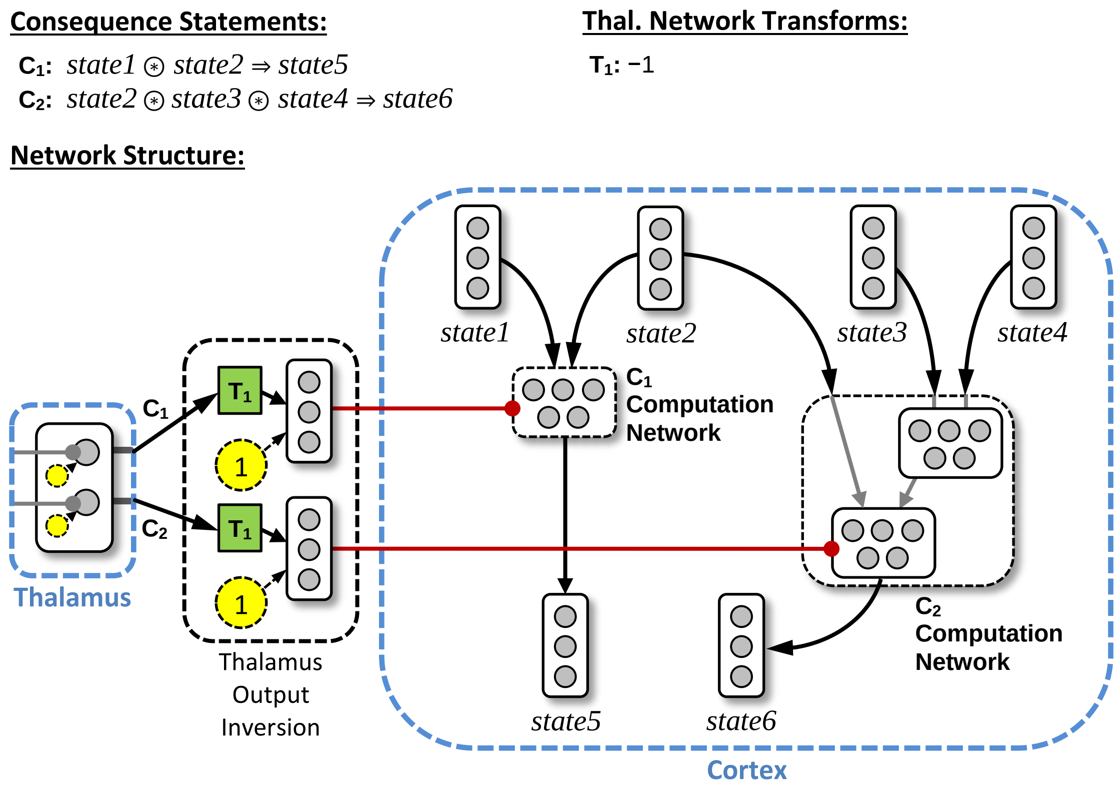


Figure 3.11: Schematic of example compound consequences in the reasoning system, illustrating a 2-set setup of compound consequences. In the first compound consequence, when it is active, the value of *state*1 ⊛ *state*2 is projected into *state*5. This is accomplished by using the inverted thalamic output (**C**$_1$) to dis-inhibit the output of the computational network related to the first consequence ("**C**$_1$ Computation Network"). In this example, the C$_1$ computation network would be configured to calculate the SPA binding operation. The second compound consequence is implemented similar to the first compound consequence, with the only difference being that the second computation network is more complex. However, as in the implementation of the first compound consequence, the second compound consequence is implemented by dis-inhibiting the output of the computation network ("**C**$_2$ Computation Network") when the consequence is active.

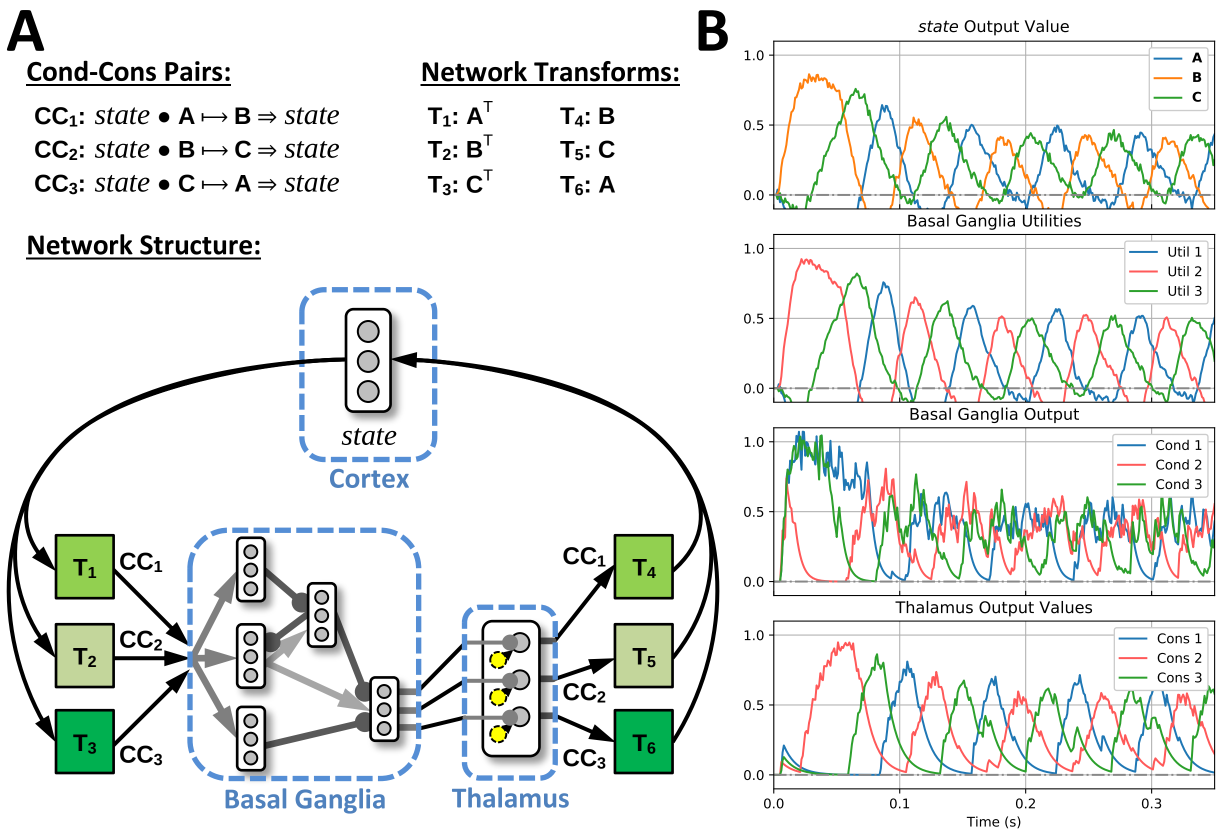


Figure 3.12: Schematic and operation of the reasoning system configured to perform the sequence repetition task. (**A**) Schematic of the reasoning system configured to perform the sequence repetition task. Shown are the condition-consequence pairs used in the network (as described in the main text), the network transforms necessary for this network, and the implementation of the network itself, using the static condition and static consequence sub-networks described previously. (**B**) Example operation of the sequence repetition task. In this example, the value represented in *state* is initialized to **B**, to demonstrate that the sequence repetition network is capable of performing the sequence repetition task from any point in the sequence. From the plots, the value representing in the *state* (Top) determines the value of the basal ganglia input utilities (Second). The input utility values then affect the output of the basal ganglia network (Third), in turn modifying the output of the thalamus network (Bottom), changing the value represented in the *state* ensemble, and completing the loop. As an example, the value in *state* is initialized to **B**, resulting in the activation of the second condition ("Cond 2"), and the activation of the second consequence ("Cons 2"). The activation of the second consequence projects the value of **C** into *state*, changing its value to **C**. The process then repeats with activation of the third condition-consequence pair ("Cond 3" and "Cons 3").

For this task, the reasoning system is configured to interact with three cortical states: it receives information from a "visual processing" area (*vision*), it stores information in a "working memory" area (*memory*), and it outputs information to a "motor planning" area (*motor*). Within the model, information is represented using a schema similar to Eq. (2.5) (i.e., a collection of bound semantic pointers).

Conceptually, the question answering task can be divided into two distinct sub-tasks: storing information in memory when prompted, and retrieving the appropriate information (to answer a question) from memory when directed. To distinguish the two commands, "cue" semantic pointers are added to the visual semantic pointer representation. As an example, to direct the model to remember a scene with a blue square and a red triangle[24], the provided visual semantic pointer would be:

$$\mathbf{STATEMENT} + \mathbf{BLUE} \circledast \mathbf{SQUARE} + \mathbf{RED} \circledast \mathbf{TRIANGLE} \tag{3.1}$$

Likewise, to prompt the model to provide a response to the question "which shape is blue?", the provided visual semantic pointer would be:

$$\mathbf{QUESTION} + \mathbf{BLUE} \tag{3.2}$$

The first sub-task results in information being transferred from the visual area to the working memory area when the **STATEMENT** prompt is provided. With the reasoning system, this can be achieved using a single condition-consequence pair. As the sub-task causes information to flow from the visual area to the memory area the consequence can be stated as:

$$vision \Rightarrow memory \tag{3.3}$$

To construct the condition for this sub-task, it is noted that the condition-consequence pair should activate as long as the **STATEMENT** prompt is being provided. Thus, the condition can be stated as:

$$vision \bullet \mathbf{STATEMENT} \tag{3.4}$$

The condition takes advantage of the dot product operator, as the dot product operation of Eq. (3.1.4.5) with **STATEMENT** should produce a value close to 1. This is true as long as **STATEMENT** is present in the visual semantic pointer, regardless of the additional object infor-

[24]To reduce the complexity of the model, it is assumed that objects are described by a maximum of 2 characteristics. For the example given, each object is described by its colour and shape.

mation that is included in the collection. As an example:

$$(\mathsf{\textbf{STATEMENT}} + \mathsf{\textbf{BLUE}} \circledast \mathsf{\textbf{SQUARE}} + \mathsf{\textbf{RED}} \circledast \mathsf{\textbf{TRIANGLE}}) \bullet \mathsf{\textbf{STATEMENT}} \approx 1$$
$$(\mathsf{\textbf{STATEMENT}} + \mathsf{\textbf{GREEN}} \circledast \mathsf{\textbf{CIRCLE}} + \mathsf{\textbf{BLACK}} \circledast \mathsf{\textbf{DIAMOND}}) \bullet \mathsf{\textbf{STATEMENT}} \approx 1$$

Combining the condition and consequence results in the condition-consequence pair for the first sub-task:

$$vision \bullet \mathsf{\textbf{STATEMENT}} \longmapsto vision \Rightarrow memory \tag{3.5}$$

The second sub-task involves transferring information from the working memory area to the motor area when the **QUESTION** prompt is provided. The condition for the second sub-task follows the same style as the condition for the first sub-task, activating only when a specific prompt is provided to the model. As such, the condition for the second sub-task is written as:

$$vision \bullet \mathsf{\textbf{QUESTION}} \tag{3.6}$$

Unlike the first sub-task, the consequence of the second sub-task involves more than just a straight-forward transfer of information from one area of cortex to another. Equation (2.6) demonstrates that in order to extract the appropriate information from the stored semantic pointer, it needs to be bound to the inverse of a query semantic pointer. In the case of this model, the query semantic pointer is represented in the *vision* state, implying that the consequence required for this sub-task is:

$$memory \circledast \neg vision \Rightarrow motor \tag{3.7}$$

Combining the condition and consequence for the second sub-task yields the full condition-consequence pair for the second sub-task:

$$vision \bullet \mathsf{\textbf{QUESTION}} \longmapsto memory \circledast \neg vision \Rightarrow motor \tag{3.8}$$

With the condition-consequence pairs for each sub-task, the reasoning system is fully configured to perform the simple question answering task. Figure 3.13 illustrates the complete system as well as its operation.

3.1.5 Inductive Reasoning in a Spiking Neural Network

The Raven's Progressive Matrix (RPM) model is a semantic pointer-based model designed specifically to demonstrate how semantic pointers can be used to perform the inductive reasoning necessary to solve the Raven's progressive matrices[Rasmussen, 2010]. A Raven's progressive matrix is a 3×3 grid of shapes specifically assembled with a relational dependence between them.

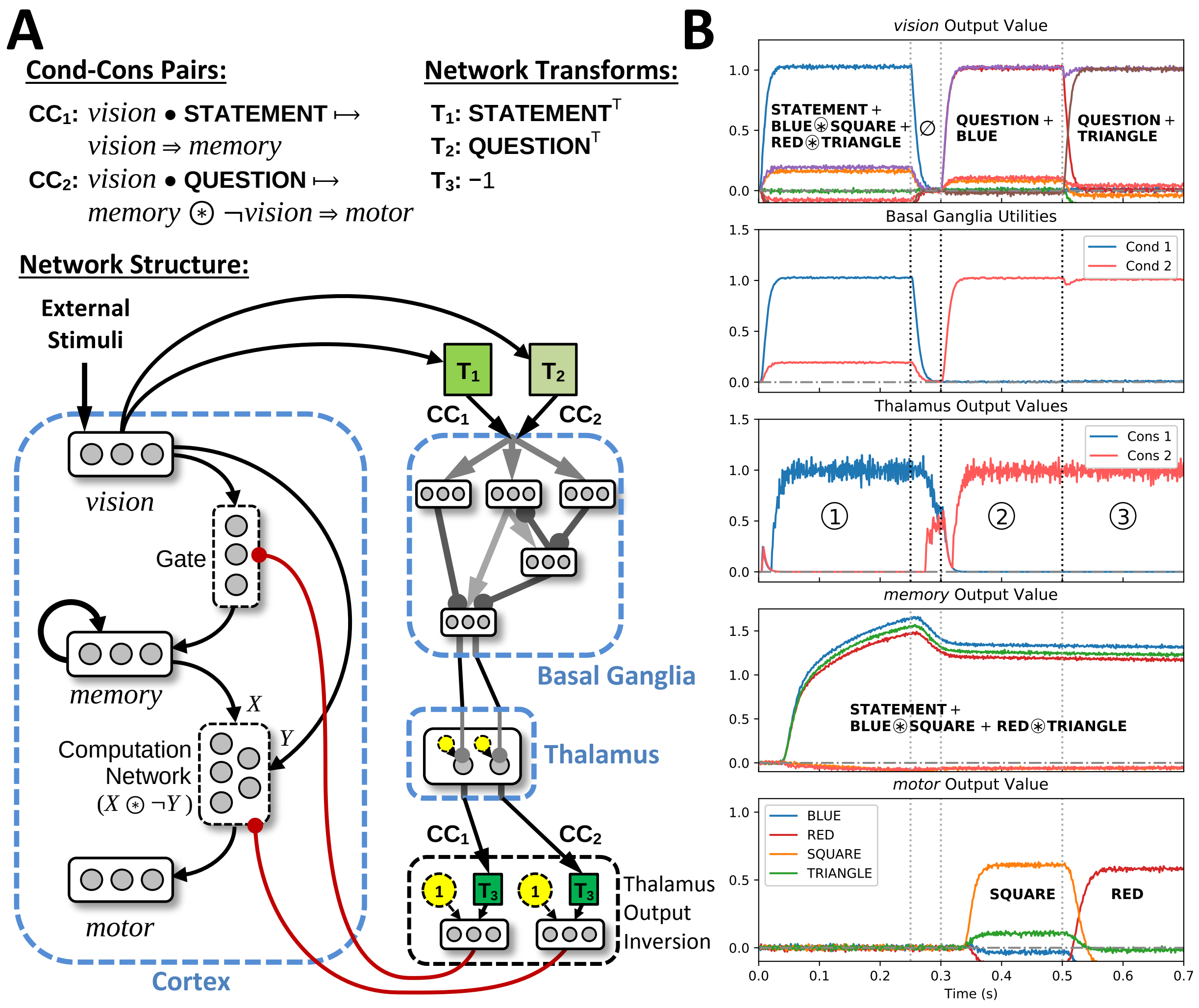


Figure 3.13: Schematic and operation of the reasoning system configured to perform the question answering task. (**A**) Schematic of the reasoning system configured to perform the question answering task. The network combines the static condition, dynamic consequence, and compound consequence sub-networks described previously. (**B**) Example operation of the question answering task. [①] To start, the network is presented (through "External Stimuli") with the statement "**STATEMENT** + **BLUE** ⊛ **SQUARE** + **RED** ⊛ **TRIANGLE**". This activates the first condition-consequence pair ("Cond 1" and "Cons 1"), resulting in the projection of the statement into the *memory* ensemble. [②] Here, the external stimulus is changed to "**QUESTION** + **BLUE**" ("What is *BLUE*?"), causing the activation of the second condition-consequence pair. This routes the semantic pointer stored in *memory* through the inverse binding computation ($X \circledast \neg Y$) network to the *motor* ensemble. The network responds correctly with "**SQUARE**". [③] Finally, the external stimulus is changed to "**QUESTION** + **TRIANGLE**", in order to demonstrate the network's ability to answer multiple (variable) questions. Here, the second condition-consequence pair remains active, and the network correctly responds with "**RED**".

The subject is tasked to choose the final (bottom-right cell) shape from a selection of 8 possible answers (see Figure 3.14A).

The relationships between each figure can be divided into three general categories: being part of a sequence, being part of a set, and being the result of the combination (or subtraction) of other figures in the matrix. Architecturally, the RPM model is divided into three individual networks, each targeted at solving each of the three aforementioned aspects of the inductive relations. (see Figure 3.14B). However, it is important to note that the networks share a common representational schema for describing the shapes in each cell of the progressive matrix.

As the Spaun model implements only the sequence solver, only it shall be discussed. Details on the set and figure solvers can be found in [Rasmussen, 2010]. The sequence solver presumes that the relationship between each adjacent cell can be described by a "transformation" semantic pointer representation $\mathbf{T}$, and that the semantic pointer representation of one cell can be computed by binding the semantic pointer representation of the previous cell with $\mathbf{T}$. In essence, given two adjacent cells, $\mathbf{CELL}_A$ followed by $\mathbf{CELL}_B$,

$$\mathbf{CELL}_B = \mathbf{CELL}_A \circledast \mathbf{T}. \tag{3.9}$$

To compute the transformation semantic pointer for the entire matrix, Equation (3.9) is reversed, calculated for each adjacent cell pair, and averaged across the entire matrix. Reversing Equation (3.9) reveals that each transformation $\mathbf{T}$ can be calculated as:

$$\mathbf{T} = \mathbf{CELL}_B \circledast \neg\mathbf{CELL}_A \tag{3.10}$$

Using the averaged transform $\mathbf{T}_{ave}$, the solution to the matrix can be found by binding the averaged transform with the cell adjacent to the missing cell.

3.1.6 Adaptive Spiking BG Model

This work combines a general purpose NEF learning rule [Stewart et al., 2012] with the basal ganglia model from Section 2.5.6.3. The basal ganglia network described in the previous section does not have a mechanism with which the basal ganglia input weights can be changed while the model is running. That is to say, once constructed, the possible conditions that trigger each basal ganglia action (condition-consequence pair) are fixed. The addition of the learning rule allows the basal ganglia actions to be altered with the use of an error signal. This is achieved by changing the connection weights between the cortex and basal ganglia according to the learning rule:

$$\Delta w_{ij} = \kappa \alpha_j \vec{\mathbf{e}}_j \mathbf{E} a_i, \tag{3.11}$$

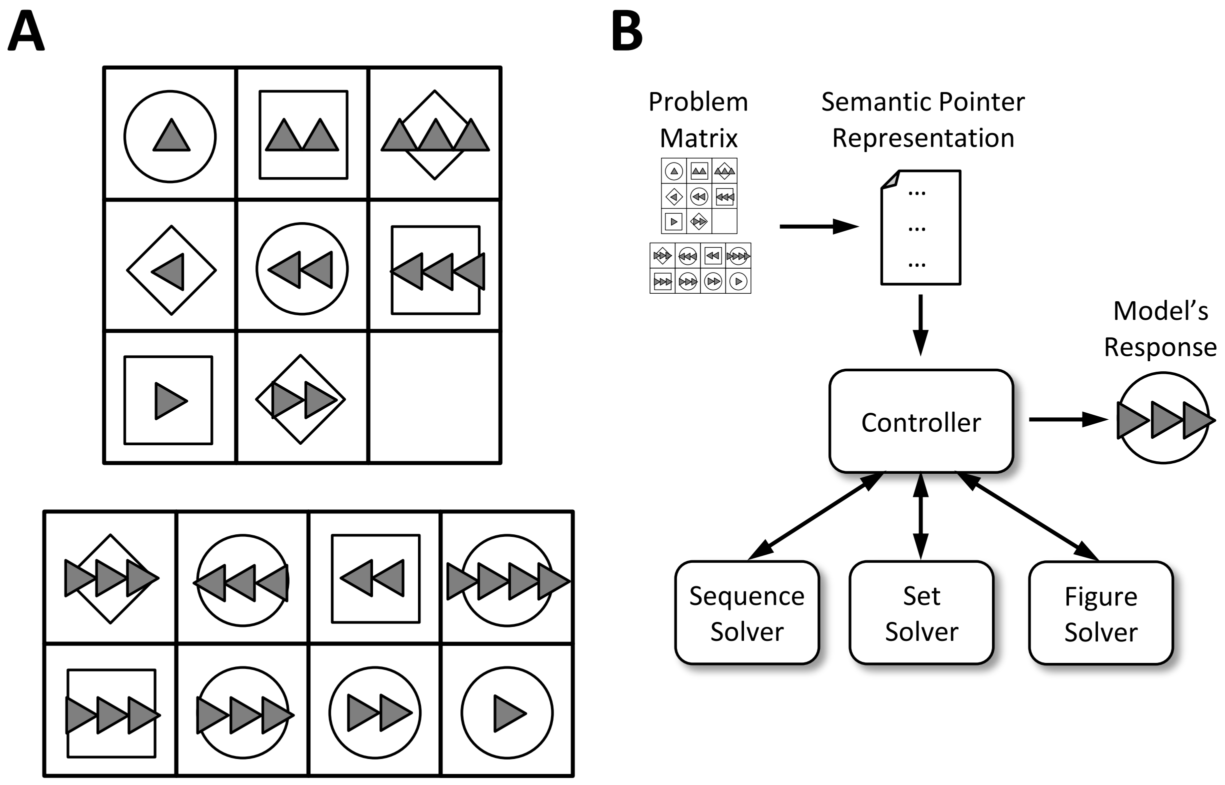


Figure 3.14: Example Ravens' style matrix and schematic of the Raven's progressive matrix solver model. (**A**) Example Raven's style matrix, consisting of a 3 × 3 grid of figures. The induction problem requires the subject to deduce the correct figure to fill in the empty cell, chosen from the 8 possible answers displayed just below the 3 × 3 matrix. (**B**) Schematic of the Raven's progressive matrix solver model. The problem matrix is manually encoded into a semantic pointer representation, which is provided to the controller. The controller is responsible for choosing the appropriate solver(s) to use, coordinating the flow of information to and from the solvers, and choosing the model's response from the list of provided answers. (Figures adapted from [Rasmussen, 2010] with permission.)

where w_{ij} is the connection weight between neuron i of the pre-synaptic population and j of the post-synaptic population, κ is the learning rate, α is the neuron gain (see Eq. (2.24)), $\vec{\mathbf{e}}$ is the neuron's preferred direction vector (see Eq. (2.29), a is the activity of the afferent neuron, and $\mathbf{E}$ is the error signal. Figure 3.15 illustrates the extra neural populations, and the additional neural projections necessary to combine the learning rule with the basal ganglia network.

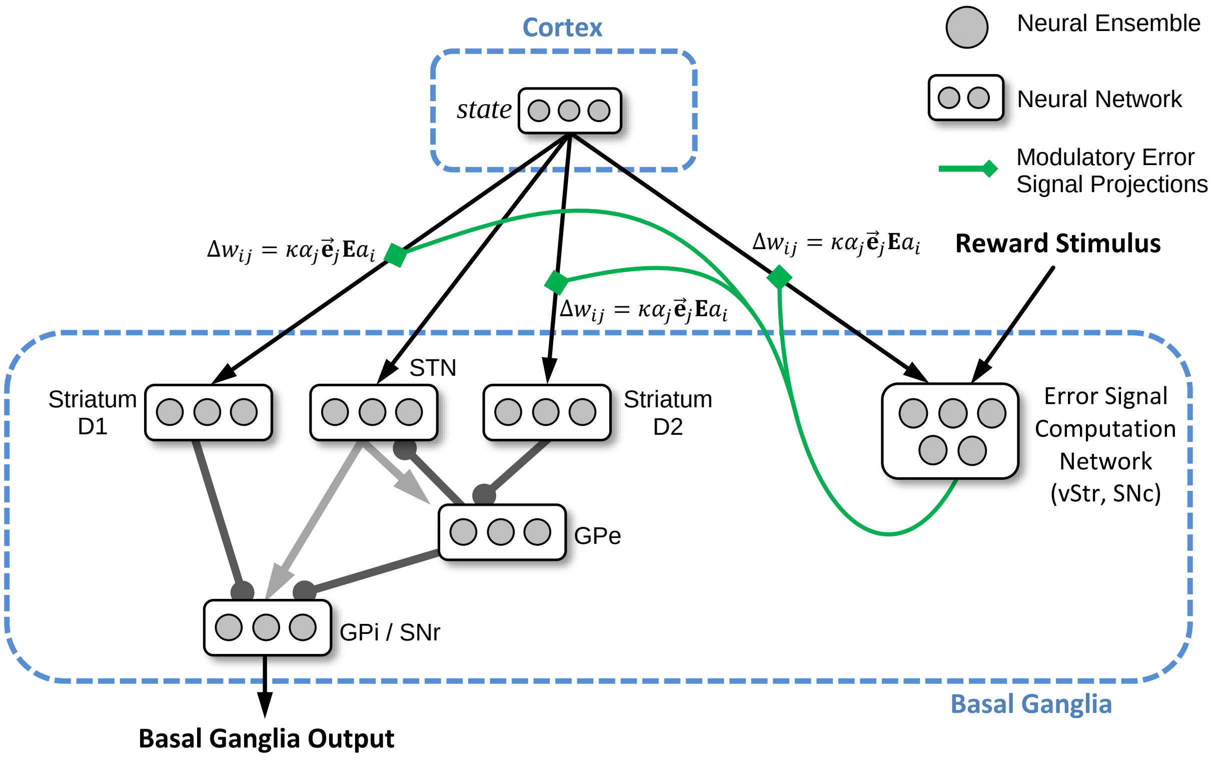

Cortex
state
Neural Ensemble
Neural Network
Modulatory Error Signal Projections
$\Delta w_{ij} = \kappa\alpha_j\vec{\mathbf{e}}_j\mathbf{E}a_i$
$\Delta w_{ij} = \kappa\alpha_j\vec{\mathbf{e}}_j\mathbf{E}a_i$
$\Delta w_{ij} = \kappa\alpha_j\vec{\mathbf{e}}_j\mathbf{E}a_i$
Reward Stimulus
STN
Striatum D1
Striatum D2
Error Signal Computation Network (vStr, SNc)
GPe
GPi / SNr
Basal Ganglia
Basal Ganglia Output

Figure 3.15: Schematic of the adaptive spiking basal ganglia model. To make the "standard" basal ganglia model adaptive, cortical information is combined with the reward stimulus to compute an error signal. This error signal is then used to modulate the weights for the neural projections between the cortical areas and the basal ganglia network. Biologically, this model hypothesizes that the error signal is computed through projections in the ventral striatum (vStr) and the substanstia nigra pars compacta (SNc), both of which are neural populations found in the basal ganglia. The vStr population receives information from cortex (*state*), and the reward stimulus to compute the reward values. These values are then projected to the SNc population that converts them to error signals, which are in turn projected back into cortex (in the figure these are the "modulatory error signal projections") through the doperminergic system.

3.2 Spaun's Tasks

Given the unique capabilities of each of Spaun's "precursor" models, 8 tasks were chosen to highlight the different cognitive abilities each model brought to Spaun, and to demonstrate that despite being vastly different in nature, using the SPA and the NEF allows all of the models to be integrated into a functional end-to-end cognitive system. The 8 tasks are:

Copy drawing: To demonstrate that the compressed semantic pointers generated by the visual system, and those used by the motor system retain enough feature information such that a for each digit, a generalized relationship between the two sets of semantic pointers can be found.

Digit recognition: To demonstrate that the vision system described in Section 3.1.1 can be successfully integrated and used in Spaun to recognize and reproduce the digits presented as inputs to Spaun.

***N*-arm bandit task:** To demonstrate that the adapting basal ganglia network (Section 3.1.6) can be integrated into Spaun and demonstrate the ability to adapt itself when provided the appropriate error feedback.

List memory: To demonstrate that after integration, the working memory model (Section 3.1.3) is still able to remember and recall lists of digits presented to Spaun.

Counting: To demonstrate that the basal ganglia network is able to perform an internally guided task (e.g., silent counting) in the context of a larger integrated system. A secondary goal is to demonstrate that Spaun to modify information stored within its memory system in order to accomplish this task.

Question answering: To demonstrate that Spaun's internal SPA representation is flexible enough to be probed for information using different types of queries.

Rapid variable creation: To demonstrate that Spaun has the ability to perform an induction task which involves finding the variable inputs amongst a set of static digits.

Fluid Induction: To demonstrate that Spaun has the ability of performing the pattern induction task similar to the sequential variants of the Raven's progressive matrices.

The following sections describe several constraints imposed on the design of Spaun, and demonstrate how the SPA has been used to perform the 8 cognitive tasks listed above.

3.2.1 Task Constraints

In addition to influencing the variety of tasks Spaun is designed to accomplish, the precursor models imposed several constraints on the design of Spaun itself. This section discusses these constraints, as well as the decisions made in order to facilitate the amalgamation of the precursor models as the Spaun network.

3.2.1.1 Input Constraints

The biggest constraint imposed on the Spaun network is the method by which information is presented to Spaun. Spaun's visual system – currently the primary method by which information is presented to Spaun – is based on the MNIST vision network described in Section 3.1.1. Since the construction and training of the MNIST vision network limits it to being able to classify images containing only a single handwritten digit, information presented to Spaun can consequently only be done one digit (character) at a time.

The single digit input constraint imposes yet another constraint on how information is fed to Spaun. With composited images (i.e., images containing multiple digits or figures), access to the multiple dimensions allows a greater depth of information to be transmitted. As an example, two groups of numbers can be made distinct simply by changing the spatial relation between the numbers within each group and space between each group. Using the single digit presentation style of Spaun however, additional digits (hereby referred to as "control characters") have to be used to relay this extra information. Figure 3.16 illustrates some examples of the control characters used in Spaun. The meaning and use of the control characters is discussed in the next section.

Because the MNIST dataset includes human handwriting, which is of varying quality, a set of "standardized" digits (generated using computer fonts) that the visual network is trained to classify correctly with high accuracy is also used. Figure 3.16 illustrates Spaun's "standardized" digits compared with examples of handwritten digits obtained using from the MNIST dataset.

3.2.1.2 Control Characters

As previously discussed, the Spaun's input character repertoire requires additional control characters in order to convey additional syntactic information to Spaun. Table 3.1 lists the common control characters (each task may have task-specific control characters) and their meaning as part of the task character sequence presented to Spaun.

3.2.1.3 Output Constraints

The constraints on Spaun's output depends on the implementation of Spaun's motor system. The NOCH motor system utilizes a concatenated set of trajectory points as its semantic pointer

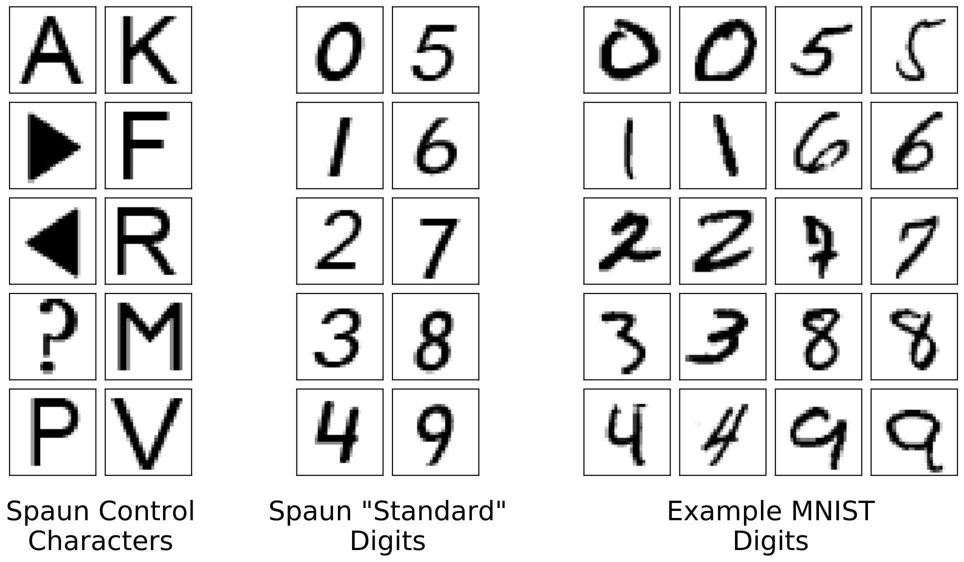


Figure 3.16: Example MNIST-based Spaun input stimuli, illustrating several Spaun control characters, the Spaun "standardized" digits, and examples from the MNIST dataset.

Pixel image	**Control character usage**
A	Used to denote the start of each task. It is followed by a digit indicating the specific task (out of the 8) to perform.
▶	Used to denote the start of a list of digits. This character may also be used to separate the task specification control sequence (see previous control character) from other task specific information.
◀	Used to denote the end of a list of digits.
?	Used indicate that an answer (motor response) is required from Spaun to complete the task.

Table 3.1: Table of common Spaun control characters and their usage in the Spaun input character sequences.

representation. This implies that Spaun's outputs must be constructed using a single unbroken trajectory. In addition, each set of trajectories is centered about a common origin. This means that tasks producing multiple character outputs will have each subsequent character in its output overwrite the preceding character.

3.2.1.4 Constraints on Internal SPA Representations

The use of the MNIST vision network as the method for classifying visual input limits Spaun's core conceptual repertoire to the set of numbers from 0 to 9. This implies that the task specific information for each task has to be numerical. In addition, the numerical range restriction means that the result of the counting task cannot exceed 9.[25]

In addition to the limiting Spaun's concepts to numerical value (0 to 9), it was decided to use the OSE's semantic pointer list representation as the basis for Spaun's internal cognitive representation. This decision implies that multiple digit sequences are treated as lists of digits rather than their numerical equivalent (e.g., "123" is considered the list "$[1, 2, 3]$" rather than the number "one hundred and twenty-three"). Section 3.2.5 discusses Spaun's list representation in greater detail. The use of a list representation also necessitates the implementation of additional cognitive processes required to convert the sequential stream of single characters into the list representation (see Section 3.3.3), and to translate the list representation into the sequentially written digits (see Section 3.3.4).

Regarding the dimensionality of the semantic pointers used in Spaun, across the entire model, semantic pointers are capped at 512 dimensions, with the particular implementation of each subsystem determining the dimensionality of its internal representation. The MNIST vision model described in Section 3.1.1 has an output dimensionality of 50, the NOCH motor model described in Section 3.1.2 has an input dimensionality of 54, and the internal cognitive representation of Spaun utilizes the maximum 512 dimensions.

3.2.2 Copy Drawing

Description: The copy drawing task requires Spaun to produce a motor output in the style of the presented digit. As an example, if Spaun were presented with a curly-bottomed "2" (as opposed to a 2 with a straight-edged bottom), Spaun's motor response should replicate the curled feature of the 2.

[25]From a semantic pointer perspective, the result of the counting task can exceed 9 without necessitating any changes to Spaun (see Section 3.2.6). However, because Spaun is only designed to output singular digits, a counting result exceeding 9 would produce an invalid output.

Task Syntax: The input character sequence for the copy drawing task is:

$$\mathbf{A}\ \mathbf{0}\ \blacktriangleright\ x\ \mathbf{?}$$

In the character sequence, "**A0**" informs Spaun that the task to be completed is the copy drawing task, and x is a place-holder for the MNIST digit desired for this task. Spaun is expected to produce the appropriate motor response *after* the question mark is shown.

Implementation: The SPA implementation of the copy drawing task makes following assumptions:

1. The visual system is capable of producing a visual semantic pointer $\mathsf{SP}_{\mathbf{VIS}}$ for the input digit images.
2. The motor system is capable of processing a motor semantic pointer $\mathsf{SP}_{\mathbf{MTR}}$ into the appropriate motor output.
3. For any given input digit image x, there is visual semantic pointer $\mathsf{SP}^{x}_{\mathbf{VIS}}$ produced by the visual system as a result of feeding it x, and a corresponding motor semantic pointer $\mathsf{SP}^{x}_{\mathbf{MTR}}$ that will generate a motor output replicating the path traced out by the digit in x.
4. A transform $\mathbf{T}$ can be found such that for a given visual semantic pointer $\mathsf{SP}_{\mathbf{VIS}}$, the motor semantic pointer $\mathsf{SP}_{\mathbf{MTR}}$ that best captures the features of the input digit can be found by computing $\mathsf{SP}_{\mathbf{MTR}} = \mathsf{SP}_{\mathbf{VIS}}\mathbf{T}$.

For Spaun, the goal is to find a generalized transformation that can be applied to any of the digits from the MNIST dataset. In order to do this, the transforms ($\mathbf{T}_0 \ldots \mathbf{T}_9$) is calculated for each one of the 10 digit classes (0 to 9) of the MNIST dataset. Computing the transform for each digit class then requires collecting $\mathsf{SP}_{\mathbf{VIS}}$ and $\mathsf{SP}_{\mathbf{MTR}}$ for multiple samples of that digit class from the MNIST dataset (preferably using samples that capture the wide range of digit styles found in the dataset) and computing the matrix inverse between the two sample sets.

As an example, to compute the transform $\mathbf{T}_2$ for the "2" digit class, five "2" images (v, w, x, y, and z) are chosen from the MNIST dataset. Next the visual semantic pointers $\mathsf{SP}^{v}_{\mathbf{VIS}}$ through $\mathsf{SP}^{z}_{\mathbf{VIS}}$ are generated by running each image through the visual network. After this, the motor semantic pointers $\mathsf{SP}^{v}_{\mathbf{MTR}}$ through $\mathsf{SP}^{z}_{\mathbf{MTR}}$ are determined such that running each motor semantic pointer through the motor system would produce outputs resembling the digits v to z.

Finally, by inverting the equation $\mathsf{SP}_{\mathbf{MTR}} = \mathsf{SP}_{\mathbf{VIS}}\mathbf{T}$, $\mathbf{T}_2$ can be calculated as:

$$\mathbf{T}_2 = \begin{bmatrix} \mathsf{SP}^v_{\mathbf{VIS}} \\ \mathsf{SP}^w_{\mathbf{VIS}} \\ \mathsf{SP}^x_{\mathbf{VIS}} \\ \mathsf{SP}^y_{\mathbf{VIS}} \\ \mathsf{SP}^z_{\mathbf{VIS}} \end{bmatrix}^{-1} \begin{bmatrix} \mathsf{SP}^v_{\mathbf{MTR}} \\ \mathsf{SP}^w_{\mathbf{MTR}} \\ \mathsf{SP}^x_{\mathbf{MTR}} \\ \mathsf{SP}^y_{\mathbf{MTR}} \\ \mathsf{SP}^z_{\mathbf{MTR}} \end{bmatrix}$$

System Requirements: For this task, Spaun is required to have:

- A visual network capable of producing $\mathsf{SP}_{\mathbf{VIS}}$.
- A motor network capable of utilizing $\mathsf{SP}_{\mathbf{MTR}}$
- When Spaun is prompted to produce its output, the input to the visual network is the **?** prompt, implying that the output of the visual network is $\mathsf{SP}^?_{\mathbf{VIS}}$ and not $\mathsf{SP}^x_{\mathbf{VIS}}$. As a consequence, a memory network is required to remember $\mathsf{SP}_{\mathbf{VIS}}$ for the last seen digit (i.e., the digit before the **?** prompt).
- Because a different transform matrix **T** is used for each of the digit classes, a separate mechanism is needed to identify the stored $\mathsf{SP}^x_{\mathbf{VIS}}$ in order to ensure that the appropriate transform is used in the $\mathsf{SP}_{\mathbf{MTR}}$ calculation. It is not possible to use the classifier allocated for the "live" digits because when Spaun is being prompted for its response, the classifier should be occupied with the task of classifying the "**?**" prompt.

3.2.3 Digit Recognition

Description: The digit recognition task requires Spaun to correctly classify a given MNIST digit into one of the ten (0 to 9) digit classes. The expected response is the Spaun's canonical representation of the numerical digit that represents each class (i.e., the "0" class should prompt Spaun to output a "0" in its own handwriting).

Task Syntax: The input character sequence for the digit recognition task is:

$$\mathbf{A}\ \mathbf{1}\ \blacktriangleright\ x\ \mathbf{?}$$

In the character sequence, "**A1**" informs Spaun that the task to be completed is the digit recognition task, and x is a placeholder for the MNIST digit desired for this task. Spaun is expected to produce the appropriate motor response *after* the question mark is shown.

Implementation: The Spaun implementation of the copy drawing task requires a separate classifier to classify the visual semantic pointer stored in memory. In theory, the output of this classifier can be used to perform the digit recognition task. However, the observation can be made that the digit recognition task is equivalent to performing the list memory task where the list consists of a single digit. Following this observation, no additional logic (apart from the logic required to perform the list memory task) is required.

System Requirements: Since the implementation of this task is identical to the list memory task, no additional requirements are needed.

3.2.4 N-arm Bandit Task

Description: The N-arm bandit task is a reinforcement learning task used to investigate animal models of learning (e.g., [Sutton and Barto, 1998; Kretchmar, 2002]). The task involves requiring the animal to make a choice of N (typically 2) actions. Based on the action chosen, the animal is rewarded according to the probability of rewarding the different actions. Generally, it is observed that the animal gravitates towards the action that provides the highest probability of reward, and crucially, when the reward probabilities for the different actions are changed, the animal is observed to be able to adapt to the changes and change their actions to consistently choose the one providing the highest probability of reward.

Task Syntax: In the case of Spaun, the N-arm bandit task requires it to choose one digit from 0 to $(N-1)$ to respond with. After Spaun responds with its action, it is presented with a "0" if no reward is given, and a "1" if it has been rewarded. Spaun is expected to continue producing responses as long as it has been prompted.

The input character sequence for the N-arm bandit task is:

$$\mathbf{A}\ \mathbf{2}\ \mathbf{?}\ r\ \mathbf{?}\ r\ \ldots$$

In the character sequence, "**A2**" informs Spaun that the task to be completed is the N-arm bandit task, "**?**" prompts Spaun to provide its chosen action, and r is a placeholder for the reward it is given ("0" for no rewards, "1" for having received a reward). The digit representing the reward value is only shown to Spaun after it has provided its action response.

Implementation: From the perspective of the SPA, the core learning mechanism required to accomplish the bandit task is already present in the adaptive basal ganglia model discussed in

Section 3.1.6. Spaun, however, requires additional circuitry to translate the "reward" digits ("0" or "1") into the error signals that the adaptive basal ganglia network requires. Additionally, the error value provided to the action that Spaun has chosen to perform differs from the error value provided to all of the other (unchosen) actions. Table 3.2 displays the error value ($\mathbf{E}_n$) that is applied to each action ($\mathbf{A}_n$) in the various scenarios of action / reward pairs for the 3-arm bandit task.

		Error Values		
Action Chosen	Reward Digit	$\mathbf{E}_1$	$\mathbf{E}_2$	$\mathbf{E}_3$
$\mathbf{A}_1$	0	1	-1	-1
	1	-1	1	1
$\mathbf{A}_2$	0	-1	1	-1
	1	1	-1	1
$\mathbf{A}_3$	0	-1	-1	1
	1	1	1	-1

Table 3.2: Table of mappings between actions chosen, reward stimulus input, and error values required for Spaun's 3-arm bandit task. Error signals are "-1" for the error values matching the chosen action, if and only if the chosen action is rewarded. Otherwise, the error signal is "1".

In Spaun, the error values are set such that if the chosen action is rewarded, a negative error[26] signal is provided for the corresponding condition-consequence pair of the basal ganglia network (thus making it more likely to be chosen again), and a positive error signal is provided to all other condition-consequence pairs (to suppress the likelihood of choosing those actions). The error values are reversed if not reward is obtained for the chosen action.

The computation of the error values ($\mathbf{E}_n$) is performed in two steps. First, intermediary values a_n and r are generated based on the chosen action, and the reward digit received, respectively. Next, using the observation that if a_n is given the value 1 if the chosen action matches the n subscript, and -1 otherwise (i.e., if $\mathbf{A}_1$ is chosen, $a_1 = 1$, and $a_2 = a_3 = \ldots = a_n = -1$), and if r is given the value 1 if a reward is received and -1 otherwise, the value of $\mathbf{E}_n$ can be computed as $\mathbf{E}_n = a_n \times (1 - r)$. Table 3.3 demonstrates this calculation for the 3-arm bandit task.

Using the SPA, the values of a_n and r can be computed using the vector dot product and the appropriate use of additions and subtractions. Assuming that the basal ganglia represent the chosen action with the semantic pointer $\mathbf{SP_{ACT}}$, and that each one of the possible actions is

[26]In Spaun, the term "error" implies that an incorrect action has been taken, and thus, a positive error signal penalizes the chosen action.

Action Chosen	a Values a_1	a_2	a_3	Reward Digit	r	Error Values $(a_n \times (1-r))$ $\mathbf{E}_1$	$\mathbf{E}_2$	$\mathbf{E}_3$
$\mathbf{A}_1$	1	-1	-1	0	-1	1	-1	-1
	1	-1	-1	1	1	-1	1	1
$\mathbf{A}_2$	-1	1	-1	0	-1	-1	1	-1
	-1	1	-1	1	1	1	-1	1
$\mathbf{A}_3$	-1	-1	1	0	-1	-1	-1	1
	-1	-1	1	1	1	1	1	-1

Table 3.3: Table demonstrating the intermediary values and computation required to produce the error values listed in Table 3.2.

represented by the semantic pointer $\mathsf{A}N$ (i.e., action $\mathbf{A}_1$ is represented by the semantic pointer $\mathsf{A1}$), using the 3-arm bandit task example, a_n can be computed for a given action semantic pointer using:

$$\begin{aligned}
a_1 &= \mathsf{SP}_{\mathbf{ACT}} \bullet (\mathsf{A1} - \mathsf{A2} - \mathsf{A3}) \\
a_2 &= \mathsf{SP}_{\mathbf{ACT}} \bullet (-\mathsf{A1} + \mathsf{A2} - \mathsf{A3}) \\
a_3 &= \mathsf{SP}_{\mathbf{ACT}} \bullet (-\mathsf{A1} - \mathsf{A2} + \mathsf{A3})
\end{aligned}$$

As an example, if $\mathsf{SP}_{\mathbf{ACT}} = \mathsf{A2}$,

$$\begin{aligned}
a_1 &= \mathsf{A2} \bullet (\mathsf{A1} - \mathsf{A2} - \mathsf{A3}) = (\mathsf{A2} \bullet \mathsf{A1}) - (\mathsf{A2} \bullet \mathsf{A2}) - (\mathsf{A2} \bullet \mathsf{A3}) \\
&\approx 0 - 1 - 0 = -1 \\
a_2 &= \mathsf{A2} \bullet (-\mathsf{A1} + \mathsf{A2} - \mathsf{A3}) = -(\mathsf{A2} \bullet \mathsf{A1}) + (\mathsf{A2} \bullet \mathsf{A2}) - (\mathsf{A2} \bullet \mathsf{A3}) \\
&\approx 0 + 1 - 0 = 1 \\
a_3 &= \mathsf{A2} \bullet (-\mathsf{A1} - \mathsf{A2} + \mathsf{A3}) = -(\mathsf{A2} \bullet \mathsf{A1}) - (\mathsf{A2} \bullet \mathsf{A2}) + (\mathsf{A2} \bullet \mathsf{A3}) \\
&\approx 0 - 1 + 0 = -1
\end{aligned}$$

In a similar fashion, if each of the possible reward values are represented by the semantic pointers **REWARD** (rewarded) and **NO_REWARD** (non-rewarded), the value of r can be computed by performing the vector dot product of the visual semantic pointer ($\mathsf{SP}_{\mathbf{VIS}}$) with the combination of the reward semantic pointers, i.e.,

$$r = \mathbf{SP_{VIS}} \bullet (\mathbf{REWARD} - \mathbf{NO\_REWARD})$$

System Requirements: For this task, Spaun is required to have:

- A visual network capable of producing $\mathbf{SP_{VIS}}$.
- An adaptive reasoning system (i.e., the reasoning system that uses the adaptive basal ganglia network) configured with the appropriate condition-consequence pairs to output the action semantic pointers ($\mathbf{SP_{ACT}}$) when the particular action is chosen.
- A neural network to calculate the error values ($\mathbf{E}_n$) following the steps described previously.

3.2.5 List Memory

Description: The list memory task requires Spaun to remember the list of digits it is presented, and recall the list (in the order it was presented) when prompted. The digits can either be from the MNIST data set, or from the set of "standardized" digits.

Task Syntax: The input character sequence for the list memory task is:

$$\mathbf{A}\ \mathbf{3}\ \blacktriangleright\ x_1\ x_2\ \ldots\ x_n\ \blacktriangleleft\ \mathbf{?}$$

In the character sequence, "**A3**" informs Spaun that the task to be completed is the list memory task, and x_n are placeholders for the desired task-specific digit inputs. The triangular symbols denote the boundaries of the list.

Implementation: Spaun's SPA list representation is based on the OSE list representation (see Section 3.1.3). That is to say that the SPA representation used within the working memory network of Spaun is of the form:

$$\mathbf{MEMORY} = \mathbf{POS1} \circledast \mathbf{DIGIT1} + \mathbf{POS2} \circledast \mathbf{DIGIT2} + \ldots + \mathbf{POS}N \circledast \mathbf{DIGIT}N$$

However, while the position semantic pointer tags ($\mathbf{POS}N$) in the OSE model are randomly generated, that approach is not feasible for the Spaun model, as there is no limit on the number of items (in a list) that Spaun can be presented. Instead, an unbounded method is required for generating the position semantic pointers. In Spaun this is accomplished with the use of two randomly generated unitary semantic pointers (see Section 2.3.5), one to represent the first item's position (**POS1**) and another to serve as an increment operation (**INC**). With these two unitary semantic pointers, the rest of the position semantic pointers can be generated using the following recursive algorithm:

$$\mathsf{\mathbf{POS}}N = \mathsf{\mathbf{POS}}(N-1) \circledast \mathsf{\mathbf{INC}} \tag{3.12}$$

As an example,

$$\mathsf{\mathbf{POS2}} = \mathsf{\mathbf{POS1}} \circledast \mathsf{\mathbf{INC}}$$
$$\mathsf{\mathbf{POS3}} = \mathsf{\mathbf{POS2}} \circledast \mathsf{\mathbf{INC}}$$

Another challenge posed by the sequential nature of Spaun's input is the process necessary to "build" the list representation as digits are presented to Spaun. Conveniently, because the list representation is a collection of bound pairs, as digits are presented to Spaun, the list representation can be constructed by adding the bound result of the current digit and position semantic pointers with the list representation stored in working memory:

$$\mathsf{\mathbf{MEMORY}} = \mathsf{\mathbf{MEMORY}} + \mathsf{\mathbf{POS}}N \circledast \mathsf{\mathbf{DIGIT}}N \tag{3.13}$$

where $\mathsf{\mathbf{POS}}N$ is the position semantic pointer of Spaun's current input, and $\mathsf{\mathbf{DIGIT}}N$ is the digit semantic pointer of Spaun's current input. If the item being presented is the first item in the list, $\mathsf{\mathbf{MEMORY}} = \emptyset$ and $\mathsf{\mathbf{POS}}N = \mathsf{\mathbf{POS1}}$.

System Requirements: For this task, Spaun is required to have:

- A visual network capable of producing $\mathsf{\mathbf{SP_{VIS}}}$, and transforming the visual semantic pointer into its cognitive equivalent $\mathsf{\mathbf{DIGIT}}N$.
- A neural network responsible for the recursive generation of the position semantic pointer $\mathsf{\mathbf{POS}}N$. It should be noted that this network is required to keep track of the "current" position semantic pointer as it is necessary for the computation of the subsequent position semantic pointers.
- A separate working memory network to store the list representation.

3.2.6 Counting

Description: The counting task requires Spaun to, given a starting digit and a desired count, silently (internally) count up from the starting digit by the desired number of counts. Spaun is required to start this counting process only when prompted, and is expected to produce a written digit response upon completion of the silent counting process.

Task Syntax: The input character sequence for the counting task is:

$$\mathbf{A}\ \mathbf{4}\ \blacktriangleright\ x\ \blacktriangleleft\ \blacktriangleright\ y\ \blacktriangleleft\ \mathbf{?}$$

In the character sequence, "**A4**" informs Spaun that the task to be completed is the counting task, the first list containing the digit x is the starting digit, and the second list containing the digit y is the number of counts Spaun is required to perform. Both the first and second list should only contain a single digit, and should sum to a maximum value of 9 (for Spaun to produce valid outputs).

Implementation: To perform the counting task using the SPA, a semantic relationship must first be established between the semantic pointer representations of Spaun's cognitive concept of the numerical digits.

In Spaun, a method similar to that used in the generation of the list position semantic pointers is used to construct this semantic relationship. In essence, two randomly generated unitary semantic pointers are used to represent the concepts of "zero" and "add one", **ZERO** and **ADD1**, respectively. With these two semantic pointers, the remaining numerical semantic pointers can be generated by binding **ZERO** with the n^{th} bound power of **ADD1**, where n is the desired number. As an example,

$$\begin{aligned}\mathbf{ONE} &= \mathbf{ZERO} \circledast \mathbf{ADD1}^1 = \mathbf{ZER} \circledast \mathbf{ADD1}\\ \mathbf{TWO} &= \mathbf{ZERO} \circledast \mathbf{ADD1}^2 = \mathbf{ZER} \circledast \mathbf{ADD1} \circledast \mathbf{ADD1}\\ &\vdots\\ \mathbf{NINE} &= \mathbf{ZERO} \circledast \mathbf{ADD1}^9 \end{aligned} \tag{3.14}$$

It is important to note that this semantic relationship (e.g., "two" is equal to "zero add one, add one") not only holds true between the concept of "zero" and the rest of the numerals, but also between any two arbitrary numerals:

$$\begin{aligned}\mathbf{TWO} &= \mathbf{ZERO} \circledast \mathbf{ADD1}^2\\ &= \mathbf{ZER} \circledast \mathbf{ADD1} \circledast \mathbf{ADD1}\\ &= (\mathbf{ZER} \circledast \mathbf{ADD1}) \circledast \mathbf{ADD1}\\ &= \mathbf{ONE} \circledast \mathbf{ADD1} \quad \text{(i.e., "One add one")}\end{aligned}$$

With the relationship between each number established, the core algorithm of the counting task can be discussed. In the counting task, two pieces of information are provided to Spaun:

the starting number, and the number of counts to perform. To complete this task, however, Spaun is required to remember three pieces of information: the number of counts to perform (**NUM_COUNT**), the current count result (**RESULT**), and the number of counts already performed (**CURR_COUNT**).

When the counting task is started, **RESULT** is initialized to the starting number provided, **NUM_COUNT** is set to the desired number of counts, and **CURR_COUNT** is initialized to zero. It is important to note that in order to reduce the complexity of the information encoding and retrieval logic of the working memory network, Spaun represents all of three pieces of information required for the counting task as single item lists. As an example, if Spaun is tasked to start at the number 5, and count by 2, at the start of the count task,

$$\begin{aligned}\mathbf{RESULT} &= \mathbf{POS1} \circledast \mathbf{FIVE}\\ \mathbf{NUM\_COUNT} &= \mathbf{POS1} \circledast \mathbf{TWO}\\ \mathbf{CURR\_COUNT} &= \mathbf{POS1} \circledast \mathbf{ZERO}\end{aligned}$$

Spaun then proceeds with the counting task by iteratively binding **ADD1** to both **RESULT** and to **CURR_COUNT**. Spaun stops the counting process when the value of **CURR_COUNT** is equal to that of **NUM_COUNT**. Table 3.4 demonstrates this process for the counting example above.

Semantic Pointer	Start	Step 1	Step 2 (Stop)
RESULT	**POS1** ⊛ **FIVE**	**POS1** ⊛ **SIX**	**POS1** ⊛ **SEVEN**
NUM_COUNT	**POS1** ⊛ **TWO**	**POS1** ⊛ **TWO**	**POS1** ⊛ **TWO**
CURR_COUNT	**POS1** ⊛ **ZERO**	**POS1** ⊛ **ONE**	**POS1** ⊛ **TWO**

Table 3.4: Table detailing the individual steps required for Spaun to perform the (internal) counting task. Shown are the intermediary values for each step of the counting process for the three semantic pointer values **RESULT**, **NUM_COUNT**, and **CURR_COUNT**. Refer to the main text for the use of each of these semantic pointers in the counting task.

System Requirements: For this task, Spaun is required to have:

- At least three working memory networks, each one to keep track of the values of **RESULT**, **NUM_COUNT**, and **CURR_COUNT** respectively.

- Neural circuitry capable of performing the **ADD1** binding operation with the values in working memory (and subsequently storing them back into working memory).
- A mechanism of internally driving the (silent) counting process without the need for external interference.
- A neural network (dot product network) to perform the comparison between the values of **NUM_COUNT** and **CURR_COUNT**, needed to stop the counting process.

3.2.7 Question Answering

Description: The question answering task requires Spaun to answer two possible questions after being provided with a list of digits. Spaun can be prompted to provide the digit found a specified position of the list, or Spaun can be asked to figure out where in the list a particular digit is located. For simplicity, it is assumed that for the latter question, Spaun is presented a list without duplicate digits.

Task Syntax: As two different questions can be asked of Spaun, the input character sequence for the question answering task comes in two variants. To prompt Spaun for the digit in a specific position, the character sequence is:

$$\textbf{A 5} \;\blacktriangleright\; x_1\, x_2\, \ldots\, x_n \;\blacktriangleleft\; \textbf{P} \;\blacktriangleright\; y \;\blacktriangleleft\; \textbf{?}$$

Likewise, to prompt Spaun for the position of a specific digit, the character sequence is:

$$\textbf{A 5} \;\blacktriangleright\; x_1\, x_2\, \ldots\, x_n \;\blacktriangleleft\; \textbf{K} \;\blacktriangleright\; y \;\blacktriangleleft\; \textbf{?}$$

As in the list memory task, x_n are placeholders for the desired list entries. Specific to the question answering task, the character **P** is used to denote the position prompt, whereas the character **K** is used to denote a query for a desired kind (digit). Additionally, y is a placeholder for the digit representing the desired value for the position or kind query.

Implementation: Because the list representation used in Spaun is a collection of bound pairs, the question answering task can be accomplished by binding the list representation with the inverse of the query prompt, similar to the method used in Equation (2.6). As an example, for the list $[2, 4, 1]$, i.e., $\mathbf{MEMORY} = \mathbf{POS1} \circledast \mathbf{TWO} + \mathbf{POS2} \circledast \mathbf{FOUR} + \mathbf{POS3} \circledast \mathbf{ONE}$, querying for the digit in the second position of the list requires the SPA operation:

$$\mathbf{MEMORY} \circledast {\sim}\mathbf{POS2} \approx \mathbf{FOUR} \tag{3.15}$$

Similarly, querying the list representation for the position of the digit “1” requires the SPA operation:

$$\textbf{MEMORY} \circledast \sim\textbf{ONE} \approx \textbf{POS3} \tag{3.16}$$

Analysis of the query mechanism, however, reveals an issue with directly using this method for querying the list representation. Since all numerical inputs provided to Spaun are encoded as lists, querying Spaun for the digit in the second results in Spaun storing “**POS1** ⊛ **TWO**” in working memory as the query for both the position and kind prompts, as opposed to required semantic pointers “**POS2**” and “**TWO**” respectively. In addition, the result of the inverse binding operation (Eq. (3.15) and (3.16)) are not list representations, meaning that the motor system will not be able to use the results directly to produce Spaun’s output.

Both of these issues can be addressed with the use of associative memory networks (see Section 2.5.5.5) to map the list representations to the appropriate numerical or position semantic pointers, and to map the output of the inverse binding operation to the appropriate list representations. This network implementation is illustrated in Figure 3.17 with the example query for either the second list position, or for the digit “2”.

System Requirements: For this task, Spaun is required to have:

- At least two working memory networks, one to store the list representation, and one to store the query probe value.
- A binding network to perform the inverse binding task.
- Associative memory networks to perform the four mapping operations (to and from the list representation onto either the digit or position representations) discussed previously.

3.2.8 Rapid Variable Creation

Description: The rapid variable creation task is a pattern induction task proposed by Hadley [2009] as a task for which he argues is impossible for neural networks to solve.

The rapid variable creation task involves the presentation of training set of input-output pairs that have been constructed with a specific pattern of constant and variable mappings, and requires the system to produce the correct output mapping given a novel input stimulus. Hadley illustrates this task with the example (with the expected answer indicated in parenthesis):

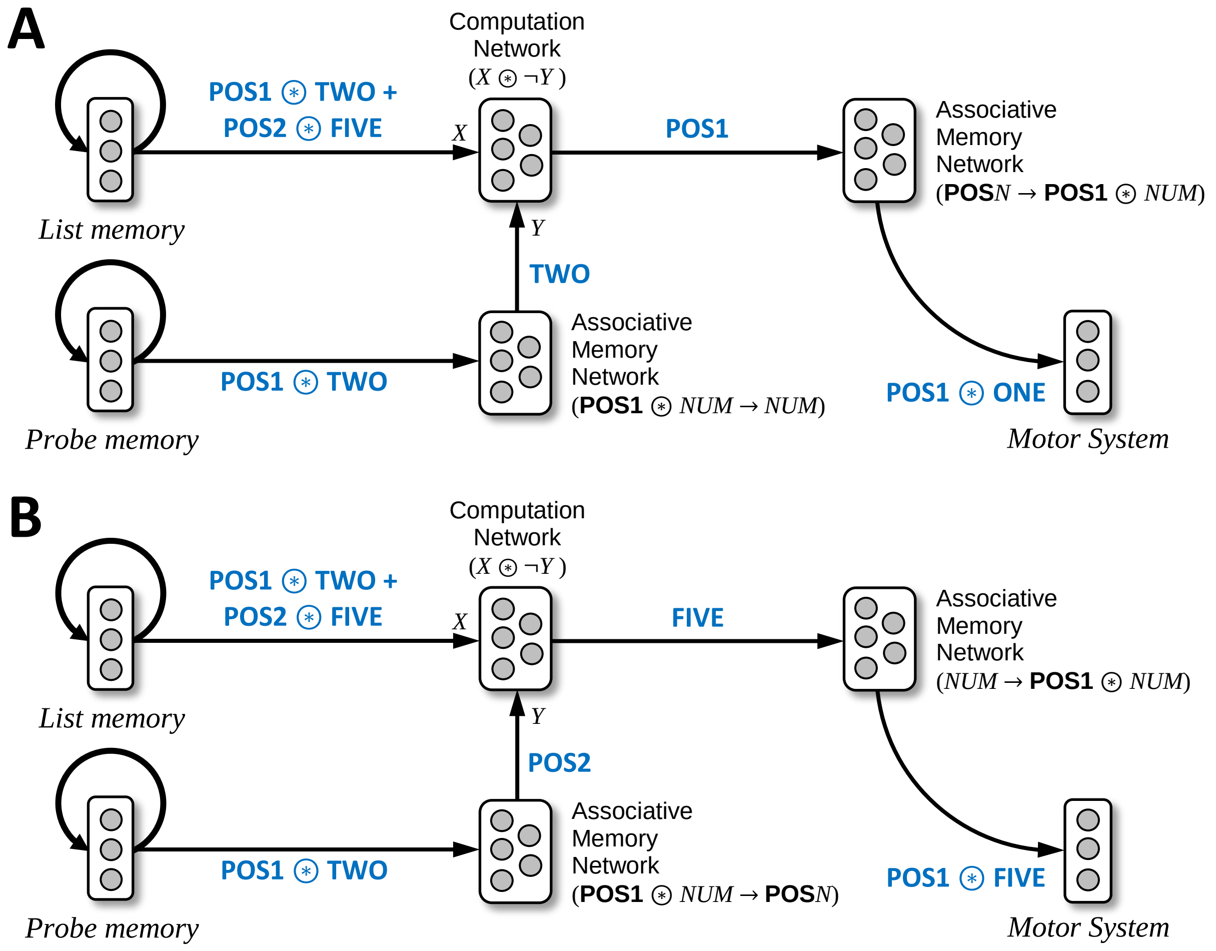

A
Computation Network ($X \circledast \neg Y$)
POS1 ⊛ TWO + POS2 ⊛ FIVE
X
POS1
Associative Memory Network (POSN → POS1 ⊛ NUM)
List memory
Y
TWO
Associative Memory Network (POS1 ⊛ NUM → NUM)
POS1 ⊛ TWO
POS1 ⊛ ONE
Probe memory
Motor System
B
Computation Network ($X \circledast \neg Y$)
POS1 ⊛ TWO + POS2 ⊛ FIVE
X
FIVE
Associative Memory Network (NUM → POS1 ⊛ NUM)
List memory
Y
POS2
Associative Memory Network (POS1 ⊛ NUM → POSN)
POS1 ⊛ TWO
POS1 ⊛ FIVE
Probe memory
Motor System

Figure 3.17: Network schematic and example operation for the "position" ("what is at position X?") and "kind" ("in what position is digit X?") variants of Spaun's question answering task. The network implementations for both variants of the question answering task are identical with the exception of the mappings performed by the associative memory networks. (**A**) Network schematic for the "kind" variant of Spaun's question answering task. In Spaun, both pieces of information are encoded as list representations, and associative memory networks are used to map the list representation of the probe semantic pointer into the appropriate digit semantic pointer (**POS1** $\circledast$ $NUM \rightarrow NUM$). An associative memory network is also required to map Spaun's answer (which is a position semantic pointer) back to the list representation (**POS**$N \rightarrow$ **POS1** $\circledast$ NUM) so that the motor system is able to parse and output the correct answer. Displayed in blue text are the semantic pointer representations at each step of the network. (**B**) Network schematic for the "position" variant of Spaun's question answering task. As in the "kind" variant, an associative memory network is required to map the probe list representation into the appropriate semantic pointer. For the position variant, the mapping is done from the list representation into a position semantic pointer (**POS1** $\circledast$ $NUM \rightarrow$ **POS**N). An associative memory network is also required to map Spaun's answer (which is a numerical semantic pointer) back to the list representation ($NUM \rightarrow$ **POS1** $\circledast$ NUM). Displayed in blue text are the semantic pointer representations at each step of the network.

	Input	Output
Training Set	Biffle biffle rose zarple	rose zarple
	Biffle biffle frog zarple	frog zarple
	Biffle biffle dog zarple	dog zarple
Test Set	Biffle biffle quoggio zarple	? (*quoggio zarple*)

In Hadley's example, the first, second and fourth item of the input lists are constants, while the third item is variable (hence the name of the task), while the output lists should contain only the third and fourth items from the input list. The task is "rapid" as the expected time for the system to produce an output is on the order of seconds, compared to the tens of trials typical learning tasks require.

As Spaun operates completely in the domain of numerical digits, the Spaun equivalent of the rapid variable creation task is:

	Input	Output
Training Set	9 9 5 4	5 4
	9 9 3 4	3 4
	9 9 7 4	7 4
Test Set	9 9 2 4	? (*2 4*)

It should be noted that the rapid variable creation tasks for Spaun do not necessarily have to follow the exact syntax described above. The only requirement for Spaun's rapid variable creation task is that both the input and output lists should contain at least one variable.

Task Syntax: The input character sequence for the rapid variable creation task is of the form:

$$\textbf{A 6} \ \blacktriangleright \ IN_1 \ \blacktriangleleft \ \blacktriangleright \ OUT_1 \ \blacktriangleleft \ \ldots \ \blacktriangleright \ IN_N \ \blacktriangleleft \ \blacktriangleright \ OUT_N \ \blacktriangleleft \ \blacktriangleright \ TEST \ \blacktriangleleft \ \textbf{?}$$

In the character sequence, the training set is provided to Spaun as alternating pairs of inputs (IN_N) and output (OUT_N) lists. The test set ($TEST$) is provided to Spaun as the last list before the prompt for an answer.

Implementation: None of the precursor models described in the previous section were specifically constructed to perform the rapid variable creation task. However, the Spaun model hypothesizes that the ordinal-based structure of the list representation of the OSE model (Section 3.1.3) can be combined with the sequence solver from the Raven's progressive matrix model (Section 3.1.5) to perform the rapid variable creation task.

Spaun's working memory network already inherently remembers information as lists of numbers, and therefore, performing the rapid variable creation task simply involves using two working memory networks to store the input (**IN**) and output (**OUT**) lists of each training pair. Following the method used in the sequence solver, the transform **T** can then be calculated as $\mathbf{T} = \mathbf{OUT} \circledast \sim\mathbf{IN}$. Following the presentation of the entire training set, the average of all **T** transforms can be averaged ($\mathbf{T}_{ave}$), and bound to the test list (**TEST**) to compute Spaun's answer for the rapid variable creation task: $\mathbf{ANSWER} = \mathbf{TEST} \circledast \mathbf{T}_{ave}$.

An issue arises, however, in the calculation of $\mathbf{T}_{ave}$: the number of pairs in the training set is not known until the answer prompt is provided. Additionally, even if Spaun were capable of remembering the number of pairs it has been shown, calculating the average through the use of a division is ill-advised as the neural (NEF) implementation of the division operator is accurate over a limited range of values for the divisor (see Section 4.4). For these reasons, Spaun uses the following moving average calculation to compute $\mathbf{T}_{ave}$:

$$\mathbf{T}_{ave} = \alpha \times \mathbf{T}_N + (1 - \alpha) \times \mathbf{T}_{ave} \tag{3.17}$$

For the Spaun model, the value of α was set to 0.275 as this minimized the difference between the value of the true average and the value of the moving average for the averages of 5 items[27] (see Figure 3.18).

System Requirements: For this task, Spaun is required to have:

- At least two working memory networks, one to store the training input list representation, and one to store the training output list representation.
- One working memory network for the computation of the moving average of the transform **T**.
- A binding network to perform the inverse binding operation necessary to compute the transform **T**.[28]
- A binding network to perform the binding operation necessary to compute Spaun's answer.[28]

[27] 5 items was chosen as it is the maximum number of transforms averaged for the RVC task, and the fluid induction task (see Section 3.2.9).

[28] It should be noted that a single binding network (with properly configured inputs) can be used to perform both the inverse binding operation and the answer binding operation.

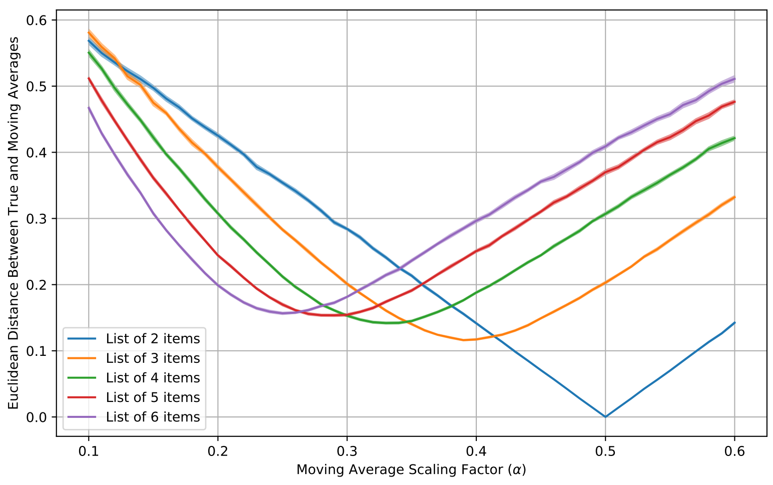


Figure 3.18: Comparison of the Euclidean differences between the true and moving averages for different average scaling factors (α). The plots are generated using 512-dimensional semantic pointers for lists varying in length from 2 to 6 items. Equation (3.17) is used to calculate the moving averages. Shaded regions indicate the 95% confidence intervals for each plot.

3.2.9 Fluid Induction

Description: Spaun's fluid induction task is based on the sequence variant of the Raven's progressive matrices, i.e., Spaun is required to solve a 3×3 matrix of entries that have been constructed to conform to a specific sequential pattern. The sequence relationship used to construct the matrix can be one of two types, either the numerical (digit) difference between each cell can be sequential, or the number of list items between each cell can be sequential. Figure 3.19 demonstrates examples of these two types of sequential matrix patterns.

Task Syntax: The Raven's progressive matrix task is typically presented as a 3×3 matrix. However, because Spaun is only able to process sequences of digits, the matrix is "flattened" to form a sequence of eight lists, with each list representing the configuration of each cell. Thus, a matrix of the form

C_1	C_2	C_3
C_4	C_5	C_6
C_7	C_8	?

is presented to Spaun as a character sequence of the form:

$$\textbf{A 7} \blacktriangleright C_1 \blacktriangleleft \blacktriangleright C_2 \blacktriangleleft \blacktriangleright C_3 \blacktriangleleft \blacktriangleright C_4 \blacktriangleleft \blacktriangleright C_5 \blacktriangleleft \blacktriangleright C_6 \blacktriangleleft \blacktriangleright C_7 \blacktriangleleft \blacktriangleright C_8 \blacktriangleleft \textbf{?}$$

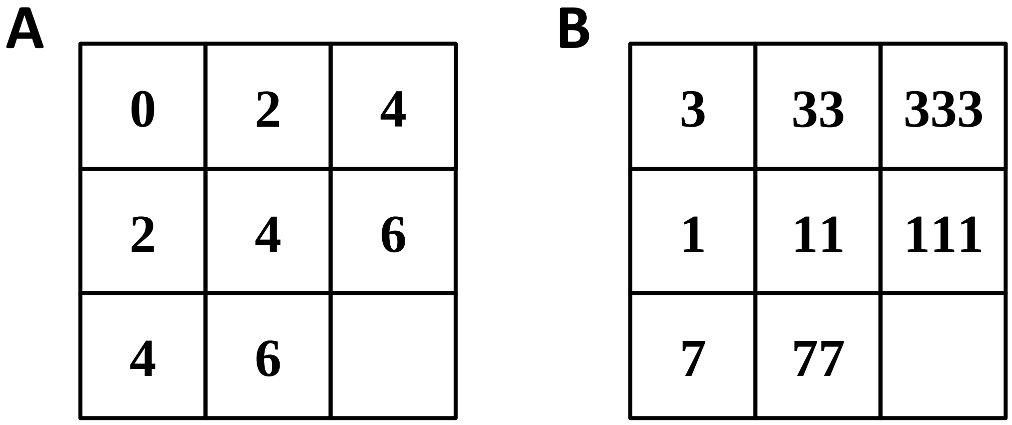


Figure 3.19: Examples of the two forms of sequential sequence matrix patterns for Spaun's fluid induction task. (**A**) Example matrix pattern where the sequential pattern is applied to the numerical difference between the numbers of each cell. (**B**) Example matrix pattern where the sequential pattern is applied to the number of list items (individual digits) present in each cell.

Implementation: The mechanism Spaun employs to perform the fluid induction task is essentially identical to that used for the rapid variable creation task with the exception of what inputs are used to calculate the transform $\mathbf{T}$. As an example, in the rapid variable creation task, the first two $\mathbf{T}$ transforms are calculated using the first and second input/output training pairs:

$$\mathbf{T}_1 = \mathbf{OUT}_1 \circledast {\sim}\mathbf{IN}_1$$
$$\mathbf{T}_2 = \mathbf{OUT}_2 \circledast {\sim}\mathbf{IN}_2$$

In contrast, the first two $\mathbf{T}$ transforms for the fluid induction task involve only the cells in the first row of the matrix:

$$\mathbf{T}_1 = \mathbf{CELL}_2 \circledast {\sim}\mathbf{CELL}_1$$
$$\mathbf{T}_2 = \mathbf{CELL}_3 \circledast {\sim}\mathbf{CELL}_2$$

It is important to note that Spaun's "understanding" of the structure of the matrix is implemented through the inputs provided to the transform calculation. As an example, the nature of the matrix implies that the last cell of the first row and the first cell of the second row may not have any relation, and thus, the third $\mathbf{T}$ transform for the fluid intelligence task should be between the fourth and fifth cells (instead of the third and fourth cells). It follows then that:

$$\mathbf{T}_3 = \mathbf{CELL}_5 \circledast \sim\mathbf{CELL}_4$$
$$\mathbf{T}_4 = \mathbf{CELL}_6 \circledast \sim\mathbf{CELL}_5$$
$$\mathbf{T}_5 = \mathbf{CELL}_8 \circledast \sim\mathbf{CELL}_7$$

Additionally, while the number of input/output pairs that can be used to calculate the transform $\mathbf{T}$ is unbounded for the rapid variable creation task, in a (well-formed) fluid induction task, only 5 transform calculations are needed. This fact, however, does not alter the algorithm Spaun uses to perform the fluid induction task.

System Requirements: The requirements for this task are identical to that of the rapid variable creation task.

3.3 Spaun Architecture

Using the system requirements detailed in the previous section, this section describes how these requirements have been organized to form Spaun's general architecture. Spaun's architecture is divided in 8 distinct modules, connected to each other as shown in Figure 3.20. In brief, these modules are:

1. The vision module – contains all of the networks related to Spaun's visual processing needs.
2. The motor module – contains all of the networks necessary for Spaun to produce a handwritten output (using its simulated arm).
3. The encoding module – the functional group of networks necessary to encode incoming visual information into Spaun's internal semantic pointer list representation.
4. The decoding module – the functional group of neural networks necessary to convert Spaun's internal list representation into the sequence of "digit" semantic pointers the motor module uses to produce Spaun's handwritten outputs.
5. The working memory module – contains the memory networks required for Spaun to complete the various tasks (all of Spaun's tasks with the exception of the copy-drawing and learning tasks).
6. The transformation module – contains the various networks to compute the various SPA operations needed to complete the question answering and induction tasks.
7. The reward evaluation module – the collection of networks required to compute the reward calculation detailed in Table 3.3.

8. The action selection module – the collection of networks responsible for keeping track of Spaun's current state, deciding on Spaun's actions with respect to its current state, and managing the flow of information throughout the Spaun model.

It should be noted that Spaun's architecture illustrated in Figure 3.20 differs slightly from the architecture used in the original Spaun model (as described in [Eliasmith et al., 2012], Figure 1B). Notably, the "motor processing" and "motor" modules of the original architecture have been combined into a singular "motor" module. Additionally, the flow control projections project directly onto each module, instead of projecting on flow control nodes as in the original architectural diagram. These changes more closely reflect the actual implementation of the Spaun model.

In addition to the changes made to Spaun's overall architectural diagram, the modules described in the following sections also differ slightly from the original Spaun implementation. The original Spaun model was constructed in a piecemeal fashion, with incremental additions made to the Spaun model to enable it to perform additional tasks. As part of this development process, the Spaun model was organized into the different functional modules (vision, motor, etc.) only after the model was mostly completed, resulting in obscure interdependencies existing between several of Spaun's modules. The Spaun architecture described in the following sections, however, is based on work done to re-analyze and formalize the Spaun model as part of the effort to re-implement and improve the Spaun model as Spaun 2.0 (see Chapter 4).

It is also important to note that architectural descriptions detailed in the following sections present a generalized Spaun architecture, with no constraints applied on the implementation of individual components. As an example, the hierarchical vision network does not have to be implemented as the network described in Section 3.1.1. Both the original Spaun model, and the Spaun 2.0 model can be considered specific instantiations of this generalized architecture.

3.3.1 Vision Module

Spaun's vision module contains all of Spaun's vision-related processing networks. This includes a visual hierarchy and a visual semantic pointer classifier, necessary for every task; and a visual working memory network, required for the copy-drawing task. In addition, as Spaun is almost entirely vision stimulus driven, a network is also required to identify changes in the visual stimuli. The output of this "stimulus change detector" network is then used to drive the action selection system, enabling Spaun to progress through each task as the visual stimuli are presented. Figure 3.21 illustrates how each of these components are connected to form Spaun's vision module.

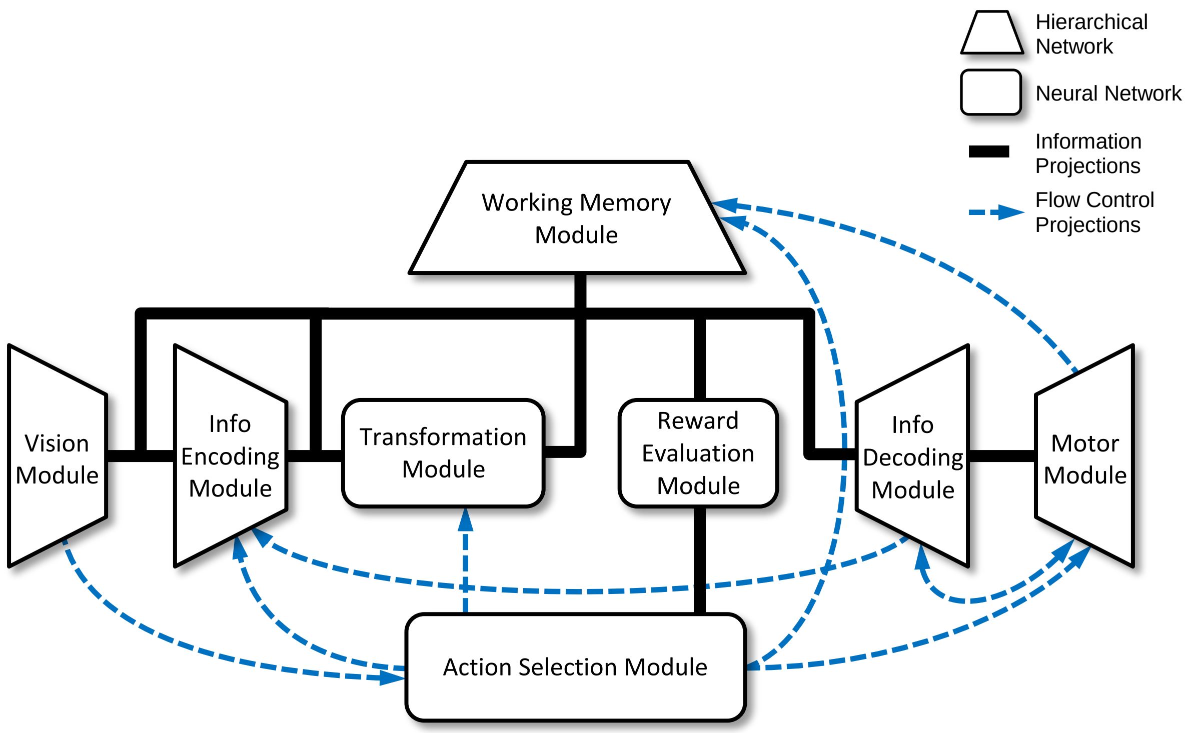


Figure 3.20: High-level architectural diagram of the Spaun model.

3.3.2 Motor Module

Spaun's motor module contains all of the networks required for Spaun to convert a motor semantic pointer into the equivalent hand-written digit output. The motor module consists of a motor hierarchy, and a network that generates the appropriate motor timing control signals necessary to coordinate the operation of the motor hierarchy with the operation of the simulated arm. Figure 3.22 illustrates how these components are connected in Spaun's motor module.

3.3.3 Information Encoding Module

Spaun's information encoding module contains the networks necessary to encode the sequential digit semantic pointers into Spaun's list semantic pointer representation. This module consists of a network to generate the position semantic pointer tags, and a network to perform the SPA operation(s) needed to generate list semantic pointer representation. Figure 3.23 illustrates the design of the information encoding module.

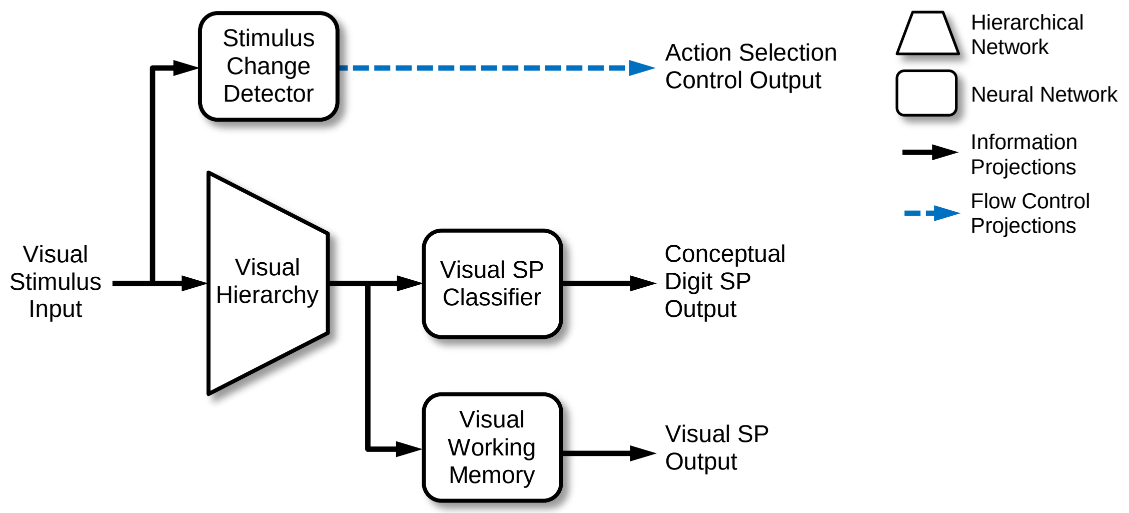


Figure 3.21: Architectural schematic of Spaun's vision module. See accompanying text for a detailed description of each component.

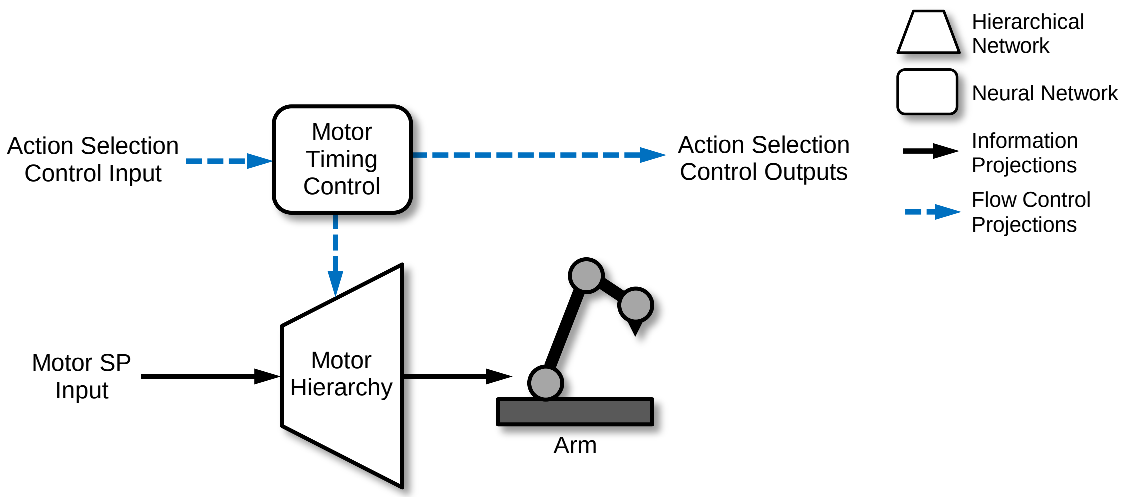


Figure 3.22: Architectural schematic of Spaun's motor module. See accompanying text for a detailed description of each component.

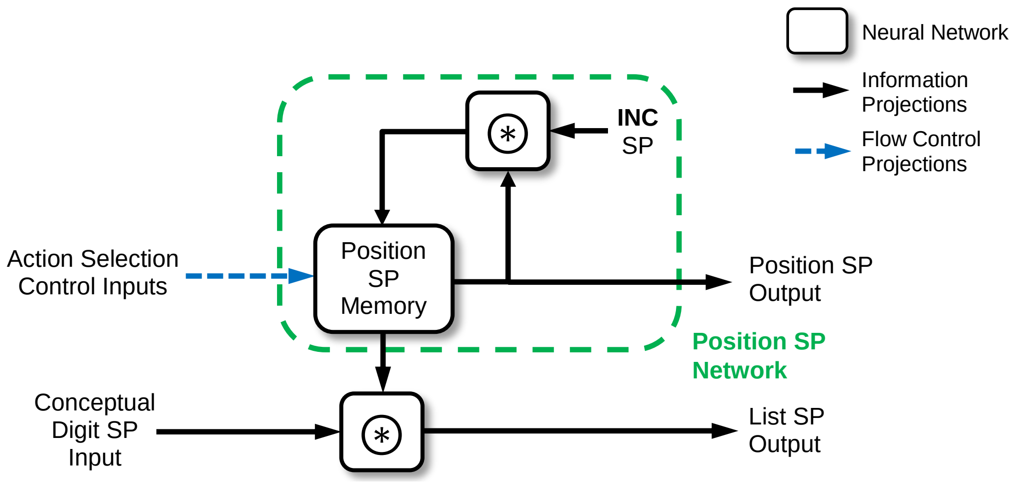


Figure 3.23: Architectural schematic of Spaun's information encoding module. See accompanying text for a detailed description of each component.

3.3.4 Information Decoding Module

Spaun's information decoding module is responsible for converting Spaun's various internal semantic pointer representations into the semantic pointer representation used by the motor module. This module includes the networks required to recall information from Spaun's list semantic pointer representation, and the networks required to perform the transformations between the visual and motor semantic pointer representations necessary for Spaun's copy-drawing task (see Section 3.2.2).

In addition to these task specific components, Spaun's information decoding module contains several flow control networks, required to generate the appropriate flow control signals based off the state of the recall networks ("decode control"), and to route the appropriate semantic pointer information to the module's output based on the task being performed ("output selector"). Figure 3.23 illustrates how these components have been connected in the information decoding module.

3.3.5 Working Memory Module

Spaun's working memory module contains the memory networks required to complete Spaun's variety of tasks. These memory networks include the three memory networks required for the counting task, which, using control signals generated by the action selection module, can be reused for Spaun's other tasks as well. In addition to the "standard" list memory networks, the

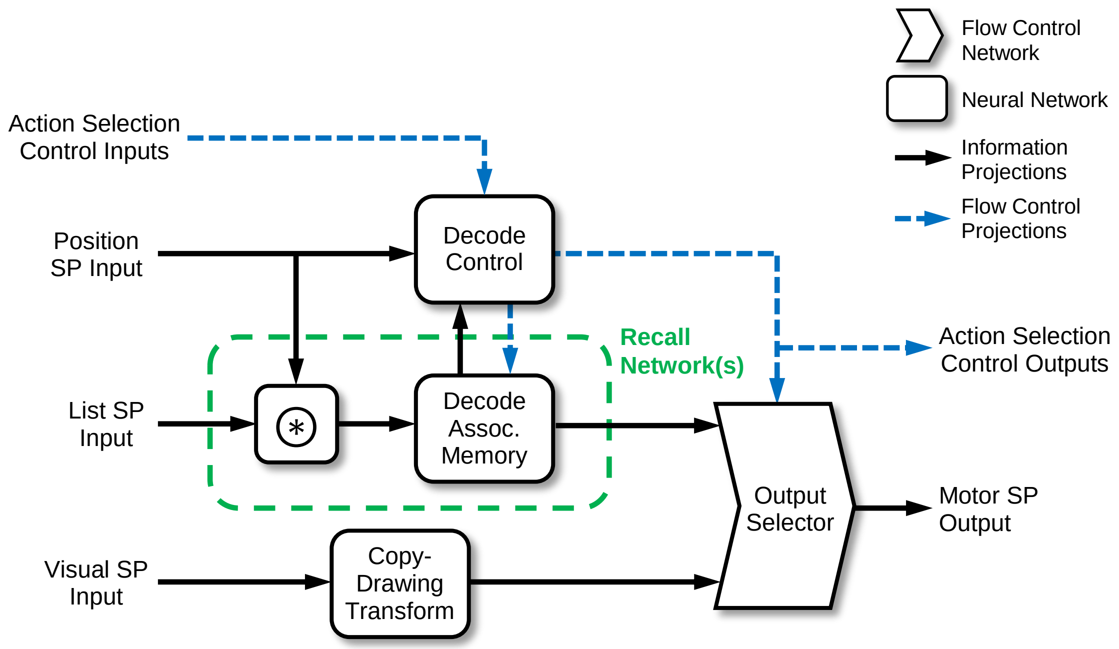


Figure 3.24: Architectural schematic of Spaun's information decoding module. See accompanying text for a detailed description of each component.

memory module also contains a specialized averaging network for Spaun's induction tasks (see Section 3.2.8, specifically Eq. (3.17)). Figure 3.25 illustrates the layout of the memory module.

3.3.6 Transformation Module

Spaun's transformation module houses the networks necessary for computing the transformation calculations necessary for the question answering (see Section 3.2.7) and induction tasks (see Sections 3.2.8, 3.2.9). Careful analysis of the individual task requirements reveals that, with the use of information flow control networks ("input selector"), only one SPA binding network is necessary to compute all of the transforms required.

In addition to the SPA binding network, the transformation module includes the associative memory networks required for the question answering task (see Figure 3.17), a dot product network to perform the semantic pointer comparison needed to generate the "stop" condition for the counting task (see Table 3.4), and an information flow network ("output selector") to ensure that depending on the task being performed, the outputs of the appropriate networks are routed to the output of the transformation module.

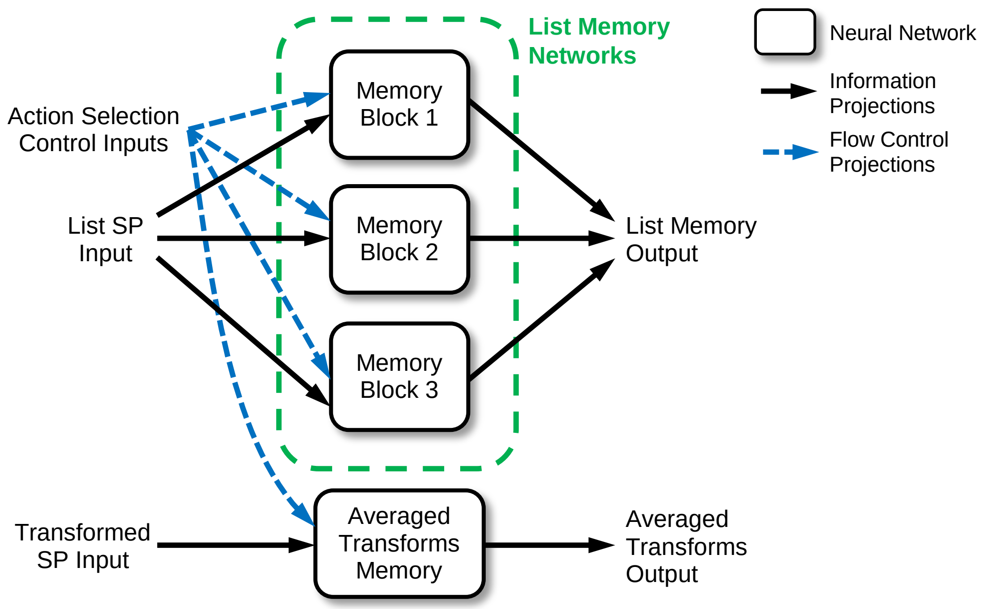


Figure 3.25: Architectural schematic of Spaun's memory module. See accompanying text for a detailed description of each component.

Figure 3.26 illustrates how each module component has been connected to achieve the functionality required of the transformation module.

3.3.7 Reward Evaluation Module

Spaun's reward evaluation module computes the appropriate learning error values detailed in Table 3.3. As such, the reward evaluation module consists solely of the network used to perform this calculation, as illustrated in Figure 3.27.

3.3.8 Action Selection Module

Spaun's action selection module is responsible for keeping track of Spaun's progress and deciding what actions to perform as it executes each task. Additionally, the action selection system is responsible for managing the flow of information throughout the different networks that make up Spaun.

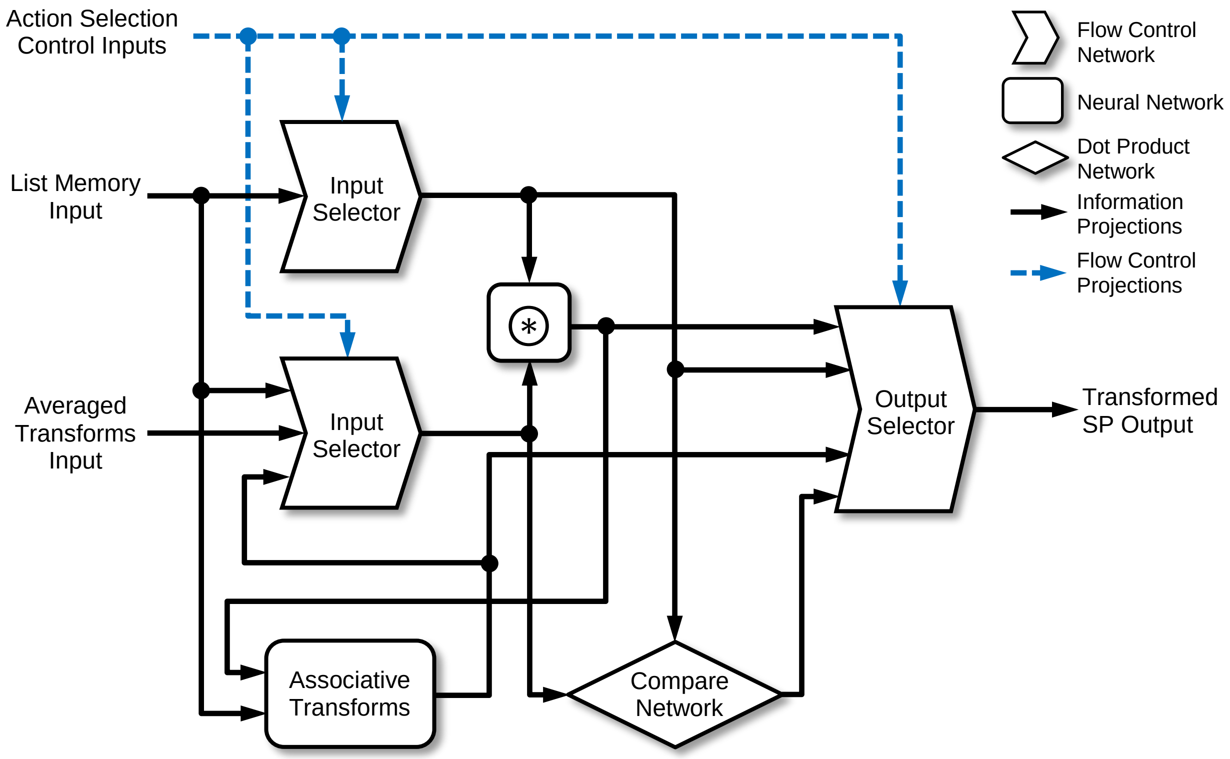


Figure 3.26: Architectural schematic of Spaun's transformation module. See accompanying text for a detailed description of each component.

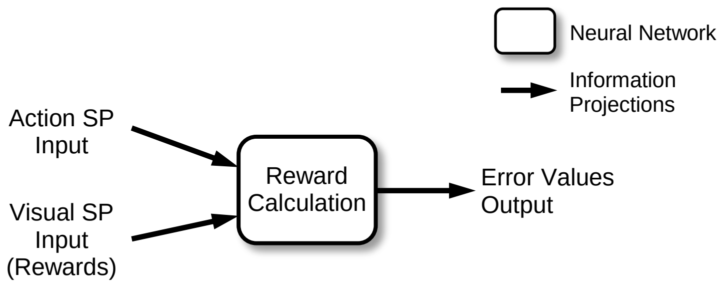


Figure 3.27: Architectural schematic of Spaun's reward evaluation module. See accompanying text for a detailed description of each component.

Naïvely, the reasoning system described in Section 3.1.4 can be used to implement all of the functionality required of Spaun's action selection module. However, as the following example will illustrate, implementing Spaun's action selection module using just the reasoning system results in increasingly complex condition-consequence pairs, and consequently, and overly convoluted implementation.

In the following example, the rapid variable creation (RVC) task (refer to Section 3.2.8 will be used to demonstrate how the condition-consequence pairs of the reasoning system can be used to implement the action selection module of Spaun. Processing the RVC task can be divided into four distinct parts: initialization of the task, storing the input list (IN_N) into memory, storing the output list (OUT_N) and averaged transform into memory, and finally, outputting the result to the motor system. As the action selection module is responsible for tracking Spaun's task progress, using the reasoning system in Spaun requires the addition of a working memory network (*task*) to store the semantic pointers that will be used to represent each of the four stages of the RVC task (**TASK_ASSIGN**, **STORE_IN**, **STORE_OUT**, **RVC_OUTPUT**).

The initialization stage of the RVC task handles the "**A6**" input character sequence that denotes the RVC task. When Spaun is shown the "**A**" character, it has to shift into the **INIT** state. If (and only if) Spaun is in the **INIT** state, and Spaun is presented with the digit "**6**" does it move then into the **STORE_IN** task stage. The condition-consequence pairs necessary for this functionality are:

$$\begin{aligned} \mathit{vision} \bullet \mathbf{A} &\longmapsto \textsf{\textbf{INIT}} \Rightarrow \mathit{task} \\ 0.5 \times (\mathit{task} \bullet \textsf{\textbf{INIT}} + \mathit{vision} \bullet \mathbf{6}) &\longmapsto \textsf{\textbf{STORE\_IN}} \Rightarrow \mathit{task} \end{aligned}$$

The next stage of the RVC task requires Spaun to store the incoming digit sequence into a working memory location (*mem*1). As part of this stage, when Spaun is shown the "▶" character (indicating the start of a list), the action selection module has to clear any semantic pointers stored in *mem*1, and reset the position working memory (*pos_mem*) representation back to **POS1** in preparation to correctly encode and store in the incoming digit list. As digits are presented, the action selection module projects the output of the encoding system (*enc*) into working memory. This stage of the RVC task is concluded when Spaun is presented the "◀" character, at which point Spaun moves into the **STORE_OUT** step of the RVC task. The condition-consequence pairs for this stage of the task are:

$$\begin{aligned} 0.5 \times (\mathit{task} \bullet \textsf{\textbf{STORE\_IN}} + \mathit{vision} \bullet \blacktriangleright) &\longmapsto \emptyset \Rightarrow \mathit{mem}1, \textsf{\textbf{POS1}} \Rightarrow \mathit{pos\_mem} \\ \mathit{task} \bullet \textsf{\textbf{STORE\_IN}} - \mathit{vision} \bullet (\blacktriangleright + \blacktriangleleft) &\longmapsto \mathit{mem}1 + \mathit{enc} \Rightarrow \mathit{mem}1, \\ &\qquad \mathit{pos\_mem} \circledast \textsf{\textbf{INC}} \Rightarrow \mathit{pos\_mem} \\ 0.5 \times (\mathit{task} \bullet \textsf{\textbf{STORE\_IN}} + \mathit{vision} \bullet \blacktriangleleft) &\longmapsto \textsf{\textbf{STORE\_OUT}} \Rightarrow \mathit{task} \end{aligned}$$

The third step of the RVC task is essentially identical to the second step of the RVC task, with two notable exceptions. First, instead of storing the incoming digit list into $mem1$, the digit list has to be stored in another memory location ($mem2$) to avoid overwriting the list information encoded in the previous stage of the RVC task. Second, when Spaun is presented with the "◀" character, the action selection module has to transition back to the **STORE_IN** state, in preparation for the next set of IN and OUT list pairs. Additionally, the inductive transform **T** needs to be calculated and stored in the "averaging" memory network (mem_ave). The condition-consequence pairs for this stage of the task are:

$$\begin{aligned}
0.5 \times (task \bullet \mathbf{STORE\_OUT} + vision \bullet \blacktriangleright) &\longmapsto \emptyset \Rightarrow mem2, \mathbf{POS1} \Rightarrow pos\_mem \\
task \bullet \mathbf{STORE\_OUT} - vision \bullet (\blacktriangleright + \blacktriangleleft) &\longmapsto mem2 + enc \Rightarrow mem2, \\
&\qquad pos\_mem \circledast \mathbf{INC} \Rightarrow pos\_mem \\
0.5 \times (task \bullet \mathbf{STORE\_OUT} + vision \bullet \blacktriangleleft) &\longmapsto mem2 \circledast \sim mem1 \Rightarrow mem\_ave, \\
&\qquad \mathbf{STORE\_IN} \Rightarrow task
\end{aligned} \tag{3.18}$$

The final step of the RVC task is initiated when Spaun is presented with the "**?**" character. When presented with the question mark, the action selection module shifts Spaun into the **RECALL** task stage, and resets the semantic pointer in position working memory back to **POS1**. In the recall phase, the output of the decoding module (dec) is projected to the motor module (mtr) every time the motor module state (mtr_state) indicates it has completed writing a digit. The condition-consequence pairs for final stage of the RVC task are:

$$\begin{aligned}
0.5 \times (task \bullet (\mathbf{STORE\_IN} + \mathbf{STORE\_OUT}) + vision \bullet \mathbf{?}) &\longmapsto \mathbf{RECALL} \Rightarrow task, \\
&\qquad \mathbf{POS1} \Rightarrow pos\_mem \\
0.5 \times (task \bullet \mathbf{RECALL} + mtr\_state \bullet \mathbf{DONE}) &\longmapsto dec \Rightarrow mtr, \\
&\qquad pos\_mem \circledast \mathbf{INC} \Rightarrow pos\_mem
\end{aligned}$$

As the example illustrates, a substantial number of condition-consequence pairs are required if the reasoning system is solely used to implement Spaun's action selection module. A contributing factor to the number of condition-consequence pairs is that while the second and third stages (**STORE_IN** and **STORE_OUT**, respectively) of the RVC task are fairly similar, the condition-consequence pairs cannot be combined because the target working memory locations are different. This issue is further exacerbated in the fluid intelligence task, with it requiring three sets of similarly structured condition-consequence pairs.

In order to reduce the number and complexity of the condition-consequence pairs, Spaun's action selection module is implemented as a three-layer hierarchy, with each layer of the hierarchy responsible for the different functions the action selection system is responsible for. The top level of the hierarchy receives input from the vision module and is only responsible for high-level tracking, planning, and execution of Spaun's task progress. In Spaun, this layer is implemented using the reasoning system described in Section 3.1.4. The second layer of the hierarchy combines the state representation of the top layer with the visual semantic pointers to generate the module-specific control signals used to control the flow of information within each of Spaun's modules. The last layer of the action selection hierarchy are the physical networks (and network architectures) necessary to accomplish the information flows dictated by the outputs of the second layer of the action selection hierarchy. Figure 3.28 illustrates the structure and projections to and from the action selection hierarchy.

3.3.8.1 Layer 1: Task Progress Management

As described previously, Spaun uses the adaptive reasoning system to implement the first layer of the action selection hierarchy. However, instead of using the naïve reasoning system implementation described in the previously described RVC example, the condition-consequence pairs used in Spaun's action selection hierarchy have been derived by analyzing and identifying commonalities in the stages necessary to process each of Spaun's tasks. It should be noted that the action selection hierarchy does not impose any restrictions on the condition-consequence pairs used in the adaptive reasoning system. Rather, the condition-consequence pairs used should be tailored to the tasks assigned to the cognitive system. As a matter of fact, because of the conclusions of the task analysis yielded different grouping of task commonalities, the condition-consequence pairs used in the original Spaun model and the Spaun 2.0 model differ slightly. The reasoning system description provided below is based on Spaun 2.0's reasoning system.

Spaun's reasoning system is implemented using three cortical state semantic pointers. The semantic pointers represent the various tasks Spaun is capable of performing, the different task processing stages (i.e., processing of task information presented before the "**?**" character), and the different task decode stages (i.e., the steps necessary to produce the correct output for a specified task).

Task Representation (*task*): Spaun's task representation is used to keep track of the task that Spaun has been instructed to perform. For its task representation, Spaun uses nine semantic pointer values, one to represent each of Spaun's eight tasks, and an additional semantic pointer to represent the "initialization" task step as described in the RVC example from the previous section. Table 3.5 lists the task semantic pointer mapping used in future sections of this document.[29]

Task Processing Stage Representation (*task_stage*): Spaun's task processing stage representation is used to represent the task stage Spaun is in the process of performing. The task

[29]It should be noted that while the semantic pointers are conceptually consistent, the names used in Table 3.5 differ from that in Spaun's code base, primarily to facilitate ease of conceptual understanding in this document.

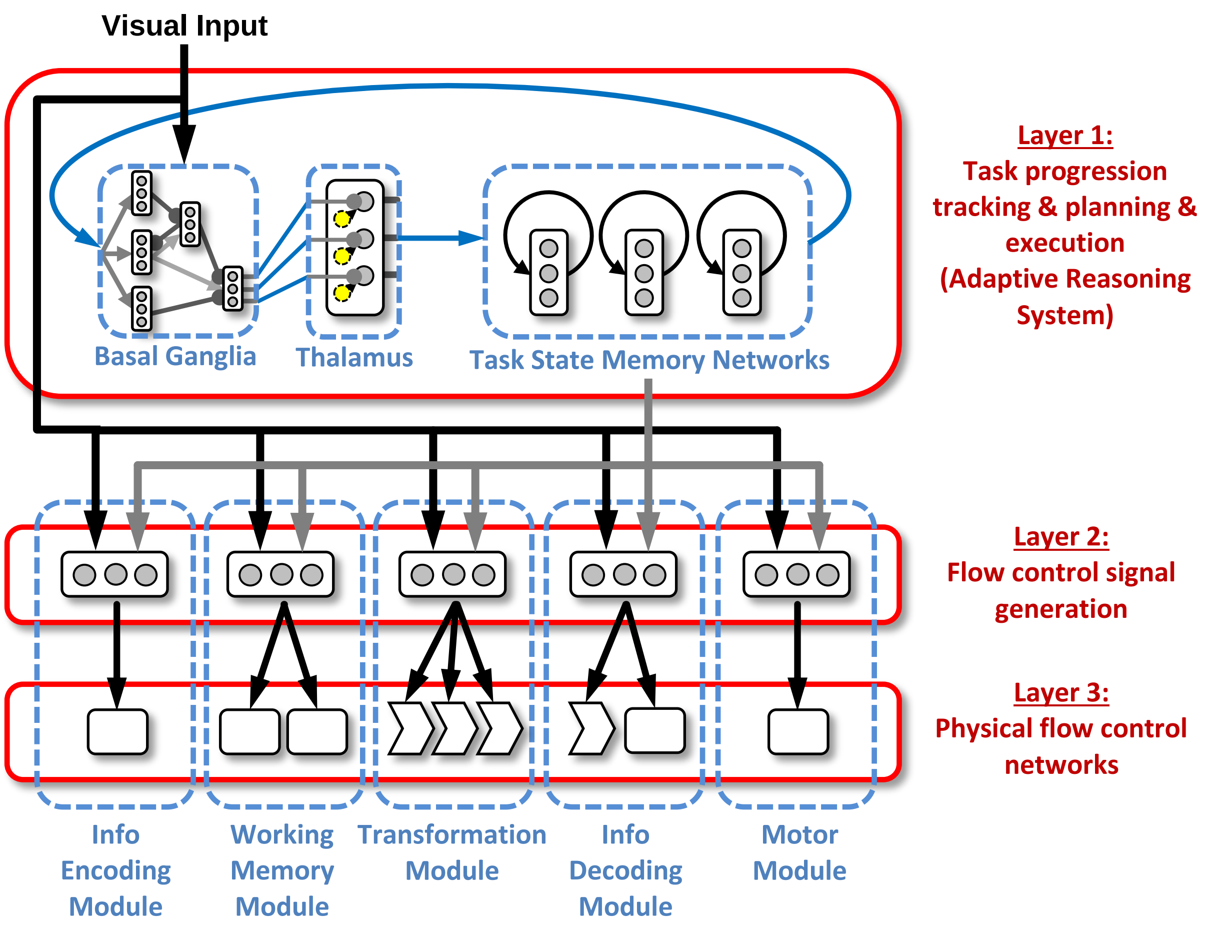

Visual Input
Basal Ganglia
Thalamus
Task State Memory Networks
Layer 1:
Task progression tracking & planning & execution
(Adaptive Reasoning System)
Layer 2:
Flow control signal generation
Layer 3:
Physical flow control networks
Info Encoding Module
Working Memory Module
Transformation Module
Info Decoding Module
Motor Module

Figure 3.28: Schematic diagram of the action selection hierarchy. The action selection hierarchy consists of three layers. The first layer is responsible for tracking, planning and executing Spaun's progression through its tasks. This layer contains the adaptive reasoning system (a combination of the networks described in Sections 3.1.4 and 3.1.6) that performs Spaun's core decision making process. The second layer consists of multiple networks that combine the output of the task state memory networks (in Layer 1) with visual information to generate the appropriate module-specific flow control signals. The final layer contains the physical flow control network implementations (i.e., the networks that receive "action selection control input" projections in Figures 3.22 through 3.26) that realizes the intended effect of the flow control signals generated in the second layer of the action selection hierarchy. It is important to note that unlike traditional hierarchies where each layer is typically contained within a network, the actions implemented by the various components in layer 2 and 3 of Spaun's action selection hierarchy are module specific, and are thus distributed amongst the different Spaun modules. In the figure, this is denoted by the vertical grouping of the individual components in layer 2 and 3 of the hierarchy.

Semantic Pointer	Conceptual Meaning
INIT	Task initialization phase.
COPY_DRAW	Copy-drawing task.
RECOG	Digit recognition task.
LEARN	Bandit task (learning).
WM	Working memory task.
COUNT	Counting task.
QA	Question answering task.
RVC	Rapid variable creation task.
FLUID_IND	Fluid induction (Raven's progressive matrix) task.
DECODE	Task output generation (decode) phase.

Table 3.5: Table of Spaun task representation semantic pointers, and their respective conceptual meanings.

stage representations are denoted to maximize its reuse amongst the variety of Spaun's tasks. As an example, the first step in each task, with the exception of the copy-drawing and learning tasks, is to process a digit sequence and store it in a memory location. In Spaun, this common first step is assigned the representation "**STORE**". Another example of a common task step is the second stage of both induction (RVC, and fluid induction) tasks. In this step, referred to as "**TRANSFORM1**", Spaun is to process a digit sequence, store it in a memory location distinct from the memory location used in the **STORE** stage, and compute the semantic pointer transform between the first (**STORE**) and second (**TRANSFORM1**) memory locations. Table 3.6 lists the various task processing stage semantic pointer representations found in Spaun.

Task Decoding State Representation (*task_decode*): Spaun's task decoding state representation is used to represent the different types of operations necessary to generate Spaun's output from the Spaun's various internal semantic pointer representations. As an example, the "**FORWARD**" decoding type is used for the digit recognition, memory, and question answering tasks, and converts the output of the transformation module into a sequence of written digits. Table 3.7 lists the various task decoding state semantic pointer representations found in Spaun.

RVC Task Implementation Example: With the three different semantic pointer representations, the condition-consequence pairs can be constructed to process each of Spaun's tasks. Once again, the RVC task is used as an example.

The initialization stage of the RVC task remains unchanged from the previous condition-

Semantic Pointer	Conceptual Meaning
STORE	Common task stage to process the digit list for the digit recognition and memory tasks, the first digit list for the counting and question answering tasks, the input digit lists for the RVC task (IN_N), and the first column of cells for the fluid induction task (C_1, C_4, C_7).
TRANSFORM1	Common task stage to process the output digit list for the RVC (OUT_N) and the second column of cells for the fluid induction task (C_2, C_5, C_8).
TRANSFORM2	Task stage to process the third column of cells for the fluid induction task (C_3, C_7).
QAP	Task stage to process the "position" query of the question answering task.
QAK	Task stage to process the "kind" query of the question answering task.
COUNT1	Task stage to process the storing of the second list in the counting task.
COUNT2	Task stage to process the internal incrementing of semantic pointers stored in memory during the silent counting process of the counting task.
LEARN	Task stage to handle the input-to-reward mapping for the bandit task.

Table 3.6: Table of Spaun task processing stage semantic pointers, and their respective conceptual meanings.

consequence implementation, with the exception that the **STORE_IN** state replaced with the **STORE** state, and that Spaun now keeps track of the task it is performing:

$$vision \bullet \mathbf{A} \longmapsto \textsf{\textbf{INIT}} \Rightarrow task$$
$$0.5 \times (task \bullet \textsf{\textbf{INIT}} + vision \bullet \mathbf{6}) \longmapsto \textsf{\textbf{RVC}} \Rightarrow task, \textsf{\textbf{STORE}} \Rightarrow task\_stage$$

After the **STORE** state completes, the next stage of the RVC task is to store the "output" list, and compute the induction transform **T**. This is denoted by shifting Spaun into the **TRANSFORM1**

Semantic Pointer	Conceptual Meaning
FORWARD	Decoding type used to convert the output of the transformation module into a list of written digits, recalled in the forwards manner.
COUNT	Decoding type used to maintain motor system perpetuation during the silent counting process.
DRAW	Decoding type used to convert stored visual semantic pointers into motor semantic pointers through the use of the copy-drawing transform $\mathbf{T}$.
INDUCTION	Decoding type used to generate the outputs of the induction task, which require binding the semantic pointers stored in memory with the calculated averaged transforms.

Table 3.7: Table of Spaun task decoding state semantic pointers, and their respective conceptual meanings.

task state like so:

$$0.5 \times (task \bullet \textsf{\textbf{RVC}} + task\_stage \bullet \textsf{\textbf{STORE}}) - (vision \bullet \textbf{?}) \longmapsto \textsf{\textbf{TRANSFORM1}} \Rightarrow task\_stage$$

Once the processing of the "output" list is completed, Spaun loops back to the processing of the "input" list:

$$0.5 \times (task \bullet \textsf{\textbf{RVC}} + task\_stage \bullet \textsf{\textbf{TRANSFORM1}}) - (vision \bullet \textbf{?}) \longmapsto \textsf{\textbf{STORE}} \Rightarrow task\_stage$$

As before, the final step of the RVC task is initiated when Spaun is presented with the "**?**" character, shifting Spaun into the "**INDUCTION**" decoding state, and appending "**DECODE**" to the task representation. It should be noted that the task stage representation is not changed as it determines which semantic pointers to use when calculating the output of the averaged transform binding operation. The condition-consequence pair for this step is:

$$vision \bullet \textbf{?} \longmapsto task + \textsf{\textbf{DECODE}} \Rightarrow task,\ task\_stage \Rightarrow task\_stage,\ \textsf{\textbf{INDUCTION}} \Rightarrow task\_decode$$

As the RVC task example above demonstrates, by restructuring the reasoning system to maximize the common processing steps identified in each task and by delegating the processing of these steps to the second and third layer of the action selection hierarchy, the number and complexity of the condition-consequence pairs can be reduced (in the case of the RVC task, the number of condition-consequence pairs has been halved). As a result of this optimization, the behaviour of the reasoning system is less prone to error (i.e., activation of the wrong condition-consequence pair) and more efficient (i.e., utilizes fewer neurons). In addition, as demonstrated in Section 5.2.2, tasks with similar structure can be added with relative ease.

3.3.8.2 Layer 2: Flow Control Signal Generation

The second layer of the action selection hierarchy generates module-specific control signals for the flow control networks used in each module. The control signals are generated using a combination of information from the visual system, and from the task information represented in the first layer of the action selection hierarchy.

Regardless of the source of information used to formulate the flow control signals, the signals themselves are usually generated using the same mechanism to compute the similarity values used in the associative memory networks (see Section 2.5.5.5) and the basal ganglia utility values used in the reasoning system (see Section 3.1.4) – that is, by using a transformation matrix combined with a neural ensemble.

The control signals computed by using the transformation matrices are typically single-dimensional "binary" values (i.e., only the "low" or "high" values convey meaning). The exact effect the control signals have on the third layer of the action selection hierarchy depends on the particulars of the flow control network (refer to the next section) the control signals project to. Examples of the interaction between the second and third layer of the action selection hierarchy are presented in Section 3.3.8.4.

3.3.8.3 Layer 3: Physical Flow Control Networks

The last layer of the action selection hierarchy consists of the physical neural networks that physically control the flow of information within and between the different modules of Spaun. The function of these flow control networks are similar to the gating component introduced in Section 2.5.6.4. However, while the gating component only has one input and one output, the flow control networks described here typically have multiple inputs or multiple outputs.

Two categories of flow control networks are used in Spaun: networks that are used to route the flow of information to different Spaun components, and networks with memory functionality that control the timing of when information is presented (or stored) in the different modules of Spaun.

Routing Networks: The routing flow control networks in Spaun are implemented by combining multiple gates together in different configurations. This section describes two flow control elements: the selector and the router.

The selector has multiple inputs and one output, and functions like a switch that connects a desired input to the output based on the control signals it is provided.[30] To implement the selector in a neural network, each of the inputs are connected to a gate, and the outputs of the gate are connected to the output of the selector network. To control the selector, a "select" control signal is needed for each of the inputs, and it is connected in such a way that when one input is selected, all other inputs are gated. Figure 3.29 illustrates the layout of a 3-input selector, along with the spiking activity graph and the decoded activity graph for the selector network.

The router is similar in function to the selector, except in reverse – it has one input that it connects to one-of-many outputs.[31] To implement the router in a neural network, one gate is used per output, and the input to the router is connected to the inputs of all of the gates. As with the selector, "route" control signals are added to the network such that when a particular output is selected, only that output gets connected to the input while all of the other outputs are inhibited. Figure 3.30 illustrates the layout of a 3-input router, along with the spiking activity graph and the decoded activity graph for the selector network.

Both the selector and router networks are used extensively in Spaun as they are necessary in order to allow commonly used components (e.g., the working memory system) to be generically available instead of task specific.

Memory Networks: The flow control memory networks control the timing of when information is stored and projected to the different modules in Spaun. The task state memory example in Section 3.3.8.4 demonstrates this particular functionality of the flow control memory networks. In addition to this functionality, the flow control memory networks are also used to implement another crucial requirement in Spaun: the simultaneous "reading" and "writing" of a semantic pointer.

The requirement to "read" and "write" to the same semantic pointer occur whenever an operation of the form arises:

$$\mathbf{A} = \mathbf{A} * \mathbf{X}, \tag{3.19}$$

where $\mathbf{A}$ and $\mathbf{X}$ are semantic pointers, and the symbol ($*$) represents any SPA operator. Examples of constructs in this form are the generation the position semantic pointers (1), the construction of list semantic pointers (2), and even the condition-consequence pairs used in the first layer of the action selection hierarchy (3):

[30] In electronic circuit design, this is known as a multiplexer.
[31] In electronic circuit design, this is known as a demultiplexer.

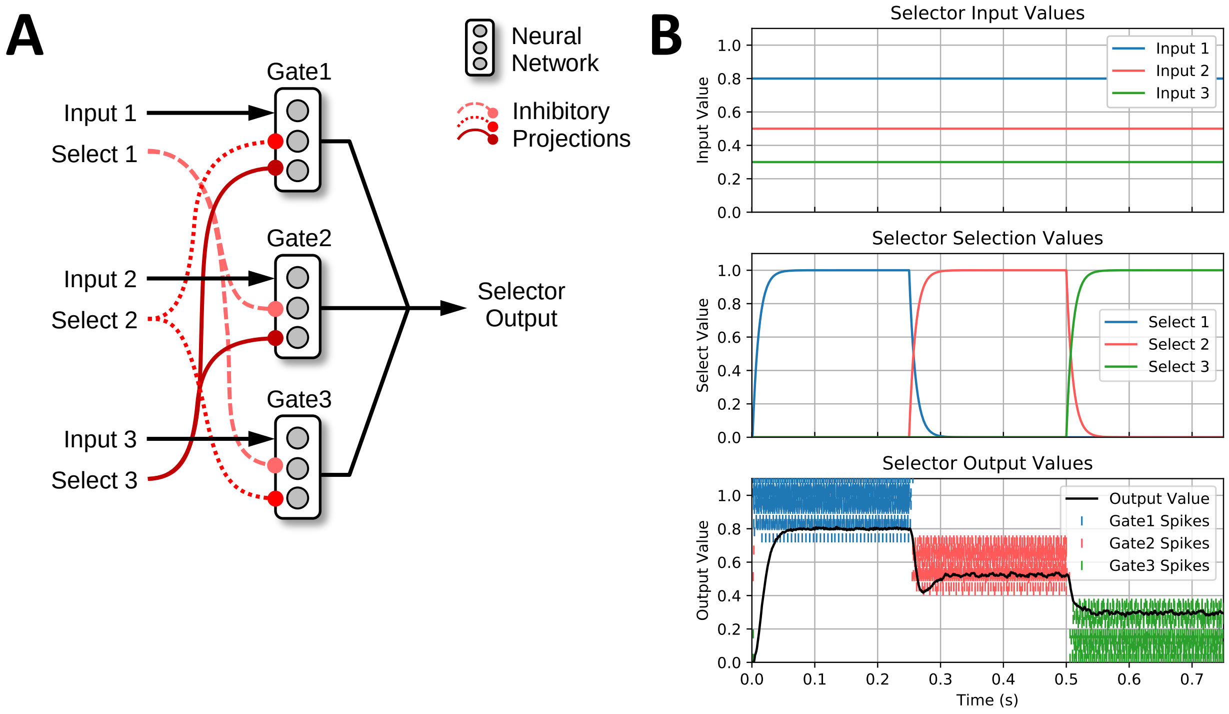
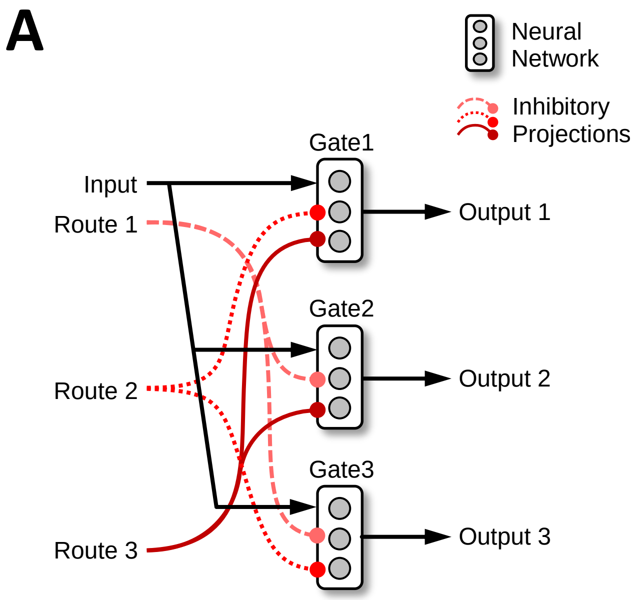


Figure 3.29: Schematic and recorded inputs and outputs of a three-channel spiking neural selector. (**A**) Schematic of a generic three-channel selector. The selector has three inputs and one output. The output of the selector is determined by the "select" control signals provided (through the use of the inhibitory connections). (**B**) Recorded inputs and outputs of a three-channel single-dimensional selector implemented in a spiking neural network. (Top) The selector is provided with three input values: 0.8, 0.5 and 0.3 for Inputs 1, 2 and 3 respectively. (Middle) The provided selector control signals. The selector is instructed to select Input 1 from $t = 0$ to $0.25s$, Input 2 from $t = 0.25$ to $0.2s$, and Input 3 from $t = 0.5$ to $0.75s$. (Bottom) Decoded output and spike plots for the selector network. As illustrated, the selector selects the appropriate values when given the select control signals. Additionally, the spike graphs indicate that for any given "select" control signal, only one of the neural populations is active.

1. $\textsf{\textbf{POS}}N = \textsf{\textbf{POS}}(N-1) \circledast \textsf{\textbf{INC}}$
2. $\textsf{\textbf{MEMORY}} = \textsf{\textbf{MEMORY}} + \textsf{\textbf{POS}}N \circledast \textsf{\textbf{DIGIT}}N$
3. $0.5 \times (task \bullet \textsf{\textbf{INIT}} + \ldots) \longmapsto \textsf{\textbf{TASK}} \Rightarrow task, \ldots$

Because the neural networks found in Spaun represent continuous values, the naïve approaches to implementing these functions, typically by using a feedback projection can result in "runaway" computations (e.g., values that keep accumulating) if nothing is done to regulate the flow of



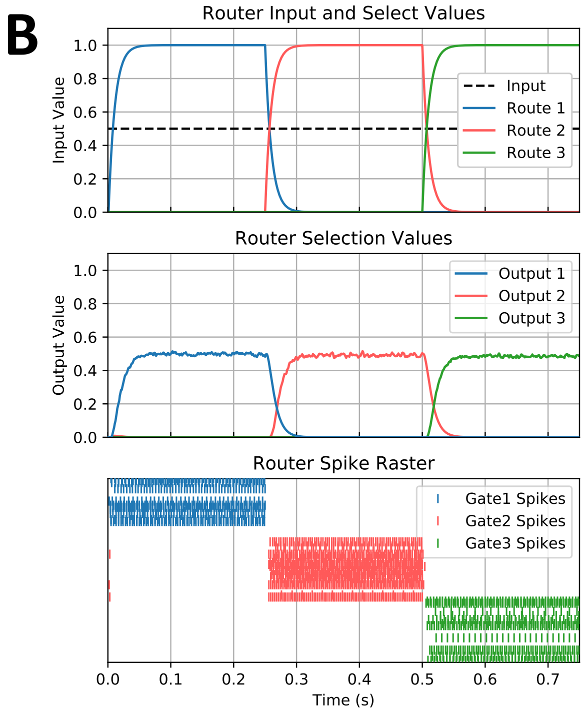


Figure 3.30: Schematic and recorded inputs and outputs of a three-channel spiking neural router. (**A**) Schematic of a generic three-channel router. The router has one input and three outputs. The value of each output of the router is determined by the "route" control signals provided. At most one output is connected to the input at any one time. (**B**) Recorded inputs and outputs of a three-channel single-dimensional router implemented in a spiking neural network. (Top) The router is provided with an input value of 0.5. Additionally, the router is instructed to route to Output 1 from $t = 0$ to $0.25s$, Output 2 from $t = 0.25$ to $0.2s$, and Output 3 from $t = 0.5$ to $0.75s$. (Middle) Decoded output plots for the router network. As illustrated, the outputs of the router take on the right value (0.5) depending on the router control signals, while all of the other (non-selected) outputs remain at 0. (Bottom) Spike plots for the router network showing that for any given "route" control signal, only one of the neural populations is active.

information through the network. Spaun's action selection hierarchy addresses these issues by using a memory-enabled flow control network referred to as a "memory block".

The core concept behind the memory blocks is the use of two memory networks (e.g., integrators), one to perform the "read" operation, and one to perform the "write" operation. The flow of information to these memory networks is controlled through the use of the gating networks described in Section 2.5.6.4.

In Spaun, the memory block networks are constructed using two memory sub-networks connected in a serial fashion, with gating ensembles located to control the flow of information into these memory sub-networks (see Figure 3.31). For a given flow control signal value, gating signals are generated such that information is permitted to flow into one of the memory sub-networks, and prohibited from flowing into the other. With this mechanism, information can be projected ("written") to the memory block without affecting the information being projected out ("read") from the memory block. Figure 3.31 illustrates the network structure of the memory block, and demonstrates an example operation of the memory block, showing the interaction between the value of the flow control signal and the information stored within each memory sub-network.

It is important to note that the "simultaneous-read-and-write" functionality can be achieved using just the integrator memory network discussed in Section 2.5.6.1 – through the use of intermediate "delay" neural ensembles, and by balancing the input weight into the integrator and the synaptic time constants of the various feedback projections. However, the Spaun model opts to use the somewhat engineered approach described above as it results in a memory network that performs robustly and consistently without the need to re-balance the memory network whenever slight changes are made to the Spaun model (as part of Spaun's development and testing process).

3.3.8.4 Action Selection Hierarchy Examples

As demonstrations of how the second layer of the action selection hierarchy performs its role, two examples will be used: the control signal generation for the task state memory networks found in the first layer of the action selection hierarchy, and the generation of the control signals specific to the transformation module during the second phase of the RVC task.

Task State Memory Control Signals Example: A closer look at the condition-consequence pairs derived for the RVC task reveals that they should oscillate, much like the example sequence repetition implementation of the reasoning system (see Section 3.1.4.4). The condition-consequence pairs in question are:

$$0.5 \times (task \bullet \textsf{\textbf{RVC}} + task\_stage \bullet \textsf{\textbf{STORE}}) \longmapsto \textsf{\textbf{TRANSFORM1}} \Rightarrow task\_stage \quad (3.20)$$

$$0.5 \times (task \bullet \textsf{\textbf{RVC}} + task\_stage \bullet \textsf{\textbf{TRANSFORM1}}) \longmapsto \textsf{\textbf{STORE}} \Rightarrow task\_stage, \quad (3.21)$$

which are of the sequence repetition form:

$$state \bullet \textsf{\textbf{A}} \longmapsto \textsf{\textbf{B}} \Rightarrow state$$
$$state \bullet \textsf{\textbf{B}} \longmapsto \textsf{\textbf{A}} \Rightarrow state$$

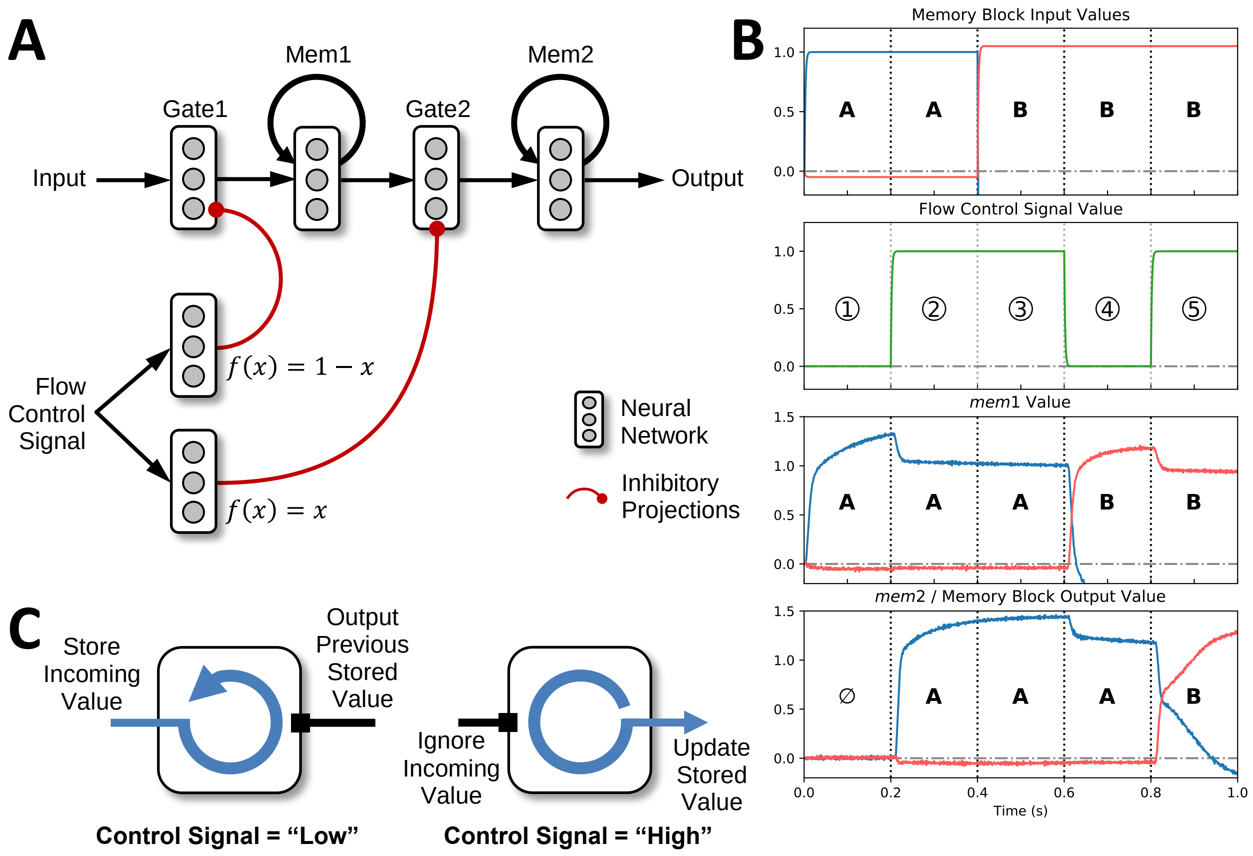


Figure 3.31: Schematic and recorded inputs and outputs of a memory block network. (**A**) Schematic of a generic memory block network. The memory block network has two inputs, one for information to be stored within the network, and one that serves as the flow control signal to the network. The flow control signal is used to generate the gating signals to each gate ("Gate1" and "Gate2"). The gating signals are configured such that information is allowed to pass only one gate at any moment in time. (**B**) Recorded inputs and outputs of a 256-dimensional memory network. [①] In this segment, the provided control signal has a "low" value, thus allowing the input value to be projected to "Mem1". [② & ③] In this segment, the provided control has a "high" value, thus inhibiting "Gate1" and preventing the input from modifying the value stored in "Mem1". Meanwhile, "Gate2" is dis-inhibited, allowing "Mem2" to be updated with the value stored in "Mem1". As segment ③ demonstrates, changing the input value to the memory block has no effect on the output values. [④] In this segment, the control value is once again "low", allowing the input value **B** to be projected into "Mem1". However, since "Gate2" is inhibited, the output of the network is unchanged. This thus demonstrates the "simultaneous-read-and-write" ability for the memory block network. [⑤] The flow control signal is "high" once again, allowing the output value of the network to be updated. (**C**) Schematic representations of the two states of the memory block network. (*Left*) This diagram represents the "control signal low" state of the memory block, whereby the input value is stored in "Mem1", while the output of the network remains unchanged. (*Right*) This diagram represents the "control signal high" state of the memory block, whereby the output of the network is updated with the values previously stored in "Mem1", meanwhile ignoring any changes to the input to the memory block network.

A comparison with the naïve derivation of the RVC task's condition-consequence pairs (reproduced below for convenience) reveal that the oscillatory behaviour is not desired. Rather, the *task_stage* representation should only change when the "◀" character is presented to Spaun:

$$0.5 \times (task \bullet \textsf{\textbf{STORE\_IN}} + vision \bullet \blacktriangleleft) \longmapsto \textsf{\textbf{STORE\_OUT}} \Rightarrow task$$
$$0.5 \times (task \bullet \textsf{\textbf{STORE\_OUT}} + vision \bullet \blacktriangleleft) \longmapsto \textsf{\textbf{STORE\_IN}} \Rightarrow task$$

Spaun action selection hierarchy addresses this issue using the same 3-layer hierarchical structure: the first layer determines the state transitions based on the current state representation (as described previously); using visual information, the second layer generates the flow control signals; and finally, using the memory block networks, the third layer breaks the oscillatory feedback by enabling the controlled simultaneous reading and writing of the state representations. Figure 3.32 illustrates this hierarchical structure and demonstrates an example of how the task representation is updated.

Transformation Module Control Signals Example: The previous sections describe how the first layer of the action selection hierarchy is used to generalize common task stages into abstract task states, and how the second and third layer are supposed to be used to implement the desschematic ired actions (consequences) for each of the abstract task states. This example will demonstrate how this process has been accomplished for Spaun's transformation module, focusing primarily on how the second layer of the hierarchy is configured to generate the desired flow control signals.

This example examines two scenarios of the transformation module configuration during different task processing stages. In the first scenario, the flow control signals generated by the second layer of the action selection hierarchy configures the transformation module for the "**TRANSFORM1**" task stage. Referring to table 3.6, this task stage is common to both the RVC and fluid induction tasks, and it involves computing the transformation matrix between the semantic pointer representations stored in *mem*1 and *mem*2 (see condition-consequence pair (3.18)). The transformation module specific condition-consequence pair for this task stage is:

$$task\_stage \bullet \textsf{\textbf{TRANSFORM1}} - task\_decode \bullet \textsf{\textbf{INDUCTION}} \longmapsto mem2 \circledast {\sim}mem1 \Rightarrow output \tag{3.22}$$

Using this condition-consequence pair, the action selection hierarchy's second layer transform can be determined as illustrated in Figure 3.33.

In the second scenario, the flow control signals generated by the second layer of the hierarchy configures the transformation module for the "**INDUCTION**" task decode stage. The configura-

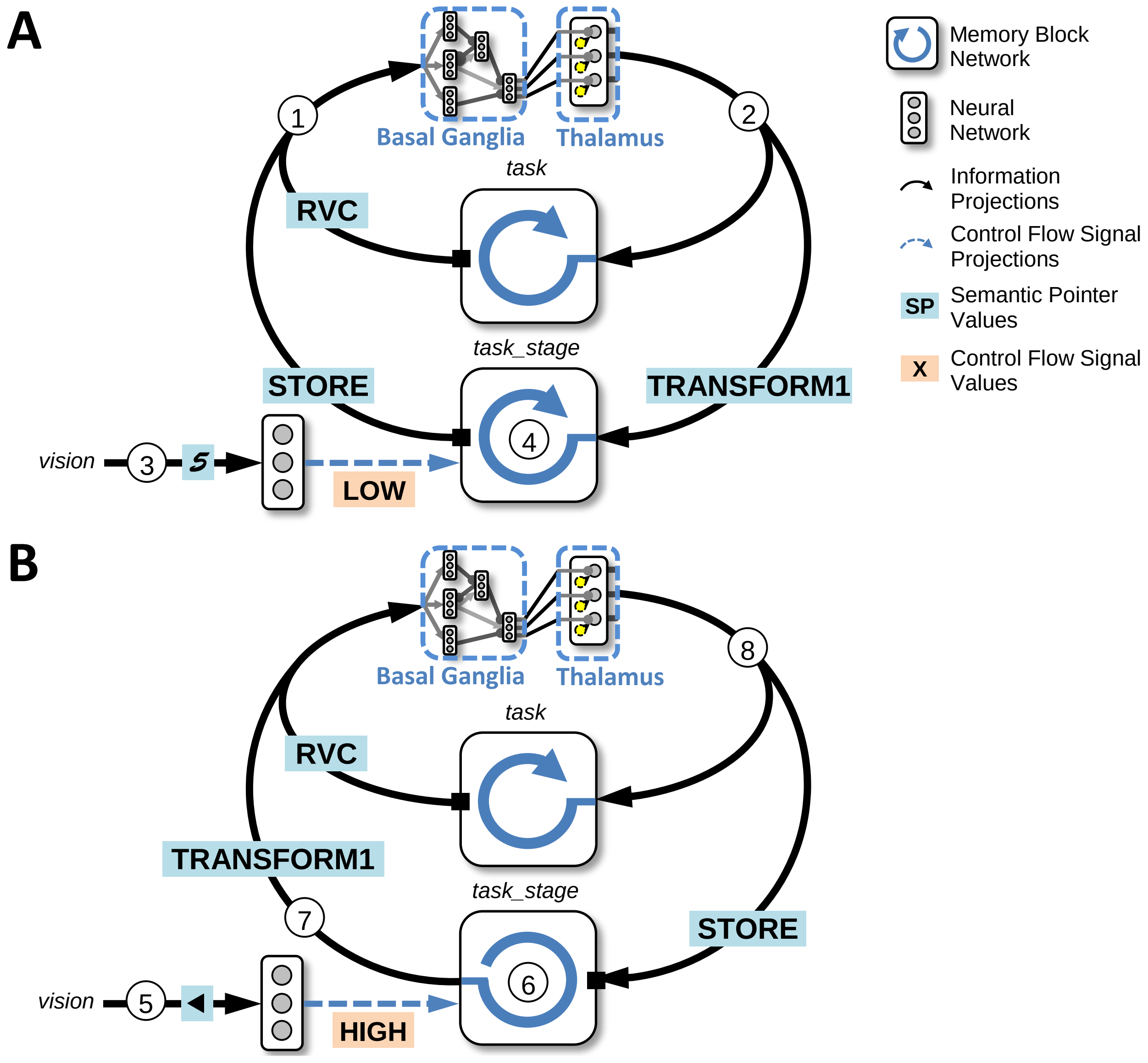


Figure 3.32: Example update operation of the task state representation in Spaun's action selection hierarchy. (**A**) In this scenario, the output of the *task* and *task_stage* memory blocks are **RVC** and **STORE**, respectively (①). This activates condition-consequence pair (3.20), which projects "**TRANSFORM1**" to the input of *task_stage* (②). However, because the visual system is being presented a "**5**" (③), the flow control signal value projected to *task_stage* is "low", resulting in no change to its output (④). Consequently, no oscillation happens in the *task_stage* representation. (**B**) In this scenario, the visual system is being presented with the ◄ character (⑤), resulting in a "high" value being projected as the *task_stage* flow control signal. This in turn causes the *task_stage* memory block to switch its state (⑥), updating its output value to "**TRANSFORM1**" (⑦). This activates condition-consequence pair (3.21), resulting in the projection of "**STORE**" to the input of *task_stage* (⑧). However, as the flow control signal is "high", the *task_stage* memory block ignores its input (see ⑥), once again preventing an oscillation in the *task_stage* representation.

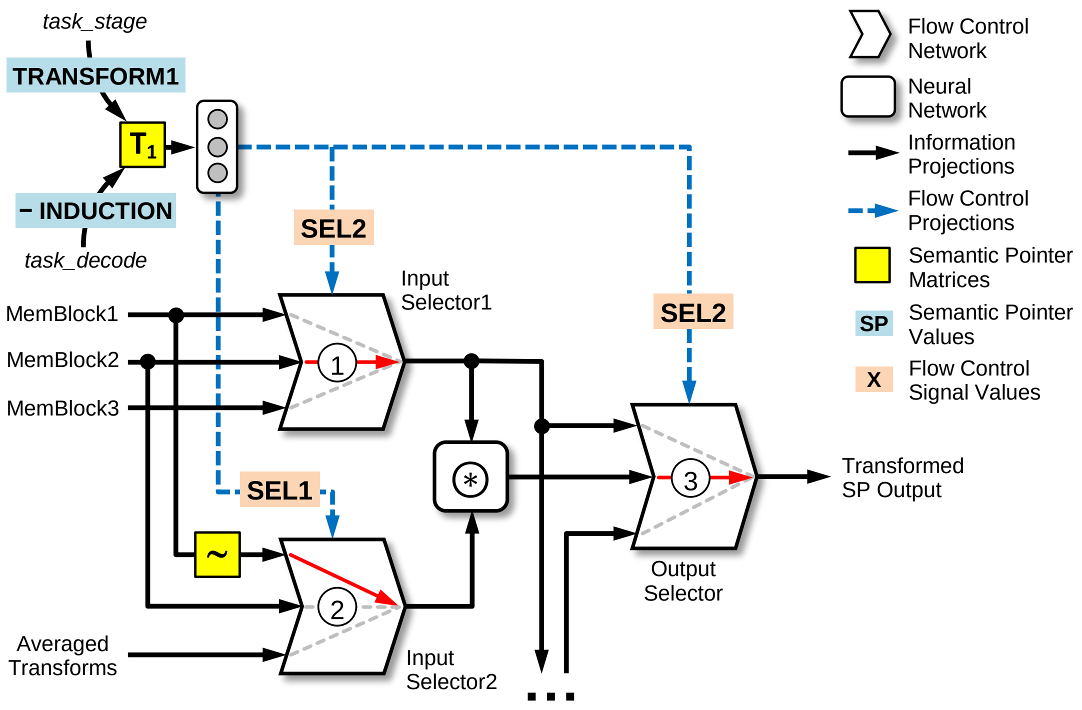


Figure 3.33: Illustration of the flow control signal configuration for the transformation module in the "**TRANSFORM1**" task stage. Here, the hierarchy's second layer transform ($\mathbf{T}_1$) is set such that when *task_stage* has a value of "**TRANSFORM1**", the "select2" signal is set for "input selector 1", the "select1" signal is set for "input selector 2", and the "select2" signal is set for the "output selector". This ensures that the SPA binding network receives inputs from $mem2$ (①), and $\sim mem1$ (②), and projects the output of the binding network to the output of the module (③), thereby configuring the transformation module to performed the desired consequence "$mem2 \circledast \sim mem1 \Rightarrow output$" (see (3.22)) for this task stage.

tion of the transformation module for this task decode stage is dependent on the stimuli Spaun received before the "**?**" prompt was presented. Here, Spaun is assumed to have been presented with the first digit list of the inductive problem (e.g., the C_7 list for the fluid induction task) before being prompted for an answer. This results in the value of *task_stage* being "**TRANSFORM1**" when the "**?**" prompt is presented. The transformation module specific condition-consequence pair for this task stage is:

$$0.5 \times (task\_stage \bullet \textsf{\textbf{TRANSFORM1}} + task\_decode \bullet \textsf{\textbf{INDUCTION}}) \longmapsto \\ mem1 \circledast \sim mem\_ave \Rightarrow output$$

Using this condition-consequence pair, the action selection hierarchy's second layer transform can be determined as illustrated in Figure 3.34.

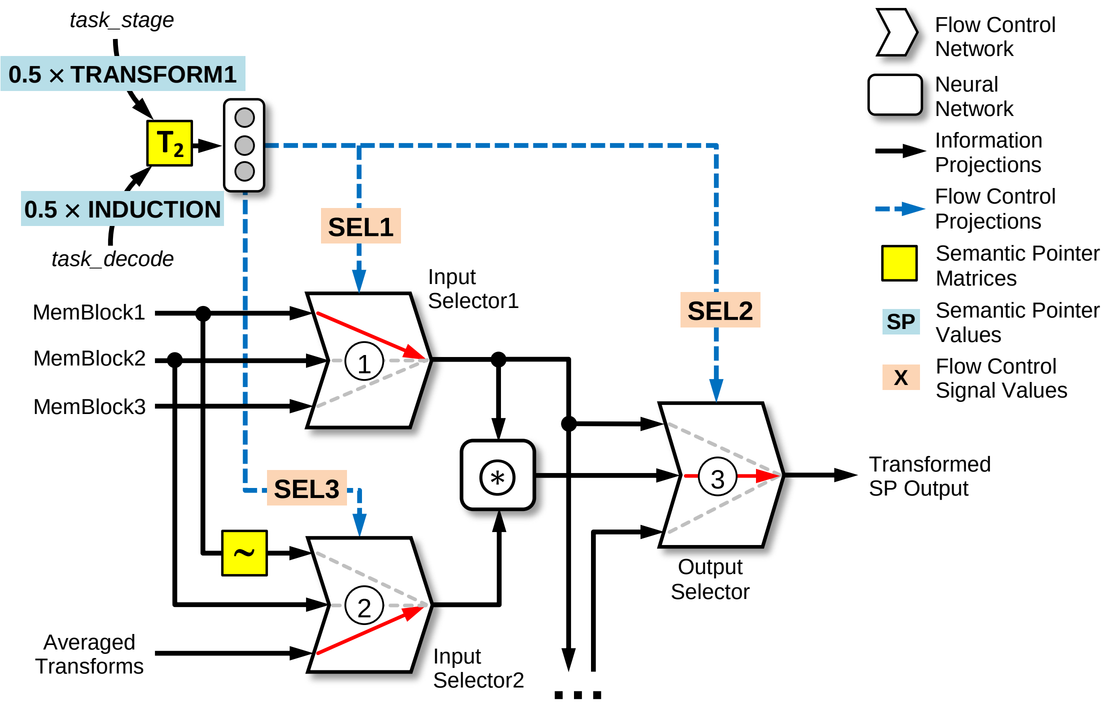


Figure 3.34: Illustration of the flow control signal configuration for the transformation module in the "**INDUCTION**" task decode stage. Here, the hierarchy's second layer transform ($\mathbf{T}_2$) is set such that when *task_decode* has a value of "**INDUCTION**", the "select1" signal is set for "input selector 1", the "select3" signal is set for "input selector 2", and the "select2" signal is set for the "output selector". This ensures that the SPA binding network receives inputs from *mem*1 (①), and the stored averaged transforms (②), and projects the output of the binding network to the output of the module (③), thereby configuring the transformation module to performed the desired consequence for this task stage.

Chapter 4

Spaun 2.0

The original Spaun model was completed in 2012, and was designed to be simulated using the Nengo (version 1.4) [Stewart et al., 2009] neural simulation software. In 2013, work began on a new version of the Nengo software (dubbed Nengo 2.0 [Bekolay et al., 2014]). Concurrently, it was decided to update the Spaun model for this new version of Nengo. While it would have been straight-forward to port Spaun directly from Nengo 1.4 to Nengo 2.0, a quick analysis of the Spaun code base revealed that in order to maximize the longevity and usefulness of the Spaun project, it was more beneficial to analyze and properly formalize the Spaun architecture, and completely rewrite the Spaun model in the Nengo 2.0 software.

The re-implemented Spaun model (henceforth referred to as "Spaun 2.0") is the focus of this section of this thesis. Improvements have been made to the majority of Spaun's systems to address concerns with the original model, and this section details the most important changes to each of Spaun's systems (excluding the encoding and decoding systems[32]).

Following the description of the updated Spaun 2.0 system, this section compares the results of the re-implemented Spaun 2.0 model with the original Spaun model to see how these changes have affected Spaun's performance on the eight original Spaun tasks.

4.1 Vision System

Architecturally, Spaun 2.0's vision system has not changed significantly from the original Spaun implementation. However, both the visual hierarchy and the stimulus change detector networks

[32]The encoding system did not receive substantial changes between Spaun and Spaun 2.0, while the changes made to the decoding system were not motivated by any of the critical issues of the original Spaun model.

(see Figure 3.21) have been modified to address important concerns about Spaun's original visual system.

4.1.1 Visual Hierarchy

Issue Description: The original Spaun visual hierarchy was constructed using the methods outlined in Section 2.5.6.2, whereby a hierarchical (non-spiking) neural network is trained using conventional machine learning techniques and then "converted" into a spiking neural network by using neural ensembles to approximate the non-linearities used in each node of the original hierarchy. While this network proved sufficient for Spaun's purposes, the non-linearities in the hierarchical network required approximately 200 spiking LIF neurons to approximate well enough for the spiking neurons to achieve the same level of classification accuracy as the original non-spiking network.

Considering that Spaun's original visual hierarchy comprised a total of 1,850 nodes (see Section 3.1.1), the spiking network equivalent required 370,000 spiking LIF neurons, which is roughly 16% of Spaun's 2.3 million neuron count (this count excludes the visual hierarchy – refer to Section 4.6.1 for a detailed breakdown of the neuron count). In an effort to reduce the overall simulation time of the Spaun model, the total neuron count of the running mode was reduced by configuring the visual hierarchy to run in a purely function-based (i.e., non-spiking) mode.

Updated Implementation: Conceptually, the machine learning techniques used to train the hierarchical networks (similar in type to the one used in Spaun's visual system) do not place restrictions on the types of functions that can be used as the individual node's non-linearities – with the exception that the functions need to be continuous and differentiable. Unfortunately, the standard LIF neuron tuning curves (see Figure 2.13, and Figure 4.1) violate the differentiable requirement, as they contain a disjoint transition at the neuron firing threshold.

In 2015, Hunsberger and Eliasmith [2015] successfully trained a hierarchical network using a modified LIF neuron (called the "softLIF" neuron) which has a continuously differentiable tuning curve (see Figure 4.1). With this modified tuning curve, they were able to train a 2-layer hierarchical network that could perform with the same classification accuracy as Spaun's original visual hierarchy. Importantly, because the tuning curves of the softLIF neurons are almost identical to their LIF counterparts, a spiking version of the hierarchical network can be constructed by replacing all of the softLIF neurons with equivalent LIF neurons, without the need of modifying any of the trained neural connection weights, and without requiring additional neurons to represent the sigmoid non-linearity (as was the case in the original Spaun model) [Hunsberger and Eliasmith, 2015].

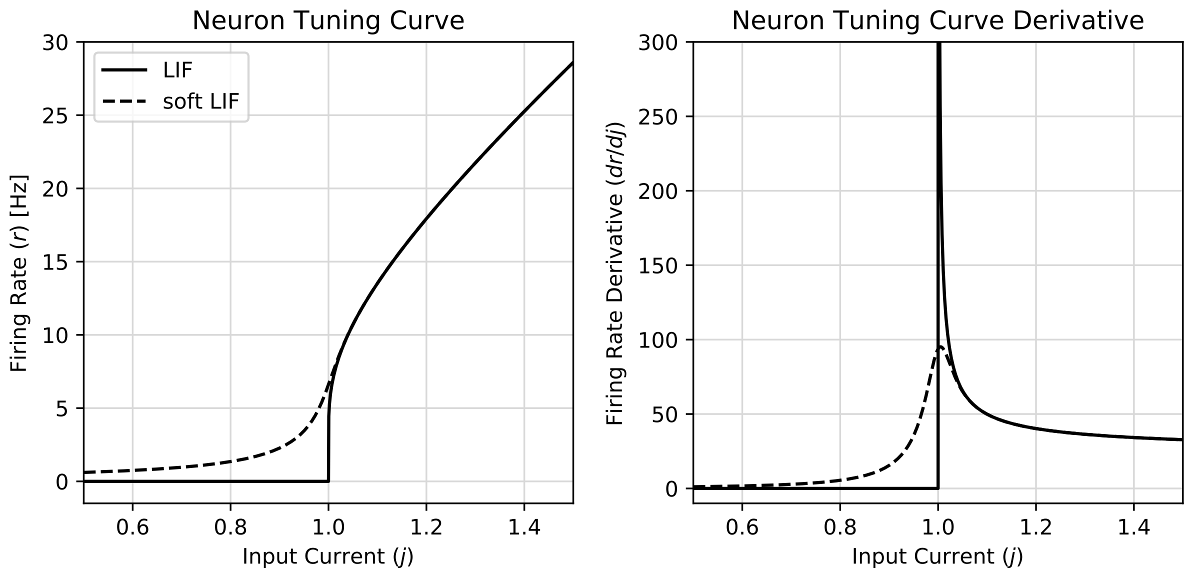


Figure 4.1: Tuning curve (left) and tuning curve derivative (right) plot comparison between the LIF and softLIF neuron types. Note that while the derivative of the LIF neuron tuning curve is infinite at the firing threshold (i.e., when the input current $j = 1$), the derivative of the softLIF neuron tuning curve remains well defined.

Spaun 2.0's updated visual hierarchy is identical to the softLIF hierarchical network, with the exception that it has been trained to classify Spaun's control characters (see Section 3.2.1.1) as well. The final implementation of the visual hierarchy contains 2 layers of spiking LIF neurons, which contain 500 neurons and 200 neurons, respectively. Using the MNIST dataset, the reported classification accuracy of Spaun's original visual hierarchy (non-spiking) is 98.76% [Tang and Eliasmith, 2010], while the fully spiking LIF visual hierarchy achieved a classification accuracy of 98.37% [Hunsberger and Eliasmith, 2015]. It is important to note that both of these accuracy figures are just for the hierarchical networks, and exclude any impact Spaun's remaining systems (particularly the working memory system) might have on its performance.

4.1.2 Stimulus Change Detector

Issue Description: As mentioned previously, the Spaun model is primarily driven by visual stimuli, to the extent that Spaun will not make any progress on the task it is performing if no changes are detected in the visual input. It is thus crucial for Spaun to employ a robust mechanism for detecting changes in the visual stream so that the appropriate control signals can be generated and propagated to the rest of the Spaun network (see Section 3.3.1).

The original Spaun model achieved this by containing a network specifically designed to detect when the visual input is "blank" (i.e., contains no bright pixels). This was achieved by training the visual network to classify the "blank" input as a Spaun specific control character (i.e., associating it with its own semantic pointer representation). A simplified associative memory network was then used to detect the presence of the "blank" semantic pointer (at the output of the visual semantic pointer classifier), and produce a "high" signal value, which is then used to generate the control signals for the rest of Spaun's systems.

While robust, this method of detecting changes in Spaun's visual stimulus introduced the artificial constraint that a "blank" character is needed to separate *all* of the items in the stream of visual images – a constraint that, to someone without full knowledge of the internal workings of the Spaun network, seems to be an unnecessary addition.

Updated Implementation: Unlike the original Spaun model, Spaun 2.0's visual system does not rely on a "blank" stimulus to determine if changes have occurred to the visual image stream. Instead, the updated stimulus change detector network uses a differentiator circuit to detect changes in the visual stimulus.

Conceptually, the differentiation operation (i.e., derivative) for a time-varying signal is defined as the difference between the input signal and a time-delayed version of the input signal (normalized by the amount of temporal delay), as the size of the temporal delay approaches 0. Implementing a differentiator network in neurons thus requires three inputs: the input signal, a delayed version of the input signal, and a scaling factor to account for the normalization of the signal difference.

Computing the signal that is the exact delay of the input is possible [Voelker and Eliasmith, 2015], however, for the purposes of the change detect network, the exact delay is unnecessary. Rather, the change detect network uses projections with differently valued synaptic time constants to achieve the approximation of the delay. As illustrated in Figure 4.2, for a neural ensemble that is provided with a step input, and using the first-order exponential synapses used in the Spaun model, a longer synaptic time constant results in the output signal of the neural ensemble taking longer to reach the value of the stepped input. Essentially, the longer synaptic time constant has "delayed" the time it takes for the output signal to reach the value of the input.

By using differing post-synaptic time constant values to generate a "pass-through" and "delayed" version of the input signal, the differentiator circuit can then be constructed by subtracting the "delayed" signal from the "pass-through" signal [Tripp and Eliasmith, 2010]. Figure 4.3 demonstrates this circuit, and provides a graph of the expected output of the differentiator when

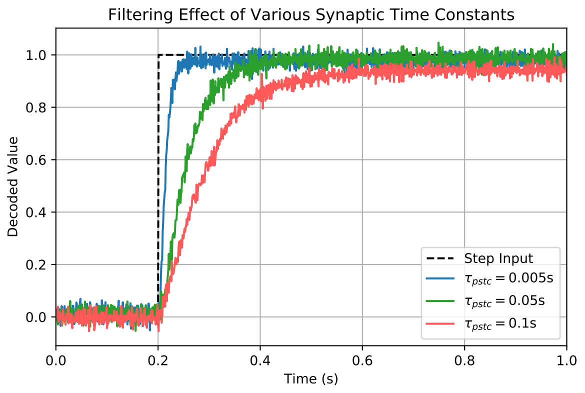


Figure 4.2: Plot of the decoded output response of an ensemble to a given step input. The decoded output plots are shown for varying values for the input post-synaptic time constant, ranging from 0.005s to 0.1s.

it is provided with a step input. It should be noted that because the exact derivative of the input signal is not required for the change detect network, the normalization scaling factor can be arbitrarily set (for the circuit in Figure 4.3, it is set to 1).

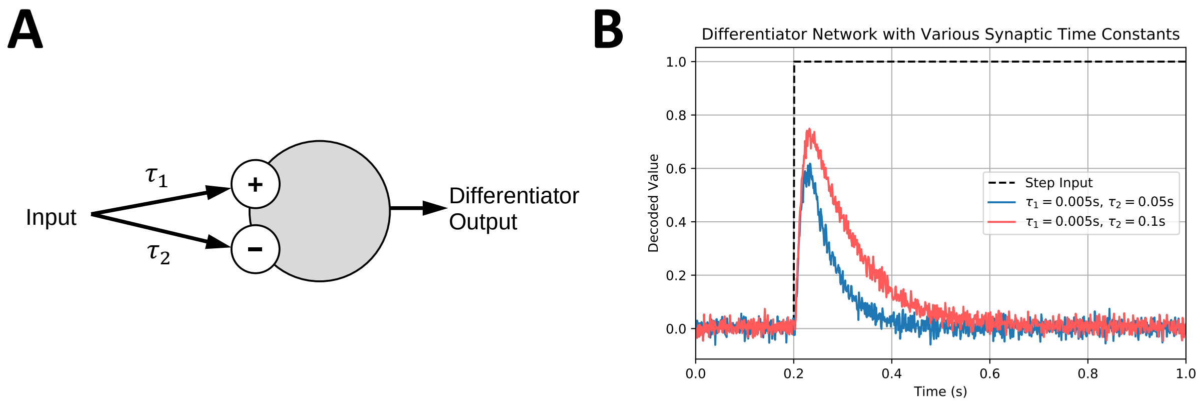


Figure 4.3: Schematic and decoded output plots of the differentiator circuit. (**A**) Network schematic of the differentiator circuit. The input is projected to the same neural ensemble through two projections with different values for their synaptic time constants. The projection with the longer synaptic time constant (τ_2) has a negative transform matrix to subtract its value from the input signal. (**B**) Decoded output value plots for two different differentiator circuits. Both have identical "fast" synaptic time constants ($\tau_1 = 0.005$s), with different "delayed" synaptic time constants ($\tau_2 = 0.05$s and $\tau_2 = 0.1$s, respectively). It is observed that a bigger relative difference between τ_1 and τ_2 results in a larger output "spike".

To construct the full change detect circuit, a bank of differentiator circuits is constructed, each receiving a projection from different pixels in Spaun's input stimulus image.[33] Next, the outputs of the differentiator array is projected to another neural ensemble to sum and produce the final change detect "spike" output. Because the output of the differentiator circuit can be both a negative and positive spike, depending on the "direction" of change exhibited by the input signal, the projections from the differentiators to the output ensemble are configured to compute the absolute value function $f(x) = abs(x)$. Finally, the goal of the change detect network is to indicate when the stimulus has changed from one image to another. Thus, to avoid the appearance of two spikes when blank input stimuli (the blank input is not considered a "proper" image) are used[34], the output of a "blank detect" circuit is also projected to the output ensemble. Because the change detect circuit operates using the individual pixel values in the input image, the blank detection circuit is constructed by projecting the sum of all of the pixel values into an inhibited biased ensemble (see Section 2.5.6.5) with thresholded tuning curves. With this setup, the output of the blank detection circuit will be "high" when a blank is detected, and "low" when a non-blank image is presented. Figure 4.4 shows the schematic of the final change detect network, and demonstrates the behaviour of the network to different stimuli presentation sequences.

4.2 Motor System

As with the visual system, the architecture of the motor system (see Figure 3.22) has not substantially changed between the Spaun and Spaun 2.0 models. However, the majority of the Spaun's original motor system was not implemented in a neural architecture, and this forms the primary cause of the issues addressed by the Spaun 2.0 model. In the original Spaun model, the use of a non-neural motor system affected both the motor timing controller and the motor hierarchy (arm controller).

4.2.1 Motor Timing Controller

Issue Description: In the original Spaun model, the motor timing controller had two responsibilities:

- Ensuring the timing requirements for the memory networks are met for the internally-controlled portion of the counting task.

[33]To reduce the number of neurons needed to implement the change detector circuit, the bank of differentiators can also be connected to a randomly selected subset of the input pixels, rather than to all of the pixels.

[34]"Blank" input images are still used to differentiate the appearance of two identical input images.

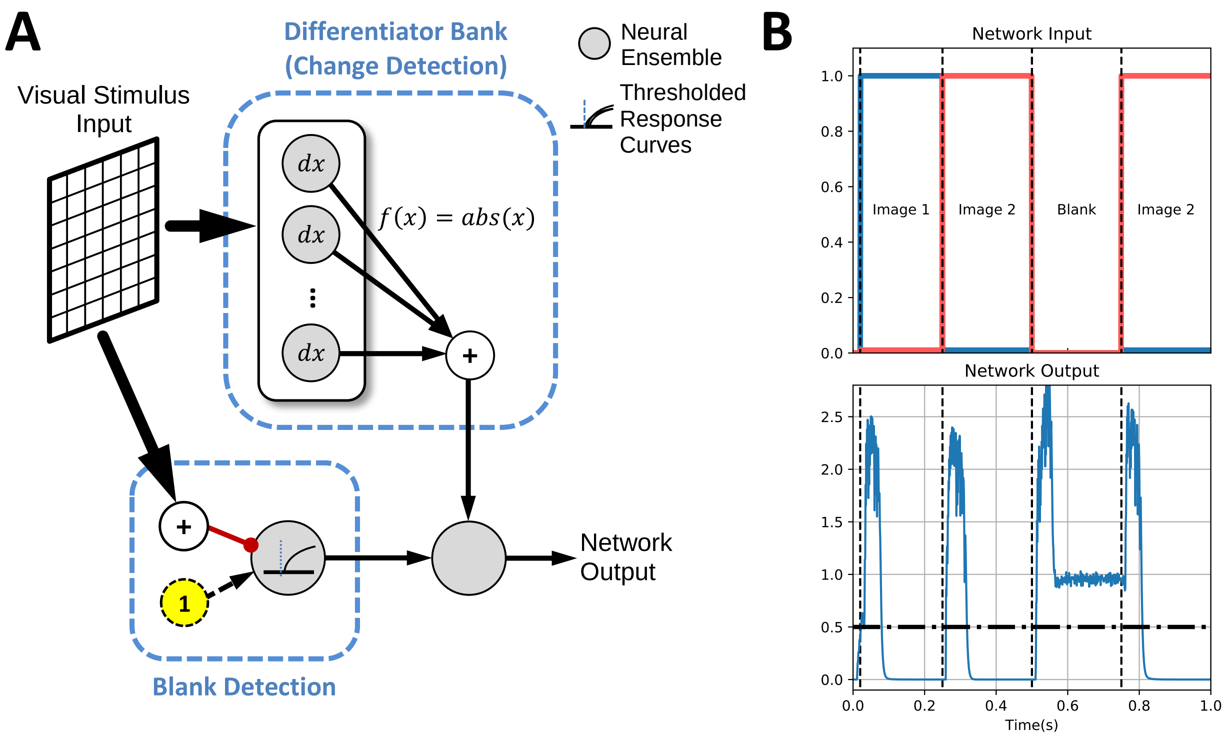


Figure 4.4: Schematic and recorded input and outputs of the change detector network used in the visual system of the Spaun 2.0 model. (**A**) Network schematic of the full change detector (combined with the blank detector) network. See accompanying text for specific details of the network implementation. (**B**) Recorded behaviour of the change detector network. Four stimulus images were presented to the change detect network, the first image ("Image 1") is followed immediately by the second image ("Image 2") at $t = 0.25$s. Next, a blank stimulus is presented at $t = 0.5$s. This is done to separate the independent instances of the sequential presentation of Image 2, of which the second presentation occurs at $t = 0.75$s. (Top) The image sequence presented to the change detect network. (Bottom) The output recorded from the change detect network. When used to generate flow control signals for the rest of the Spaun model, a value above 0.5 (dot-dashed line) is considered to be indicative of a change having occurred (or is currently occurring). Note that the inclusion of the blank detect circuit ensures that only one "change occurred" signal is conveyed to the rest of the Spaun network (because the output remains above 0.5 while the blank stimulus is being presented).

- Approximating the time taken to write a digit. This was necessary as the motor hierarchy was not implemented as part of the Spaun neural network, and thus outputs from the motor hierarchy could not be fed back to the Spaun model to generate the appropriate control signals.

To simplify the implementation of the motor timing controller, and efficiently meet the two requirements, a ramp signal (generated by an integrator network) was used. Because the ramp output of an integrator rises as a fixed rate, the time taken for the ramp signal to reach a pre-determined value can be used as a time duration to approximate the action (i.e., writing a digit) of the motor hierarchy. Additionally, the ramp signal can be used to generate the control signals for the counting task requirement.

While the ramp signal is sufficient to generate the appropriate control signals, and to "time" the actions of the motor hierarchy, for it to function appropriate for the sequential production of output digits, it needs to be reset after each digit is produced. In the original Spaun model, this is achieved by using a non-neural functional node[35] to detect when the ramp signal has crossed a pre-determined threshold, and to inhibit (reset) the ramp network when this occurs (see Figure 4.5). Because one of the goals of the Spaun 2.0 model is the use of a fully realized neural network, the "detect-and-reset" functionality of the aforementioned non-neural node has to be replicated in a neural network.

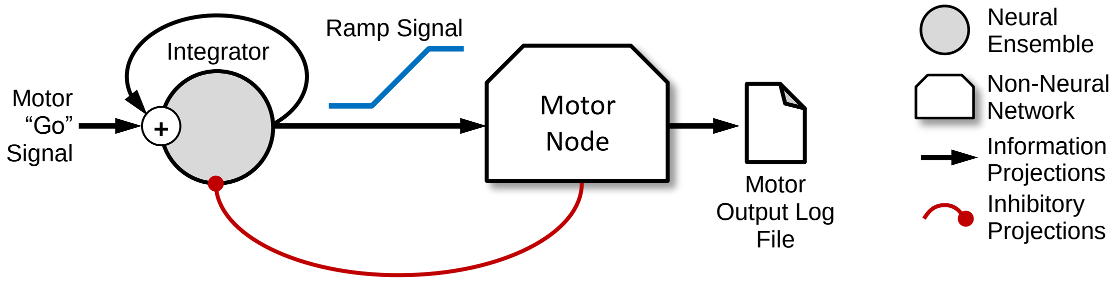


Figure 4.5: Schematic of the motor timing control circuit used in the original Spaun model.

Updated Implementation: The "detect-and-reset" functionality required by the motor timing controller can be naturally divided into two separate circuits, one to detect the ramp signal crossing the pre-determined threshold value, and another to generate the inhibitory signal necessary to reset the integrator.

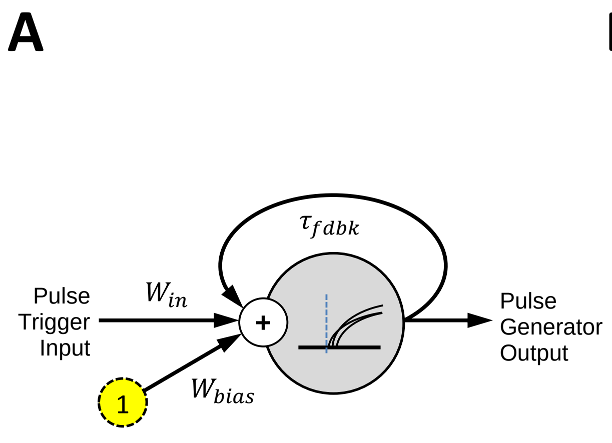
[35] A non-neural functional node was used in the original Spaun model as it was necessary to use the ramp timing signal to write the motor outputs to file, so that it could be processed by the external (coded in MATLAB®) motor hierarchy at a later time.

To detect when the ramp signal has crossed a pre-determined threshold, an ensemble with thresholded response curves (similar to the ensembles found in the associative memory network) is used. The threshold of the neurons' response curves are configured such that they are only active when the input to the neurons (i.e., the ramp signal) crosses the pre-specified value. Because of the dynamics within the integrator circuit (the synaptic filter in the feedback loop), the inhibitory signal must be presented for a minimum amount of time for the integrator value to fully reset. Since the threshold circuit is directly driven by the output of the integrator, using its output to reset the integrator instead results in the network reaching a steady state, alternating between the integrator driving the thresholded output and the thresholded output barely inhibiting the integrator.

Decoupling the output of the ramp integrator from the reset signal generation involves the use of another memory (integrator) circuit. The goal of this memory circuit is to, upon presentation of the thresholded input, "remember" to generate an output signal for a fixed amount of time. Given the description of its behaviour, this circuit will henceforth be referred to as a "pulse generator". In the Spaun 2.0 model, the pulse generator circuit is implemented using a modified integrator circuit. First, the synaptic time constant on the feedback projection is reduced to ensure a fast rise time of the output pulse. Next, a thresholded ensemble is used to ensure that the pulse is not prematurely triggered, and to ensure that when the pulse output drops below a certain value, no neural activity will be output. This is to make sure that the ramp circuit is not accidentally inhibited by the output of the pulse generator. Finally, a small negative bias is introduced to the pulse generator, to force the pulse output to decay over a fixed time. The size of the negative bias can be altered to reduce or lengthen the shape of the pulse. Figure 4.6 illustrates the pulse generator circuit, its behaviour when the negative bias is not provided, and its behaviour when properly configured.

With both the "detect" and "reset" circuit designs, the core of Spaun's motor timing circuit can be constructed.[36] As illustrated in Figure 4.7A, the motor timing circuit is assembled by using the output of the ramp integrator to drive the output of the threshold "detect" circuit, which in turn is used to drive the pulse generator circuit, that finally inhibits the ramp integrator. Figure 4.7B demonstrates how the ramp signal and the pulse signal activate in sequence to generate the cyclic behaviour of the motor timing circuit during the Spaun's internal counting process.

[36]The full motor timing network contains additional circuitry to ensure that the ramp signal operates in synchrony with the actions of the arm's end effector. For the sake of brevity, this is not discussed in the text above.



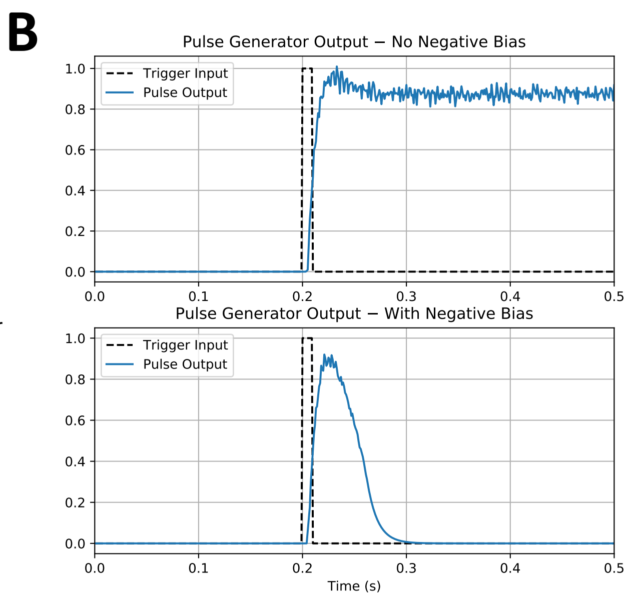


Figure 4.6: Schematic and example outputs of the pulse generator circuit used in the Spaun 2.0 model. (**A**) Diagram of the pulse generator circuit. See accompanying text for the details of its design. (**B**) Recorded outputs of the pulse generator circuit. For these plots, the pulse generator is configured with $\tau_{fdbk} = 0.005$s, $W_{in} = 5$, $W_{bias} = -0.1$, and a neuron threshold of 0.07. (Top) Output of the pulse generator when no negative bias is introduced. Note that the trigger input (dashed line) is able to successfully initiate the pulse, but when it is removed, the pulse output remains stable. (Bottom) Output of the pulse generator when a negative bias is introduced. The trigger input is able to initiate the pulse, which gradually decays after the trigger input is removed. Note that the width of the pulse can be modified by adjusting the value of W_{bias}.

4.2.2 Motor Hierarchy (Arm Controller)

Issue Description: As previously mentioned, the original Spaun model did not contain an embedded arm controller. Rather, the motor semantic pointers generated by the decoding system were recorded to file by the motor timing controller (see Section 4.2.1), and then processed by an external MATLAB® script simulating the motor hierarchy and a physics based model of Spaun's "arm", only after the Spaun simulation had completed. To further the effort of implementing the entire Spaun 2.0 model in networks of spiking neurons, a neural implementation of the arm controller is required.

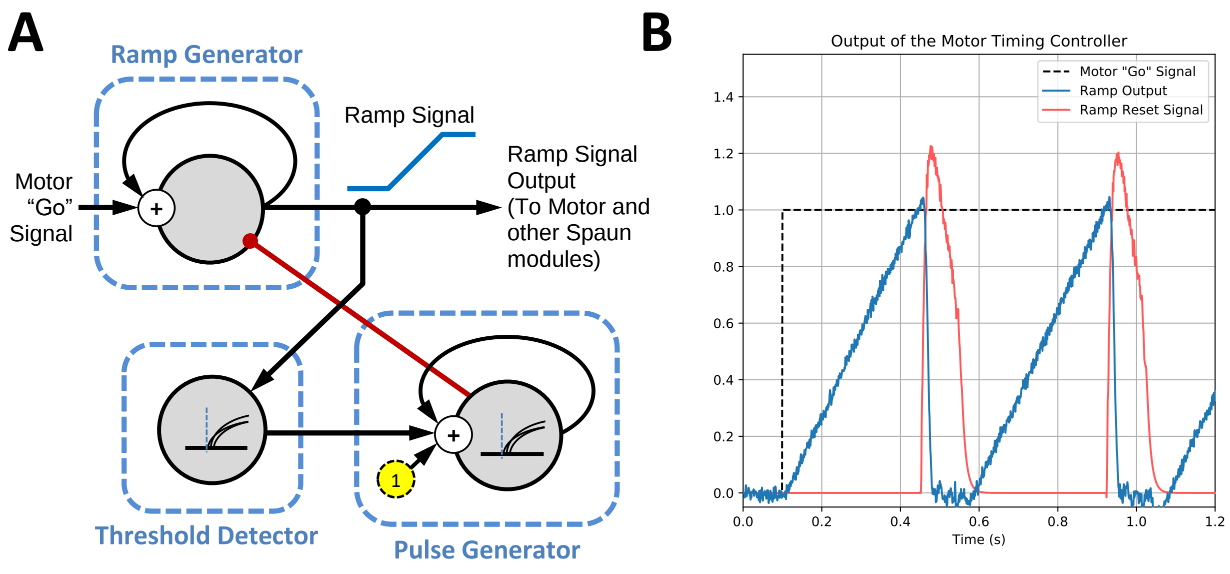


Figure 4.7: Schematic and recorded output of the core of Spaun 2.0's motor timing network. (**A**) Schematic of the Spaun 2.0's motor timing circuit. See accompanying text regarding the design of the network. (**B**) Example output of the motor timing network during the internally-driven portion of the counting task. At $t = 0.1$s, the motor "Go" signal goes to a value of 1, causing the ramp generator to start producing the ramp output ("Ramp Output"). When the ramp output exceeds a value of 1, the threshold detect circuit causes the pulse generator to generate the pulse signal used to reset the ramp ("Ramp Reset Signal"). As observed, when the ramp reset signal is active, the ramp output is nullified. When the ramp reset signal dissipates, the "Go" signal causes the ramp generator to restart the ramp, initiating another cycle of the network. The cyclic behaviour continues until the motor "Go" signal is removed.

Updated Implementation: The motor hierarchy used in the Spaun 2.0 model is based on the recurrent error-driven adaptive control hierarchy (REACH) neural model [DeWolf, 2014], modified to function within the Spaun ecosystem. The REACH model consists of two sub-components, the dynamic motor primitives (DMP) component, meant to represent the brain's pre-motor cortex, and the operational space controller (OSC), meant to represent the M1 and cerebellar regions of the brain.

Functionally, the DMP network takes a ramp signal, and an end point (in 2-D Cartesian space), and generates a 2-dimensional point that, over the course of the ramp signal, specifies a trajectory that traces out the figure the motor system is attempting to replicate. It does this by using the ramp signal to generate a forcing function that morphs (forces) the typically straight-line path of the point attractor network into the desired shape as it travels from its current position to the specified end position. The 2-dimensional point generated by the DMP system is the projected to the OSC system, which combines it with information about the current position of the arm's end effector to compute the low-level (muscle) control signals necessary for the arm to move to the point desired by the DMP system.

While the REACH system is a neural network that can be used to accurately control Spaun's arm simulation, a major hurdle is introduced by the specification of one of Spaun's tasks. Mathematically, the generation of the 2-D Cartesian (x, y) point though the use of the forcing functions is:

$$[x, y] = [FF_x(v), FF_y(v)],$$

where FF_x is the forcing function used to generate the "x" component of the Cartesian point, FF_y is the forcing function used to generate the "y" component of the Cartesian point, v is the instantaneous of the ramp input. For known trajectories (known when the model is constructed), this can be achieved in a neural ensemble by solving for a set of decoders that satisfy the desired forcing function. However, the core algorithm of Spaun's copy-drawing task uses a matrix computation to determine the trajectory to be used. This implies that a neural solution is required where the neural network solves the forcing function given the desired ramp value v, and this problem is substantially more complex than solving for the appropriate neural decoders. While it is theoretically possible to implement a solution through the use of vector-like function space representations[37] [Eliasmith and Anderson, 2003], until recently, the Nengo 2.0 simulation software did not fully support this form of representation.

Instead, the "DMP" network used in the Spaun 2.0 model borrows a technique from the original Spaun model. Rather than using a function to represent the desired trajectory, a sampled version of the trajectory is used, and the "DMP" network merely outputs the point along the trajectory that corresponds to the amount of progress made by the ramp signal. Spaun 2.0 achieves this by using a "function evaluator" network (details on the exact implementation of this network can be found in Spaun 2.0's code base – see Appendix B.1). Note that because this network is not a true implementation of the DMP system, it is referred to as the "trajectory evaluator" network instead.

Fortunately, the OSC network from the REACH model can be used in the Spaun architecture without any modifications. Thus, the final motor hierarchy of the Spaun 2.0 model consists of the trajectory evaluator network, which projects to the OSC network. Figure 4.8 shows the network schematics of a hypothetical DMP-based implementation of Spaun 2.0's motor hierarchy (naturally, it would be unable to perform the copy-drawing task), as well as the final trajectory-

[37]Each component in the vector represents a coefficient of a basis function. Weighting the basis functions by their respective coefficients and combining them together results in the equivalent function form of the vector representation.

evaluator-based implementation of the Spaun 2.0 motor hierarchy.

4.3 Working Memory

While the general network layout (see Section 3.3.5) of the working memory has not changed significantly between Spaun and Spaun 2.0, the specific implementation of several memory networks within the working memory module has been altered to address several shortcomings of the original Spaun implementation. These affect both the list memory networks (Section 4.3.1) and the averaging memory network (Section 4.3.2).

4.3.1 List Item Memory

Issue Description: While the integrator network described in Section 2.5.6.1 can be used as a memory network, it requires precise stimulus presentation durations to store a specific value. As an example, to store a value x, a "standard" integrator ($A' = \tau_{fdbk}$, $B' = 1$) requires the input value x to be presented for exactly 1 second.

To circumvent this problem, the integrator networks in the OSE model used the difference between the input value (desired value to be stored) and the value being stored in the integrator as the input to the integrator (see Figure 4.9A). With this configuration (hereby referred to as the "gated difference integrator"), the value stored in the integrator will always approach the input value, regardless of how the input value changes (positive or negative) with respect to the value stored in the integrator. In addition, a weighting factor can be introduced (see Figure 4.9A: W_{in}) to adjust the time required for the integrator value to equalize to the input value (see Figure 4.9B). When no further changes to the stored integrator value is desired, the difference-calculation population is inhibited, resulting in no further inputs to the integrator (and the integrator holding its value). Assuming the integrator is given enough time to equalize to the input value, this operational mechanism removes the strict dependence on the presentation duration of the input signal, removing the need for additional timing circuitry to be implemented.

The gated difference integrator network was used as Spaun's prototype integrator network. However, in an effort to reduce Spaun's total network simulation time (which took 2.5 hours to simulate 1 second of the network [Eliasmith et al., 2012]), the presentation duration of each symbol of Spaun's input stimuli was reduced from 500ms (the OSE model's standard) to 150ms. This change revealed that regardless of the value of the scaling factor W_{in}, the gated difference integrator network was not capable of accurately storing a value with a presentation duration of 150ms (see Figure 4.9B, graphs III and IV). For this reason, a different integrator network design (the gated feedback integrator) was used in the final version of the Spaun model.

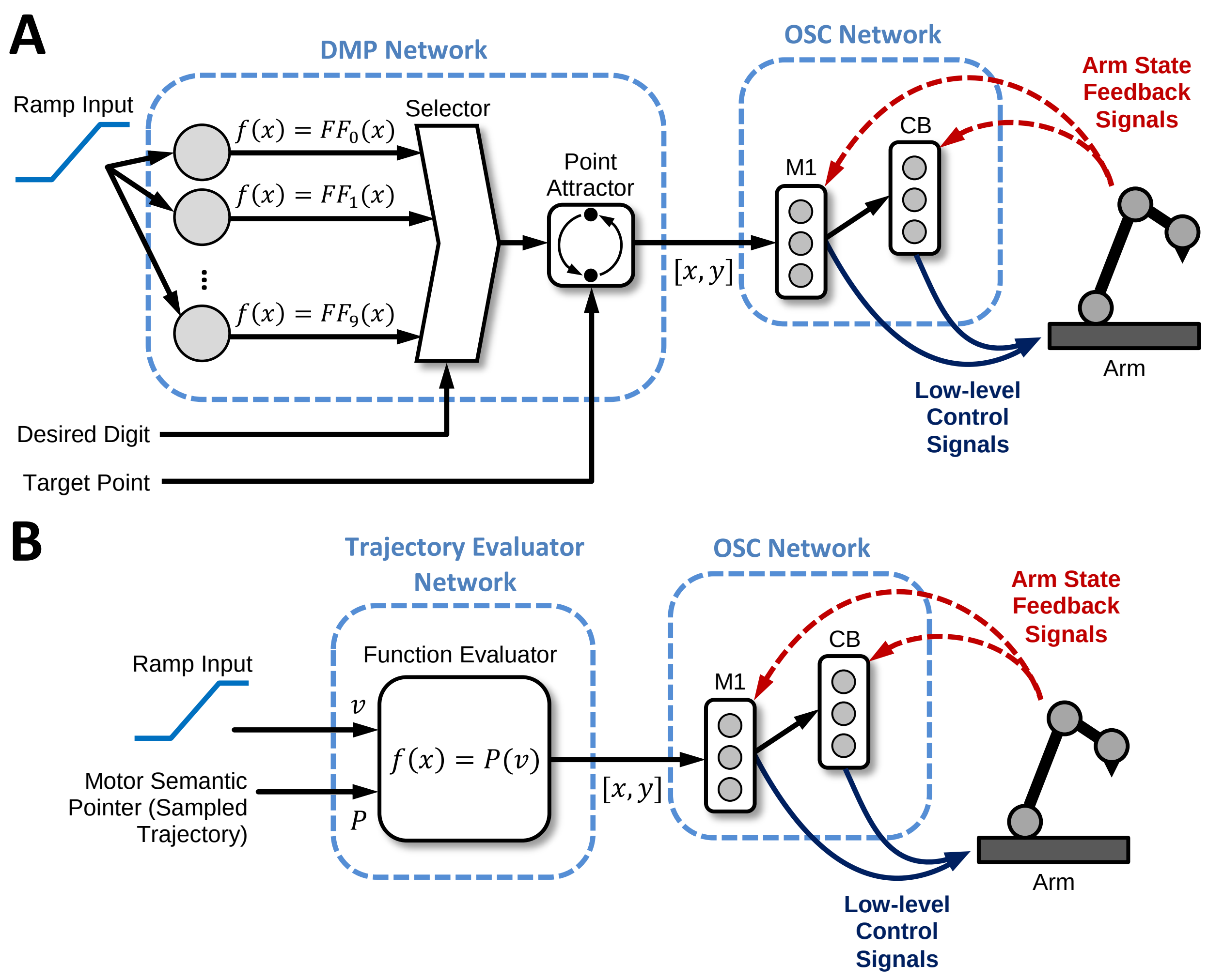
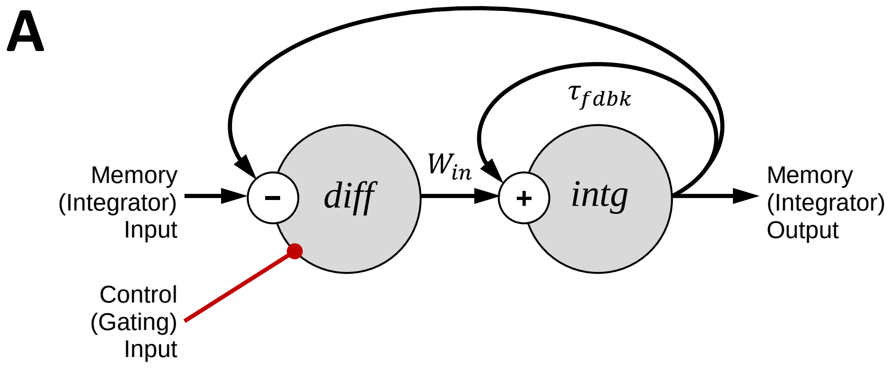


Figure 4.8: Two possible implementations of Spaun 2.0's motor hierarchy. (**A**) Hypothetical DMP-based implementation of the motor hierarchy. Here, the ramp signal is used to generate the forcing functions (FF_0 to FF_9) for each of Spaun's output digits. A selector then used to select the forcing function for the desired digit (determined by the decoding system) and projects it to a point attractor which generates a 2-D Cartesian point along the digit's trajectory. This information is used by the OSC network to move the arm along the provided trajectory, causing it to trace out the desired digit. (**B**) Current implementation of the Spaun 2.0 motor hierarchy. The value of the ramp signal (v), along with the desired motor semantic pointer (P – which is a sampled version of the desired trajectory) is projected to the function evaluator network which computes the function $P(v)$. This generates a 2-D Cartesian point along the digit's trajectory, which is fed to the OSC network as before, enabling Spaun's arm to reproduce the desired digit.

A



B

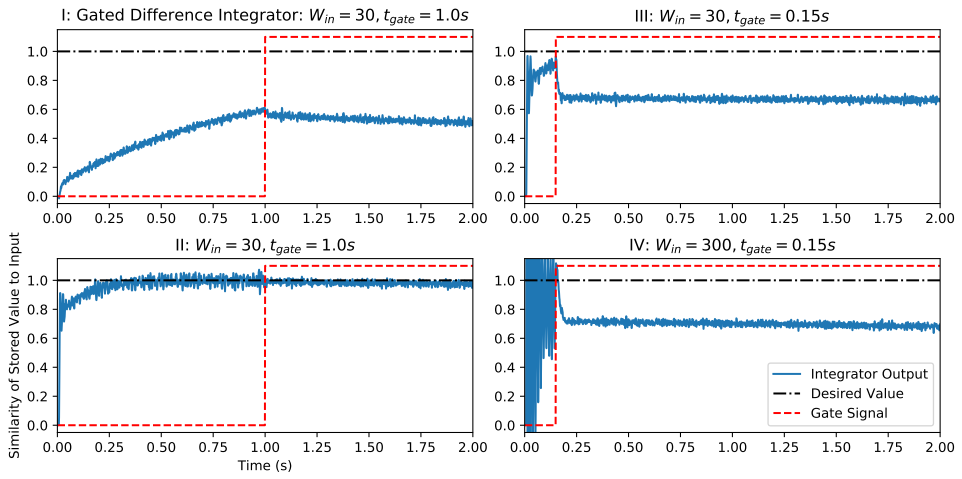


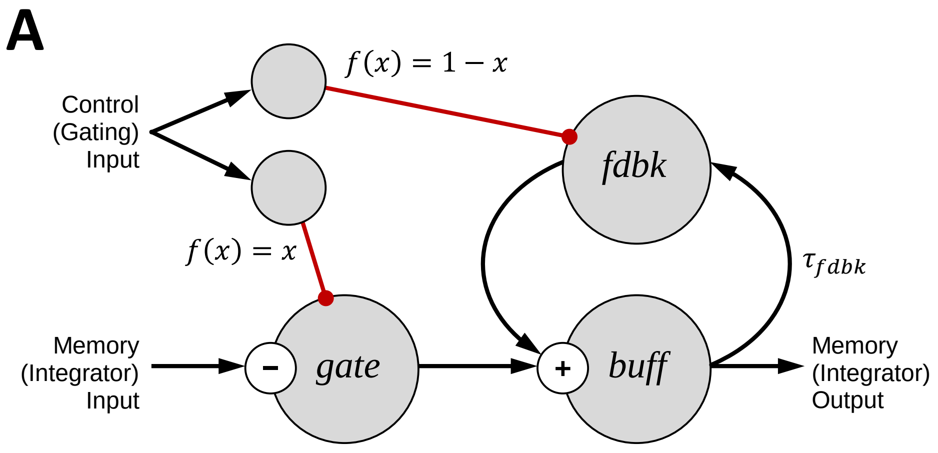
Figure 4.9: Schematic and output graphs for the various configurations of the gated difference integrator. (**A**) Schematic of the gated difference integrator. The gated difference integrator consists of the integrator population (*intg*) and a differencing population (*diff*), which calculates the difference between the integrator output and the input to the memory. When the gating input is "high" the difference population is inhibited, resulting in no input values fed to the integrator, and it holding its value. (**B**) Recorded output graphs for various gated difference integrator configurations. For each of the different integrator configurations, $\tau_{fdbk} = 0.1s$. (I) The "default" configuration, where the difference scaling factor (W_{in}) is 1, and the gate signal is set "high" at $t = 1s$. Note that for this configuration, 1 second is insufficient for the output of the integrator to match the desired input value. (II) Increasing the difference scaling factor to 30 enables the gated difference integrator to accurately store the desired input value. (III and IV) For these plots, the gate signal is set after only 150ms (see text for details), and regardless of the difference scaling factor (W_{in} = 30 and 300, respectively), the network is unable to accurately store the desired input value.

The gated feedback integrator network operated differently compared to the gated difference integrator network. Rather than having the stored integrator value approach the input value asymptotically, the gated feedback integrator used two differentially gated population to control the flow of information input to the integrator, and in the integrator's feedback loop (see Figure 4.10A). In the first mode of operation, the feedback population (*fdbk*) is inhibited, and the remaining uninhibited populations (*gate*, *buffer*) behaves exactly like a communication channel (i.e., the networks represent the input value with no change). Inhibiting the feedback population also interrupts the flow of information in the feedback loop, allowing the value represented by the integrator population (*buffer*) to quickly change, thus allowing the network to meet the 150ms input stimulus duration requirement. In the second mode of operation the input population (*gate*) is inhibited, and the feedback population (*fdbk*) is dis-inhibited. This allows any value being represented in the integrator to be fed back on itself, with the combined feedback and integrator populations (*fdbk*, *buffer*) to behave like a standard integrator.

While the gated feedback integrator is able to load values within the 150ms requirement, it does have a couple of disadvantages. Because the inhibition of the input and feedback populations (*gate*, $feedback$) is not instantaneous, and because the propagation of values between the neural populations is limited by the synaptic time constant, the value stored in the gated feedback integrator network is subject to a consistent drop when compared to the desired input value. Additionally, over the same interval of time, the value stored in the gated feedback integrator decays faster than the gated difference integrator (see Figure 4.10B). While the consistent drop in stored value can be partially mitigated through the use of scaling factors elsewhere in the Spaun network, no mechanisms exist for compensating for the faster decay.

Updated Implementation: As part of Spaun 2.0's development, the implementation of the integrator networks was re-examined, as it was hypothesized to be the primary contributor of the reduction in recall accuracy in Spaun's working memory task (see Section 4.6.3.4). By implementing the gated difference integrator network in a network of "rate" neurons (non-spiking neurons), it was observed that the failure of the gated difference integrator network with large input scaling factors was due to induced oscillations in the output of the integrator – which was also fed back to the input of the integrator via the feedback projection (see Figure 4.11).

The origin of these oscillations was traced to the mismatch in the rate of change of the output of the integrator and the rate of change of the input to the integrator (computed as the difference between the input and the output of the integrator), and was exacerbated by the feedback projection. The rate of change of the integrator's output is determined by the synaptic time constant of the feedback projection. Likewise, the rate of change of the integrator's input is determined by the synaptic time constant of the input projection. By using identical synaptic time constants for both projections (see Figure 4.12A), the oscillations in the integrator output



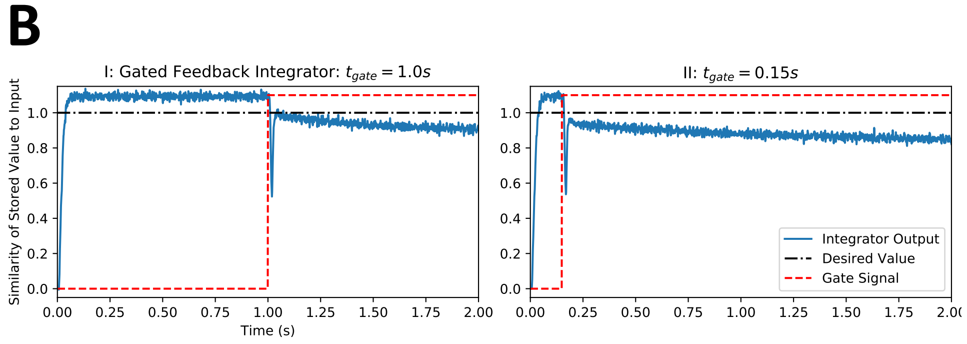


Figure 4.10: Schematic and output graphs for the various configurations of the gated feedback integrator. (**A**) Schematic of the gated feedback integrator. The gated difference integrator three neural populations: the input gate population (*gate*), the feedback population (*fdbk*), and the "integrator" population (*buff*). When the gate signal is "low", the *fdbk* population is inhibited, and the combination of *gate* and *buff* populations function like a communication channel. When the gate signal is "high", the *gate* population is inhibited, and the *fdbk* and *buff* populations function like a "standard" integrator. (**B**) Recorded output graphs of the gated feedback integrator in different operating scenarios. Note that for both graphs, the input to the integrator has been scaled to account for the consistent dip in the output value when the gating signal switches from "low" to "high". (I) Demonstration of the gated feedback integrator operating when the gate signal switch is set at $t = 1.0s$. (II) Demonstration of the gated feedback integrator operating when the gate signal switch is set at $t = 150ms$, showing that unlike the gated difference integrator from Figure 4.9, the gated feedback integrator is able to store the desired input value, at the expense of a more complex circuit, a faster decay in the output value, and the consistent drop in the integrator output when the gate signal switches.

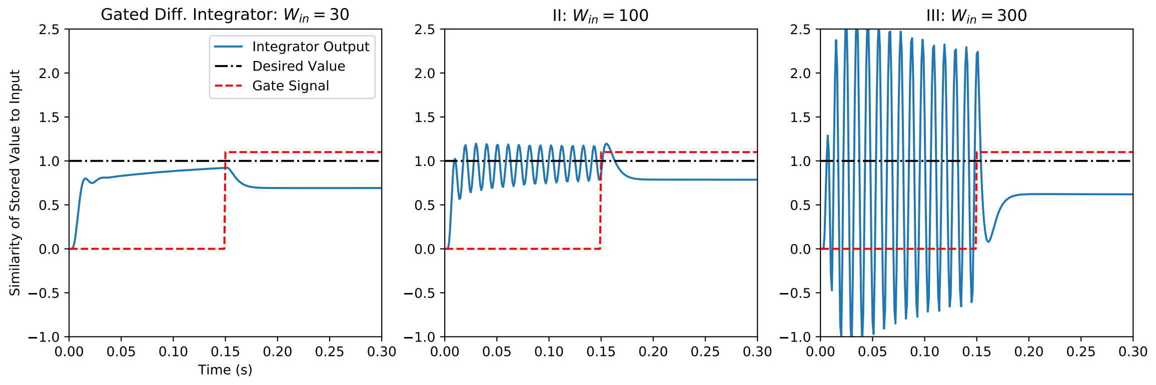


Figure 4.11: Plot of the oscillations observed in a rate neuron version of the gated difference integrator. The effect of the oscillations amplify as the difference scaling factor (W_{in}) is increased.

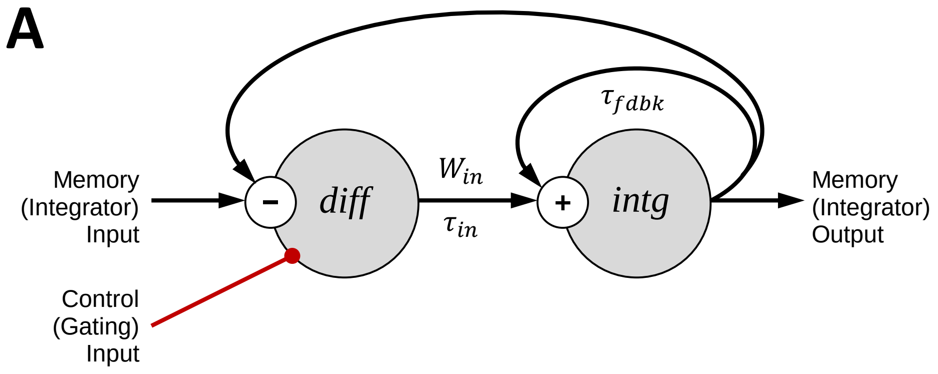
is reduced substantially, allowing the input scaling factor to be increased to a value that satisfies the 150ms stimulus duration requirement (see Figure 4.12B).

In addition to meeting the stimulus duration requirement, the updated gated difference integrator network can be implemented with fewer neurons when compared to the gated feedback integrator network, accounting for the fact that the feedback population (*fdbk*) and the differential gating populations are not required. In addition, the input scaling factor (W_{in}) can be adjusted such that the gated difference integrator can surpass the 150ms stimulus duration requirement, allowing it to successfully store values within a 50ms duration (see Figure 4.12B, graph III). This implies that unlike the memory networks used in Spaun (which are primarily driven by the changes in the visual stimulus), the updated difference integrator network can also be used for memory networks directly controlled by the basal ganglia network (which have been demonstrated to require an average of 50ms to transition between action [Anderson et al., 1995]).

4.3.2 Averaging Memory

Issue Description: The short-comings of Spaun's transform averaging network were discovered through the analysis of Spaun's performance in the fluid induction (progressive matrices) task. It was observed that while Spaun's raw performance accuracy was 70.8% for the sequential digit variant (refer to Figure 3.19A) of the progressive matrices, it was only about 26.3% for the sequential list variant (refer to Figure 3.19B) of the progressive matrices (see Section 3.2.9). Stepping through the transformation calculation for each type of progressive matrix reveals the reason behind the discrepancy in the results.

Recall that for the fluid induction task, the inductive transform $\mathbf{T}$ is computed as the bound



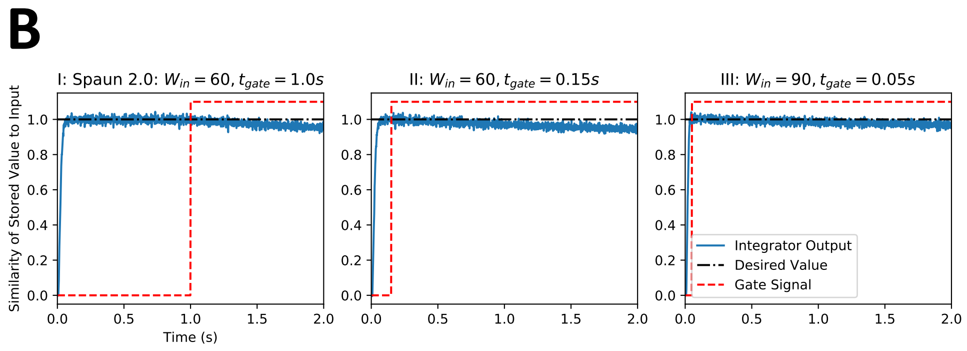


Figure 4.12: Schematic and output graphs for the updated gated difference integrator. (**A**) Schematic of the updated gated difference integrator. The updated network is identical to the network in Figure 4.9, with the exception of the inclusion of a matching synaptic time constant on the input projection to the integrator (τ_{in}). For the Spaun network, $\tau_{in} = \tau_{fdbk}$. (**B**) Recorded output graphs of the updated gated difference integrator in different operating scenarios. (I) Demonstration of the gated feedback integrator operating when the gate signal switch is set at $t = 1.0s$. (II) Demonstration of the updated integrator operating when the gate signal switch is set at $t = 150ms$, showing that unlike the gated difference integrator from Figure 4.9, the updated difference integrator is able to store the desired input value. Additionally, unlike the gated feedback integrator from Figure 4.10, the updated difference integrator is able to accurately store the input value without any substantial degradation to the value. (III) Demonstration that with an increase of the difference scaling factor ($W_{in} = 90$), the updated difference integrator is able to accurately store values when the gating signal is switched after only 50ms.

result of the SPA representation of two cells in the progressive matrix (refer to Eq. (3.9)). The result of the transformation calculation is then averaged (refer to Eq. (3.17)) and applied to the penultimate cell to compute the Spaun's SPA representation of the final cell.

Using the progressive matrix from Figure 3.19A combined with Spaun's list representation (see Eq. (3.13)) and Spaun's digit representation (see Eq. (3.14)) as an example, the transform matrix $\mathbf{T}$ between the first two cells of the sequential digit variant of the progressive matrix can be computed as follows:

$$\begin{aligned}
\mathbf{T} &= \mathbf{CELL}_2 \circledast \sim\mathbf{CELL}_1 \\
&= (\mathbf{POS1} \circledast \mathbf{TWO}) \circledast \sim(\mathbf{POS1} \circledast \mathbf{ZERO}) \\
&= \mathbf{POS1} \circledast \mathbf{TWO} \circledast \sim\mathbf{POS1} \circledast \sim\mathbf{ZERO} \\
&= (\mathbf{POS1} \circledast \sim\mathbf{POS1}) \circledast (\mathbf{TWO} \circledast \sim\mathbf{ZERO}) \\
&= \mathbf{I} \circledast (\mathbf{ZERO} \circledast \mathbf{ADD1}^2 \circledast \sim\mathbf{ZERO}) \\
&= \mathbf{I} \circledast \mathbf{I} \circledast \mathbf{ADD1}^2 \\
&= \mathbf{ADD1}^2
\end{aligned}$$

It can be shown that by performing the identical computation for the remaining cell pairs of this particular sequential digit progressive matrix, the transform matrix $\mathbf{T}$ results in $\mathbf{ADD1}^2$ for each computation. Thus, the averaged transform matrix $\mathbf{T}_{ave}$ should also have a value of $\mathbf{ADD1}^2$. Applying the averaged transform to the penultimate cell produces the result $\mathbf{POS1} \circledast \mathbf{EIGHT}$, which is the desired answer:

$$\begin{aligned}
\mathbf{CELL}_9 &= \mathbf{T}_{ave} \circledast \mathbf{CELL}_8 \\
&= \mathbf{ADD1}^2 \circledast \mathbf{POS1} \circledast \mathbf{SIX} \\
&= \mathbf{POS1} \circledast (\mathbf{SIX} \circledast \mathbf{ADD1}^2) \\
&= \mathbf{POS1} \circledast \mathbf{EIGHT}
\end{aligned}$$

Likewise, the transform matrix computation can be performed for the list sequence variant of the progressive matrix. Here, the matrix illustrated in Figure 3.19B is used as an example. The transform matrix between the first two cells can be computed as (it is important to note that the definition of Spaun's list item position – see Eq. (3.12) – is used in this computation):

$$\begin{aligned}\mathbf{T}_{1,2} &= (\mathbf{POS1} \circledast \mathbf{THREE} + \mathbf{POS2} \circledast \mathbf{THREE}) \circledast \sim(\mathbf{POS1} \circledast \mathbf{THREE}) \\ &= (\mathbf{POS1} \circledast \mathbf{THREE} \circledast \sim(\mathbf{POS1} \circledast \mathbf{THREE})) + (\mathbf{POS2} \circledast \mathbf{THREE} \circledast \sim(\mathbf{POS1} \circledast \mathbf{THREE})) \\ &= \mathbf{I} + (\mathbf{POS2} \circledast \sim\mathbf{POS1} \circledast \mathbf{THREE} \circledast \sim\mathbf{THREE}) \\ &= \mathbf{I} + (\mathbf{INC} \circledast \mathbf{I}) \\ &= \mathbf{I} + \mathbf{INC}\end{aligned}$$

Performing the transform calculation for the second and third cell generates a similar result:

$$\begin{aligned}\mathbf{T}_{2,3} = &(\mathbf{POS1} \circledast \mathbf{THREE} + \mathbf{POS2} \circledast \mathbf{THREE} + \mathbf{POS3} \circledast \mathbf{THREE}) \circledast \\ &\sim(\mathbf{POS1} \circledast \mathbf{THREE} + \mathbf{POS2} \circledast \mathbf{THREE}) \\ = &\mathbf{I} + \mathbf{INC} + \mathbf{INC}^2 + \sim\mathbf{INC} + \mathbf{I} + \mathbf{INC} \\ = &2 \times \mathbf{I} + 2 \times \mathbf{INC} + \mathbf{INC}^2 + \sim\mathbf{INC}\end{aligned}$$

Taking the average of both transforms results in the transform representation:

$$\mathbf{T}_{ave} = 1.5 \times \mathbf{I} + 1.5 \times \mathbf{INC} + 0.5 \times \mathbf{INC}^2 + 0.5 \times \sim\mathbf{INC} \tag{4.1}$$

Applying the sequential list averaged transform to the penultimate cell results in:

$$\begin{aligned}\mathbf{CELL}_9 &= \mathbf{CELL}_8 \circledast \mathbf{T}_{ave} \\ &= (\mathbf{POS1} \circledast \mathbf{SEVEN} + \mathbf{POS2} \circledast \mathbf{SEVEN}) \circledast \\ &\quad (1.5 \times \mathbf{I} + 1.5 \times \mathbf{INC} + 0.5 \times \mathbf{INC}^2 + 0.5 \times \sim\mathbf{INC}) \\ &= 2 \times \mathbf{POS1} \circledast \mathbf{SEVEN} + 3 \times \mathbf{POS2} \circledast \mathbf{SEVEN} + 2 \times \mathbf{POS3} \circledast \mathbf{SEVEN} + \\ &\quad 0.5 \times \mathbf{POS4} \circledast \mathbf{SEVEN}\end{aligned}$$

Because neural ensembles (ensemble arrays) are used to represent the semantic pointer information within Spaun, the saturation of the neurons produces an effect similar to vector normalization (see Section 2.5.7.1). Because of this effect, the smaller vector components (relative to the strengths of the other vector components) in the $\mathbf{CELL}_9$ representation are ignored during Spaun's digit recall phase, resulting in a representation approximating $\mathbf{POS1} \circledast \mathbf{SEVEN} + \mathbf{POS2} \circledast \mathbf{SEVEN} + \mathbf{POS3} \circledast \mathbf{SEVEN}$.

The use of the ensemble arrays to represent the transform semantic pointer, however, introduces a caveat as to the type of information that can be accurately represented. Because the construction of the ensemble array assumes that the vector components of the semantic pointer (to be represented) are normally distributed, the radii of each sub-ensemble in the ensemble array is configured to be $3.5/\sqrt{d}$, where d is the dimensionality of the semantic pointer (see Section 2.18). However, because the transform representation of the sequential list matrix contains the identity vector (which violates the normal distribution assumption), the ensemble arrays used in the original Spaun model could not accurately represent the transform representation, resulting in a decrease in the performance accuracy for the sequential list variant of the fluid induction task.

To elaborate, because each sub-ensemble in the ensemble array had a radius of $3.5/\sqrt{d}$, the maximal value an individual element in the semantic pointer could take is approximately $4.5/\sqrt{d}$ (because of the effect of neural saturation demonstrated in Figure 2.5.7.1). Because all of the information in an identity semantic pointer is contained only within the first vector element (see Section 2.3.4.1), the consequence of the neural saturation is that identity pointers can only be maximally represented as $(4.5/\sqrt{d}) \times \mathbf{I}$. For the 512-dimensional semantic pointers used in the Spaun model, this is approximately $0.2 \times \mathbf{I}$.

Applying the saturation effect to the transform representation computed in Eq. (4.1) results in:

$$\mathbf{T}'_{ave} = 0.2 \times \mathbf{I} + 1.5 \times \mathbf{INC} + 0.5 \times \mathbf{INC}^2 + 0.5 \times {\sim}\mathbf{INC}$$

Propagating this change to the computation of the final cell produces the following result:

$$\begin{aligned}
\mathbf{CELL}'_9 &= \mathbf{CELL}_8 \circledast \mathbf{T}'_{ave} \\
&= (\mathbf{POS1} \circledast \mathbf{SEVEN} + \mathbf{POS2} \circledast \mathbf{SEVEN}) \circledast \\
&\quad (0.2 \times \mathbf{I} + 1.5 \times \mathbf{INC} + 0.5 \times \mathbf{INC}^2 + 0.5 \times {\sim}\mathbf{INC}) \\
&= 0.7 \times \mathbf{POS1} \circledast \mathbf{SEVEN} + 1.7 \times \mathbf{POS2} \circledast \mathbf{SEVEN} + 2 \times \mathbf{POS3} \circledast \mathbf{SEVEN} + \\
&\quad 0.5 \times \mathbf{POS4} \circledast \mathbf{SEVEN}
\end{aligned}$$

Compared to the previous result ($\mathbf{CELL}_9$), the first item of the list representation of $\mathbf{CELL}'_9$ is smaller relative to the vector contributions of the remaining items. Additionally, the vector contribution of the extraneous fourth list item is larger relative to the vector contribution of the remaining items. Experimentally, this results in a decrease in Spaun's ability to correctly identify the first item of its response, or erroneously append a fourth item to its response, both of which results in a decrease in the generative accuracy of the sequential list variant of the fluid intelligence task.

Updated Implementation: Because the configuration of the ensembles within the ensemble array is the primary cause of the reduction in performance accuracy, Spaun 2.0's ensemble arrays are configured such that they are able to accurately represent the identity semantic pointer. This

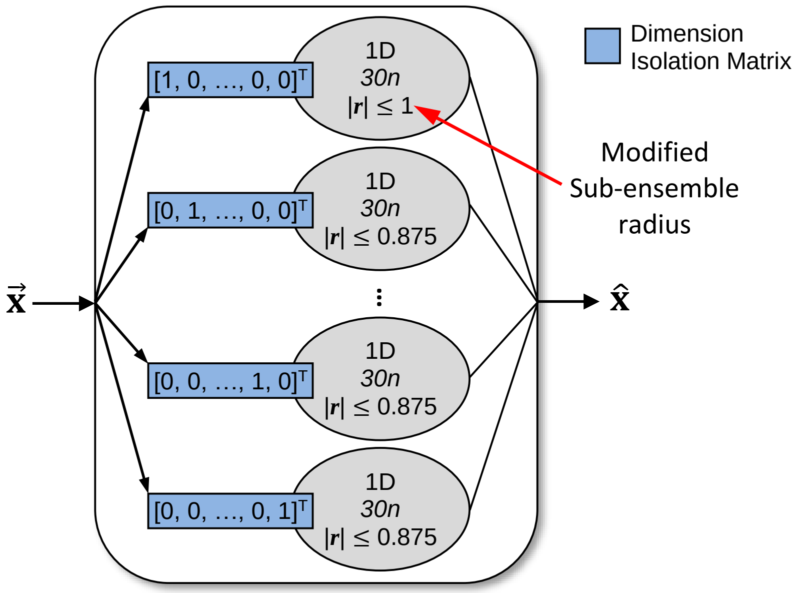


Figure 4.13: Schematic of the updated ensemble array networks used in Spaun 2.0. Shown is a sixteen-dimensional ensemble array comprised of 16 one-dimensional ensembles. Each ensemble in the ensemble array consists of 30 neurons and has a representational range of $\pm 3.5/\sqrt{16}$ ($|r| \leq 0.875$), with the exception of the first ensemble, which has a representational range (radius) of 1 (see accompanying text). Also illustrated are the appropriate transformation matrices (dimensional isolation matrices) necessary to achieve the correct representation for each ensemble.

is made possible by adjusting the radius of the first (and only the first) sub-ensemble in each ensemble array to have a value of 1, instead of $3.5/\sqrt{d}$ (see Figure 4.13).

Setting the radius of the first sub-ensemble to 1 does reduce the ability for the sub-ensemble to represent the first vector component (because of the loss of representational resolution). However, for semantic pointers of sufficiently large dimensionality, the impact on the overall representation of the semantic pointer is negligible.

In addition to the changes to the ensemble arrays, Spaun 2.0 also slightly modifies the computation of the running-average transform. The formulation:

$$\mathbf{T}_{ave} = \begin{cases} \mathbf{T}_N & \text{if } \|\mathbf{T}_{ave}\| = 0 \\ \alpha \times \mathbf{T}_N + (1-\alpha) \times \mathbf{T}_{ave} & \text{otherwise} \end{cases} \tag{4.2}$$

is used instead of the running average formulation described by Eq. (3.17). The updated for-

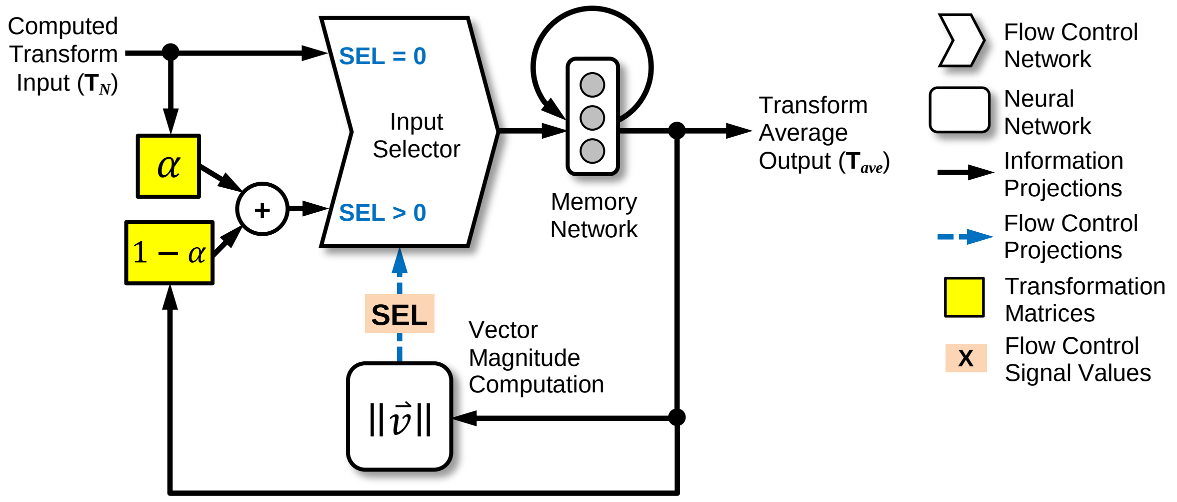


Figure 4.14: Schematic of the updated transform averaging memory network used in Spaun 2.0, modified from the original Spaun averaging memory network to account for the initial condition in Eq. 4.2.

mulation accounts for the initial case where the value of $\mathbf{T}_{ave}$ starts at $\emptyset$, and more accurately approximates the true average.

In the neural implementation, the updated running-average computation is achieved by using a selector, and an ensemble that computes the (vector) magnitude of the information stored in the averaged transform memory (see Figure 4.14). The result of the vector magnitude computation is used to determine the configuration of the selector such that when no information is stored within the averaging memory (i.e., when the vector magnitude is 0), the selector routes the unscaled version of the input vector into the memory network. If the memory network contains a semantic pointer representation, the selector network routes the appropriately scaled version of the input vector (along with the scaled contribution of the averaged transform) into the memory network.

4.4 Transformation System

The transformation system has been significantly changed between the original Spaun and Spaun 2.0 models. However, most of the changes involve the reorganization of the sub-networks and neural projections to improve the re-use of the functional components (e.g., the SPA binding network) across Spaun's different tasks. Figure 3.26 illustrates the result of the functional re-organization, with the major difference being that Spaun 2.0's transformation system requires only one SPA binding network compared to Spaun's transformation system, which required two SPA binding networks.

Apart from the functional re-organization, one significant functional addition has been made to the Spaun 2.0 transformation system to improve Spaun's performance accuracy for the induction tasks, and Spaun's performance reliability while executing the counting task.

Issue Description: During the process of re-implementing the Spaun model, it was discovered that the lack of a robust normalization function partially contributed to the reduction in performance accuracy of the induction tasks (both the rapid variable creation task and the fluid induction task), and a reduction in the behavioural consistency of the counting task (i.e., maintaining the internal dynamics that govern the sub-vocal counting process).

Typically, neural SPA systems are not required to accommodate a semantic pointer normalization feature, since the majority of SPA operations do not modify the magnitude of the semantic pointers.[38] However, the internal dynamics of Spaun's memory system results in Spaun breaking with this convention.

Spaun's memory system (see Section 3.1.3) contains a sub-component that actively decays stored information over time, resulting in stored semantic pointers that, over time, approach $\mathbf{\emptyset}$. The memory system also contains a sub-component that applies a multiplicative increase to the stored semantic pointer representation for each item presented. This results in stored semantic pointer representations that increase in magnitude over multiple stimulus presentations. Further complicating the behavioural dynamics, the output of each sub-component is added to generate the output of Spaun's memory system, which (anecdotally) results in output semantic pointer representations that have magnitudes with an approximate range of 0.7 to 2.5.

Performing an example induction transform computation demonstrates the effect non-unitary magnitudes can have on the output accuracy of Spaun's induction tasks. Using the progressive matrix list sequence variant example from Section 4.3.2, the transform $\mathbf{T}$ can be computed for semantic pointer representations with non-unity magnitudes. In the following example, the semantic pointer representations for each cell in the progressive matrix has been doubled in magnitude. The transform representation $\mathbf{T}_{1,2}$ for the first two cells in the progressive matrix is:

$$\begin{aligned}\mathbf{T}_{1,2} &= 2 \times (\mathbf{POS1} \circledast \mathbf{THREE} + \mathbf{POS2} \circledast \mathbf{THREE}) \circledast {\sim}2 \times (\mathbf{POS1} \circledast \mathbf{THREE}) \\ &= 4 \times ((\mathbf{POS1} \circledast \mathbf{THREE} + \mathbf{POS2} \circledast \mathbf{THREE}) \circledast {\sim}(\mathbf{POS1} \circledast \mathbf{THREE})) \\ &= 4 \times (\mathbf{I} + \mathbf{INC})\end{aligned}$$

[38] An exception is the SPA collection operator, which does modify the semantic pointer magnitude. Assuming the semantic pointer representations are roughly orthonormal, the SPA collection operator will increase the magnitude of the output representation by approximately $/sqrtN$, where N is the number of semantic pointers in the collection.

Likewise, performing the same computation for the next two cells in the progressive matrix reveals that the transform $\mathbf{T}_{2,3}$ for the second and third cell is:

$$\begin{aligned}\mathbf{T}_{2,3} &= 4 \times (2 \times \mathbf{I} + 2 \times \mathbf{INC} + \mathbf{INC}^2 + \sim\mathbf{INC}) \\ &= 8 \times \mathbf{I} + 8 \times \mathbf{INC} + 4 \times \mathbf{INC}^2 + 4 \times \sim\mathbf{INC}\end{aligned}$$

And averaging the two transform representations result in:

$$\mathbf{T}_{ave} = 6 \times \mathbf{I} + 6 \times \mathbf{INC} + 2 \times \mathbf{INC}^2 + 2 \times \sim\mathbf{INC}$$

Applying the averaged transform representation to the eighth cell (once again, doubled in magnitude) produces the representation:

$$\begin{aligned}\mathbf{CELL}_9 &= \mathbf{CELL}_8 \circledast \mathbf{T}_{ave} \\ &= 2 \times (\mathbf{POS1} \circledast \mathbf{SEVEN} + \mathbf{POS2} \circledast \mathbf{SEVEN}) \circledast (6 \times \mathbf{I} + 6 \times \mathbf{INC} + 2 \times \mathbf{INC}^2 + 2 \times \sim\mathbf{INC}) \\ &= 16 \times \mathbf{POS1} \circledast \mathbf{SEVEN} + 18 \times \mathbf{POS2} \circledast \mathbf{SEVEN} + 16 \times \mathbf{POS3} \circledast \mathbf{SEVEN} + \\ &\quad 4 \times \mathbf{POS4} \circledast \mathbf{SEVEN}\end{aligned}$$

As the calculations above demonstrate, the non-unity magnitudes multiplicatively propagate through the transform calculations, resulting in an output representation that is 8 times the magnitude of the input semantic pointers. Because Spaun's decoding system (i.e., the associative memories) can only be configured to expect semantic pointer representations of a fixed (and small) range of magnitudes, the increase in vector magnitudes in the induction task output could result in Spaun producing extraneous outputs (in the case of the example above, Spaun would most likely include the "fourth" digit in its response).

Un-normalized semantic pointers can also affect Spaun's behaviour in the counting task. Spaun's counting mechanism uses on the dot produce operator to determine if the stopping condition has been met (as described in Section 3.2.6), typically by checking if the dot product between the **NUM_COUNT** representation and the **CURR_COUNT** representation is above a specified threshold. If the two representations contain semantic pointers that match, but have under-unity magnitudes, the result of the dot product may not exceed the required threshold, thus preventing Spaun from stopping the internal counting process at the appropriate count. Conversely, if the two representations contain semantic pointers that are non-match but have above-unity magnitudes, the dot product operation could produce a result that is above the required threshold, thus causing Spaun to prematurely stop the internal counting process.

It is important to note that the importance of normalizing the semantic pointer representations was known during the construction of the original Spaun model, and some of the issues described above were somewhat mitigated through the use of a pseudo-normalization network. This network (described in further detail below) uses the saturating property of the neural ensembles to approximate the vector normalization function. However, as illustrated in Figure 2.29 and Figure 4.16, these rudimentary networks were limited in their effectiveness.

Updated Implementation: To address the issues outlined in above, the Spaun 2.0 model implements a robust normalization network. Three different implementations of the normalization network were explored, and are briefly discussed in this section.

The first method of normalization is identical to the method used in the original Spaun model. That is to say, the neural saturation effects demonstrated in Figure 2.28 are used to approximate the vector normalization function. This effect is most pronounced in the scalar case, where the decoded output of a neural ensemble saturates at a value of approximately 1.3 for input values exceeding 2 (i.e., the input value has been "normalized" to a value of 1.3). For multi-dimensional neural ensembles, the saturation has the effect of restricting the vector magnitude while keeping the vector's orientation mostly unchanged. The original Spaun model employs a 512-dimensional ensemble array consisting of 8-dimensional sub-ensembles to approximate the desired vector normalization. However, from Figure 2.29, it can be inferred that this method is only effective for vector magnitudes exceeding 3 (and the vectors are normalized to a magnitude of about 3).

Both the second and third method for computing the normalization function consist of two steps. The first step, common to both methods, requires that the magnitude of the semantic pointer be computed. Recall that the magnitude of the d-dimensional semantic pointer $\mathbf{A}$ is computed as follows:

$$\|\mathbf{A}\| = \sqrt{\sum_{k=0}^{d-1} a_k^2}, \tag{4.3}$$

where a_k is the k^{th} vector element of the semantic pointer $\mathbf{A}$. From Eq. (4.3), the vector magnitude can be computed in a neural network by having a set of neural ensembles (an ensemble array) compute the square of each vector element, the results of which are projected (and summed) into a neural ensemble that computes the square root of the summed squares, as illustrated in Figure 4.16.

Following the computation of the vector magnitude, the second method for computing the normalization function involves performing the vector normalization directly (i.e., each element of the input vector is divided by the total vector magnitude). Just as how a 2-dimensional ensemble can be used to compute the product of two numbers (by computing a set of decoders that approximates the multiplication function), a similar method can be used to compute the division function. However, this method cannot be used to compute the division function for the "standard" range of input values (typically -1 to 1) because of the singularity at $x = 0$ (where the function $1/x$ is undefined). However, for the purpose of computing the vector magnitude in the Spaun network, this problem can be mitigated by restricting the range of the dividend to match the expected input vector magnitudes (defined using empirical data as 0.7 to 2.5).

While the method of directly computing the vector normalization function is sufficient for Spaun's transformation network, the method described below produces more accurate results (using neural networks with identical neuron counts) and has the added advantage of including the functionality of being able to disable the normalization computation when desired. Instead of using a divisive process to reduce the magnitude of the input vector, the third normalization method computes the amount of vector "overshoot" (or "undershoot") and subtracts it from the original input vector in order to normalize it. Mathematically, this can be formulated as:

$$\frac{\mathbf{A}}{\|\mathbf{A}\|} = \mathbf{A} - \frac{\|\mathbf{A}\| - 1}{\|\mathbf{A}\|} \times \mathbf{A}$$

Implementing the third vector normalization technique in a neural network consists of four steps. First, the vector magnitude $\|\vec{v}\|$ for the input vector $\vec{v}$ is computed as described above. Second, the amount of "overshoot" is computed using the function $f(m) = (m - 1)/m$, where m is the value of the input vector magnitude. Next, the input vector is multiplied with the "overshoot" magnitude m to produce a scaled version of the input vector $m\vec{v}$. Finally, the scaled vector $m\vec{v}$ is subtracted from the input vector to produce the normalized vector result. Note that in the neural implementation, it is possible to combine the first and second step into one computation using the function $f(s) = (\sqrt{s} - 1)/\sqrt{s}$, where s is the sum of squares computed by the vector magnitude ensemble array. With this simplification, both the direct normalization method and the subtractive normalization method can be implemented using neural networks of the identical sizes. Figure 4.15 compares the neural implementations of both of these networks. Additionally, with this method of vector normalization, it is possible to disable the vector normalization computation (e.g., for computations where vector normalization is not required) by inhibiting the multiplication network. When the multiplication network is inhibited, nothing is subtracted from the input vector, resulting in no change to the input vector.

Choosing the appropriate implementation for Spaun 2.0 involved comparing the computational accuracy of all three normalization networks. Figure 4.16 plots the input vector magnitudes with the output ("normalized") vector magnitudes of all three normalization methods for input vector magnitudes ranging from 0.5 to 3.0. As the figure demonstrates, the overshoot subtraction method for vector normalization produces the most accurate results over the optimized range. This, and the advantage of being able to inhibit the effects of the normalization is the reason why this network is used in the updated Spaun 2.0 model.

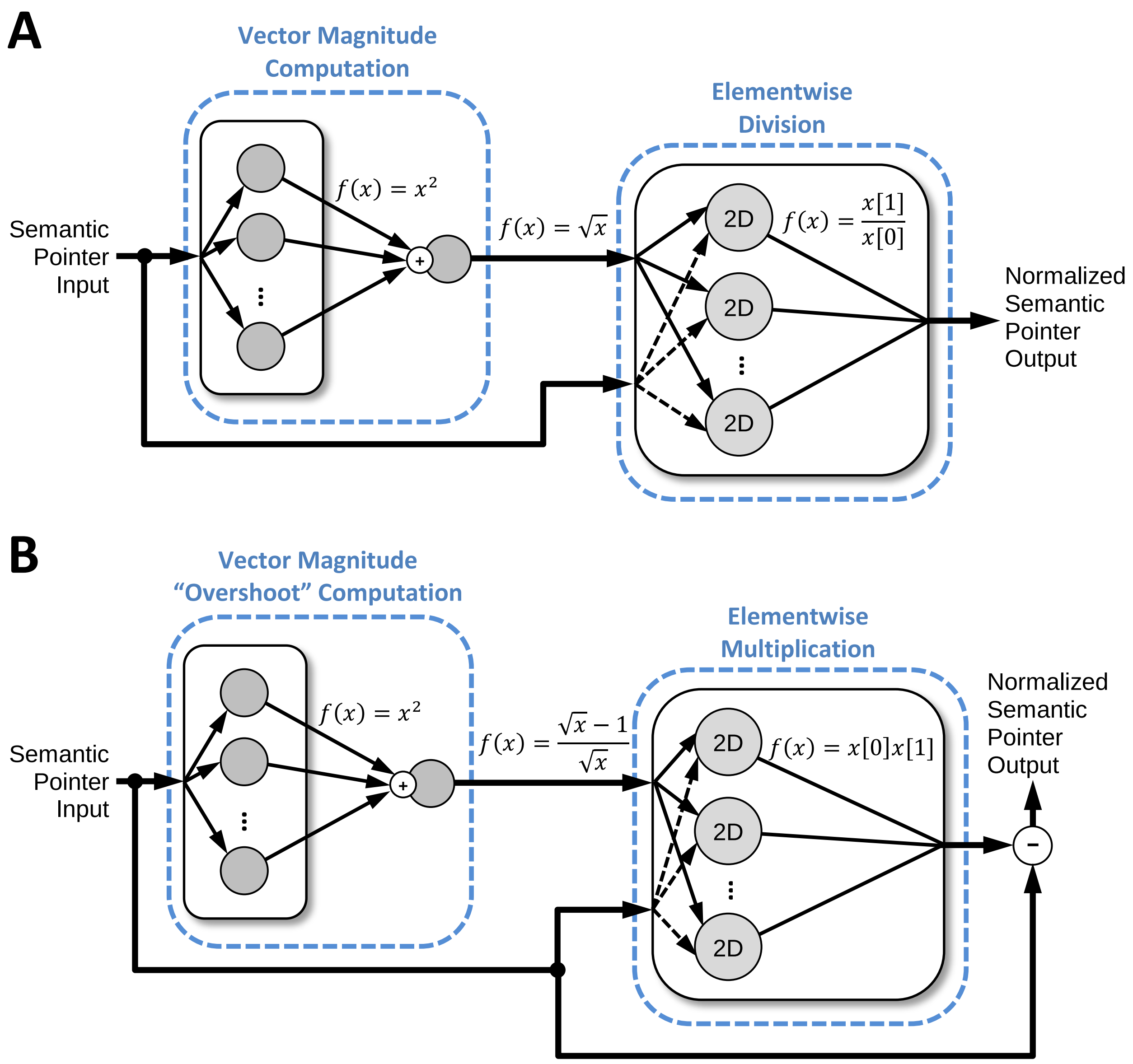

A
Vector Magnitude Computation
$f(x) = x^2$
$f(x) = \sqrt{x}$
Elementwise Division
2D
$f(x) = \frac{x[1]}{x[0]}$
Semantic Pointer Input
Normalized Semantic Pointer Output
B
Vector Magnitude "Overshoot" Computation
$f(x) = x^2$
$f(x) = \frac{\sqrt{x}-1}{\sqrt{x}}$
Elementwise Multiplication
2D
$f(x) = x[0]x[1]$
Semantic Pointer Input
Normalized Semantic Pointer Output

Figure 4.15: Comparison of the network designs for the direct method of vector normalization and overshoot subtraction method of vector normalization. Comparison of the network designs for the direct method of vector normalization and overshoot subtraction method of vector normalization. (**A**) Network schematic for the direct method of vector normalization, whereby the division function is computed in a neural network ("Elementwise Division"). See accompanying text for a description of the limitations of this network. (**B**) Network schematic for the overshoot subtraction method of vector normalization. See accompanying text for a description of the limitations of this network. It is important to note that an additional network is not necessary to perform the subtraction operation as it can be computed at the input of the target network (i.e., the network using the normalized results).

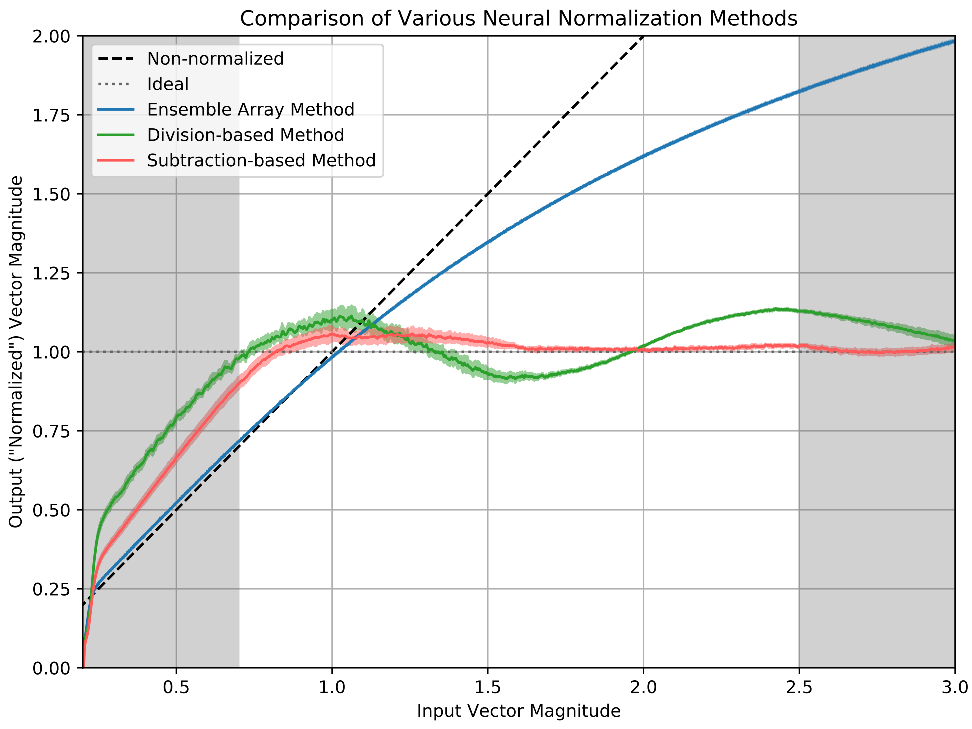


Figure 4.16: Comparison of neural normalization results for the ensemble array normalization network, the division-based (direct) normalization network, and the subtraction-based (overshoot subtraction) normalization network. The data presented is collected over 10 trials testing the normalization efficacy of each network on randomly generated 512-dimensional semantic pointers. The ensemble array network contains 25,600 spiking LIF neurons, while the division and subtraction networks both contain 102,450 neurons. The networks were tested on semantic pointers with vector magnitudes ranging from 0.2 to 3.0, while the networks were optimized for the range of vector magnitudes between 0.7 and 2.5 (non-shaded region). The dashed line indicates the result if no normalization were to occur, while the dotted line indicates the result assuming ideal normalization. Also included is the 95% confidence interval of the results for each network (shaded coloured regions for each plot line). This graph demonstrates that amongst the three networks, the overshoot subtraction network produces the most accurate results for inputs within the optimized range of vector magnitudes.

4.5 Reward Evaluation System

Referring to Section 3.3.7, Spaun's reward evaluation system is relatively simple, compared to Spaun's other functional modules. Recall that the reward evaluation system is responsible for combining Spaun's visual stimuli with Spaun's current actions to generate the error signals required for the adaptive reasoning system. The error signals generated by the reward evaluation system modify the weights[39] projecting from cortex to the basal ganglia, thereby altering the utility value of these conditions. Despite the simplicity of its operation, the Spaun 2.0 model makes one significant change to this system to increase the robustness of Spaun's behaviour during the learning task (the n-arm bandit task).

Issue Description: As part of the process to formalize and re-implement Spaun, it was discovered that the error value computation described in Section 3.2.4 did not impose any upper or lower bounds on the learned basal ganglia utility values. Without strict bounds, given sufficient reward, the utility values can increase to a value large enough to disrupt the proper operation of the basal ganglia network, resulting in a basal ganglia network that chooses the wrong actions, or ceases to function entirely (refer to Section 3.1.4 and Figure 4.17).

While potentially catastrophic, this issue did not arise in the original Spaun model as it was only tested on tasks where rewards were probabilistically assigned (the maximum probability of reward was 72% – see Figure 4.22). In addition, the duration of each phase in the learning task was shortened (from 40 trials to 20 trials) in order to reduce the time required to collect necessary data for the task[40], which meant that Spaun was never presented with number of sequential reward stimuli required to increase the utility values beyond the defined operating range of the basal ganglia network.

Updated Implementation: In Spaun 2.0, enforcing an upper and lower bound on the utility values is achieved by factoring in the current utility values in the error signal calculation. For negative error signals (i.e., Spaun is rewarded for the action taken), the error value is calculated as the difference between 1 and the current utility value of the action. This ensures that the maximum value the utility can take is 1, in which case, the computed error value is 0 (i.e., the action is neither rewarded nor penalized).

[39]Recall that these weights determine the condition of the condition-consequence pairs.

[40]Spaun was run on a compute cluster with a maximum job run time of 7 days. Given that the compute cluster required 2.5 hours to simulate 1 second of the Spaun simulation, the maximum total simulation time was 67 seconds. Since each trial of the learning task required about 1 second to complete, for the 3-arm bandit task in Figure 4.22, a maximum of 20 trials could be run for each "arm".

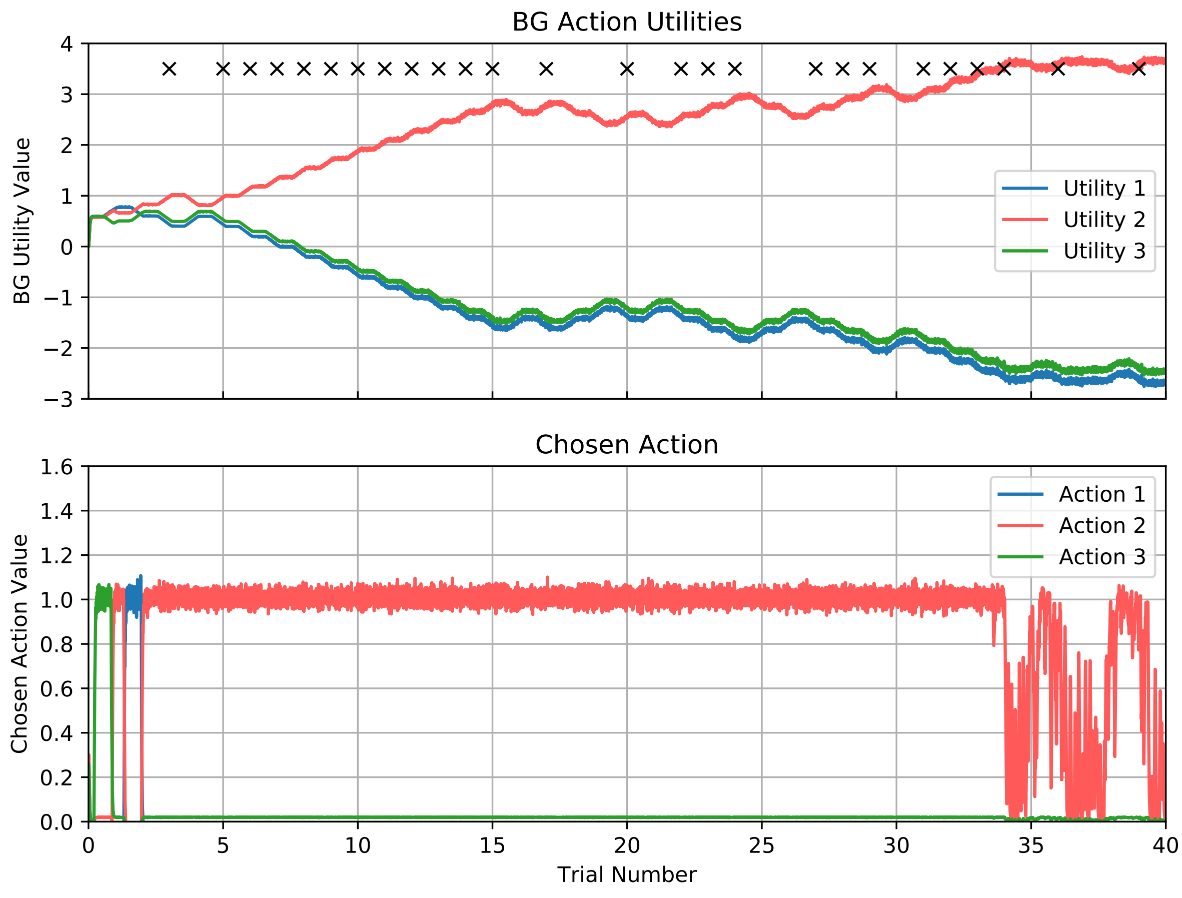


Figure 4.17: Basal ganglia utility value plots using Spaun's (original) reward evaluation system. The graphs show plots of the basal ganglia utilities and Spaun's chosen action for the three-arm bandit task over 40 learning trials. The reward chances for the task are: Action 1 – 12%, Action 2 – 72%, Action 3 – 12%. It is important to note that a simplified Spaun network (containing just the basal ganglia and reward evaluation networks) was used to generate the data for this figure. (Top) Plot of the effect the error signals generated by the reward evaluation system have on the values of the input utilities to the basal ganglia network. Black 'x' marks indicate a trial in which Spaun received a reward. From the graph, it can be seen that the original reward evaluation system allows the basal ganglia utility values to grow beyond the expected range of [0, 1]. (Bottom) Graph of the Spaun's chosen action throughout the learning task, demonstrating that Spaun is able to correctly identify the task with the highest reward (Action 2). However, when the basal ganglia utility values get too high (outside it's operating range), the basal ganglia network starts exhibiting errors in its operation, being unable to consistently indicate a chosen action (trials 34 – 40).

Conversely, for positive error signals (i.e., Spaun not rewarded for the chosen action), the error value is calculated as the difference between 0 and the current utility value of the action. As with the negative error signals, this difference ensures that 0 is the minimal value the utility can have. Table 4.1 illustrates the updated error values for every combination of actions taken and reward stimuli for the 3-arm bandit task.

		Error Values		
Action Chosen	Reward Digit	$\mathbf{E}_1$	$\mathbf{E}_2$	$\mathbf{E}_3$
$\mathbf{A}_1$	0	$\mathbf{U}_1$	$\mathbf{U}_2 - 1$	$\mathbf{U}_3 - 1$
	1	$\mathbf{U}_1 - 1$	$\mathbf{U}_2$	$\mathbf{U}_3$
$\mathbf{A}_2$	0	$\mathbf{U}_1 - 1$	$\mathbf{U}_2$	$\mathbf{U}_3 - 1$
	1	$\mathbf{U}_1$	$\mathbf{U}_2 - 1$	$\mathbf{U}_3$
$\mathbf{A}_3$	0	$\mathbf{U}_1 - 1$	$\mathbf{U}_2 - 1$	$\mathbf{U}_3$
	1	$\mathbf{U}_1$	$\mathbf{U}_2$	$\mathbf{U}_3 - 1$

Table 4.1: Table of the updated Spaun 2.0 mappings between actions chosen, reward stimulus input, basal ganglia utility values, and error values required for Spaun's 3-arm bandit task. If presented with a reward stimulus, the error value is $\mathbf{U}_n - 1$ for the chosen action, and $\mathbf{U}_n$ for the other actions, where $\mathbf{U}_n$ is the basal ganglia utility value for the n^{th} action (i.e., $\mathbf{A}_n$). If presented with a non-reward stimulus, the error is $\mathbf{U}_n$ for the chosen action, and $\mathbf{U}_n - 1$ for the other actions. It is important to note that for the basal ganglia network, the utility values should be bounded by the range $[0, 1]$. Additionally, recall that a positive error value penalizes the action (decreasing its utility value), while a negative error value rewards the action (increasing its utility value).

Unlike the original Spaun model where the intermediary action values (a_n) are given a value 1 if the chosen action is the n^{th} action, and -1 otherwise; Spaun 2.0 assigns "standard" binary values to the intermediary action values – where a value of 1 is assigned to chosen action, and 0 otherwise. An identical assignment is made for the intermediary reward values (r) as well – where a value of 1 is assigned if a reward is presented, and a value of 0 is assigned otherwise. Using this binary system, the value for the error signals can be computed as $\mathbf{E}_n = \mathbf{U}_n - (a_n \times r) - (1 - a_n)(1 - r)$, where $\mathbf{U}_n$ is the basal ganglia utility value of the n^{th} action. The details for the derivation of this formulation can be found in Appendix B.3. Table 4.2 demonstrates how this computation is applied to the three-arm bandit task.

While more complex than the original Spaun implementation, the updated Spaun 2.0 reward evaluation computation offers two advantages. First, because the updated intermediary values are

	a Values					Error Values $(\mathbf{U}_n - (a_n r) - (1-a_n)(1-r))$		
Action Chosen	a_1	a_2	a_3	Reward Digit	r	$\mathbf{E}_1$	$\mathbf{E}_2$	$\mathbf{E}_3$
$\mathbf{A}_1$	1	0	0	0	0	$\mathbf{U}_1$	$\mathbf{U}_2 - 1$	$\mathbf{U}_3 - 1$
	1	0	0	1	1	$\mathbf{U}_1 - 1$	$\mathbf{U}_2$	$\mathbf{U}_3$
$\mathbf{A}_2$	0	1	0	0	0	$\mathbf{U}_1 - 1$	$\mathbf{U}_2$	$\mathbf{U}_3 - 1$
	0	1	0	1	1	$\mathbf{U}_1$	$\mathbf{U}_2 - 1$	$\mathbf{U}_3$
$\mathbf{A}_3$	0	0	1	0	0	$\mathbf{U}_1 - 1$	$\mathbf{U}_2 - 1$	$\mathbf{U}_3$
	0	0	1	1	1	$\mathbf{U}_1$	$\mathbf{U}_2$	$\mathbf{U}_3 - 1$

Table 4.2: Table demonstrating the intermediary values and computation required to produce the error values listed in Table 4.1.

"standard" binary values, the SPA computation needed to generate these values are simplified. Shown below is a comparison of the SPA operations needed to compute the intermediary action value a_n for the original Spaun model (left), and the Spaun 2.0 model (right). Recall that the semantic pointer $\mathsf{SP}_{\mathbf{ACT}}$ is the basal ganglia representation for the chosen action, and the semantic pointer $\mathsf{A}N$ represents each of the possible actions.

$$a_1 = \mathsf{SP}_{\mathbf{ACT}} \bullet (\mathsf{A1} - \mathsf{A2} - \mathsf{A3}) \qquad a_1 = \mathsf{SP}_{\mathbf{ACT}} \bullet \mathsf{A1}$$
$$a_2 = \mathsf{SP}_{\mathbf{ACT}} \bullet (-\mathsf{A1} + \mathsf{A2} - \mathsf{A3}) \qquad a_2 = \mathsf{SP}_{\mathbf{ACT}} \bullet \mathsf{A2}$$
$$a_3 = \mathsf{SP}_{\mathbf{ACT}} \bullet (-\mathsf{A1} - \mathsf{A2} + \mathsf{A3}) \qquad a_3 = \mathsf{SP}_{\mathbf{ACT}} \bullet \mathsf{A3}$$

The SPA computation for the intermediary reward value r has been similarly simplified to:

$$r = \mathsf{SP}_{\mathbf{VIS}} \bullet \mathsf{REWARD}$$

The second advantage to the updated reward evaluation computation involves the neural implementation of the error signal computation. The original Spaun formulation required the use of the multiplication operator to generate the error values. This in turn requires a 2-dimensional neural ensemble to compute the multiplication function. However, because the updated error signal formulation involves a multiplication with 0 (and not a real-valued number, as did the original formulation), the multiplication can be achieved through the use of inhibitory projections. That is to say, if the function $x \times y$ is required, and y can only take on the values 0 and 1, the aforementioned function can be neurally implemented by inhibiting the neural ensemble representing x with the binary inverse of y (i.e., $1-y$). This forgoes the need for the 2-dimensional neural ensemble, reducing the overall neuron count and the complexity of the system.

Figure 4.18 illustrates how this multiplication implementation is combined with the SPA computations to form the updated neural implementation of the reward evaluation network. Because the utility values are included in the error value calculations, the updated network also features an error signal suppression network that inhibits the error signal when neither an action has been taken nor a reward presented.

Figure 4.19 demonstrates the updated reward evaluation network used in a simplified Spaun network, containing just the basal ganglia and reward evaluation networks. As in Figure 4.17, the network was tested using the three-arm bandit task. However, in Figure 4.19, the reward probabilities were updated such that only 1 action (Action 2) has a 100% chance of being rewarded. This was done to demonstrate that even when receiving continuous reward signals, the updated reward evaluation network is able to keep the basal ganglia values bounded to the expected range of [0, 1] so that it functions correctly.

4.6 Comparison of Spaun and Spaun 2.0

While architecturally similar, due to the major changes to several core systems, the Spaun 2.0 model is significantly different to the original Spaun model. This section briefly explores the logistical differences between the two Spaun models (Section 4.6.1), the changes made to the organization of the Spaun code base (Section 4.6.2), and most importantly, a comparison between the task performances of the original Spaun and the updated Spaun 2.0 model (Section 4.6.3).

4.6.1 Logistics

Perhaps the core logistical difference between the original Spaun model and the Spaun 2.0 model is that the Spaun 2.0 model is entirely implemented using spiking LIF neurons, whereas parts of the original Spaun model (e.g. visual hierarchy, motor system) were not. As such, the number of neurons used to implement the model has increased from approximately 2.34 million neurons (Spaun) to approximately 4.58 million neurons (Spaun 2.0). Table 4.3 provides a more detailed comparison of the neuron counts of the major systems in the Spaun and Spaun 2.0 models.

As Table 4.3 demonstrates, most of Spaun's modules have increased in neuron count between the Spaun and Spaun 2.0 implementations. The following list briefly summarizes the modifications in each module that can be attributed to the change in neuron count.

- Vision Module: While the implementation of a fully spiking visual hierarchy contributed to the change in neuron count for this module, the primary reason for the increase in neuron count is the reorganization of the visual working memory as part of the vision system (the

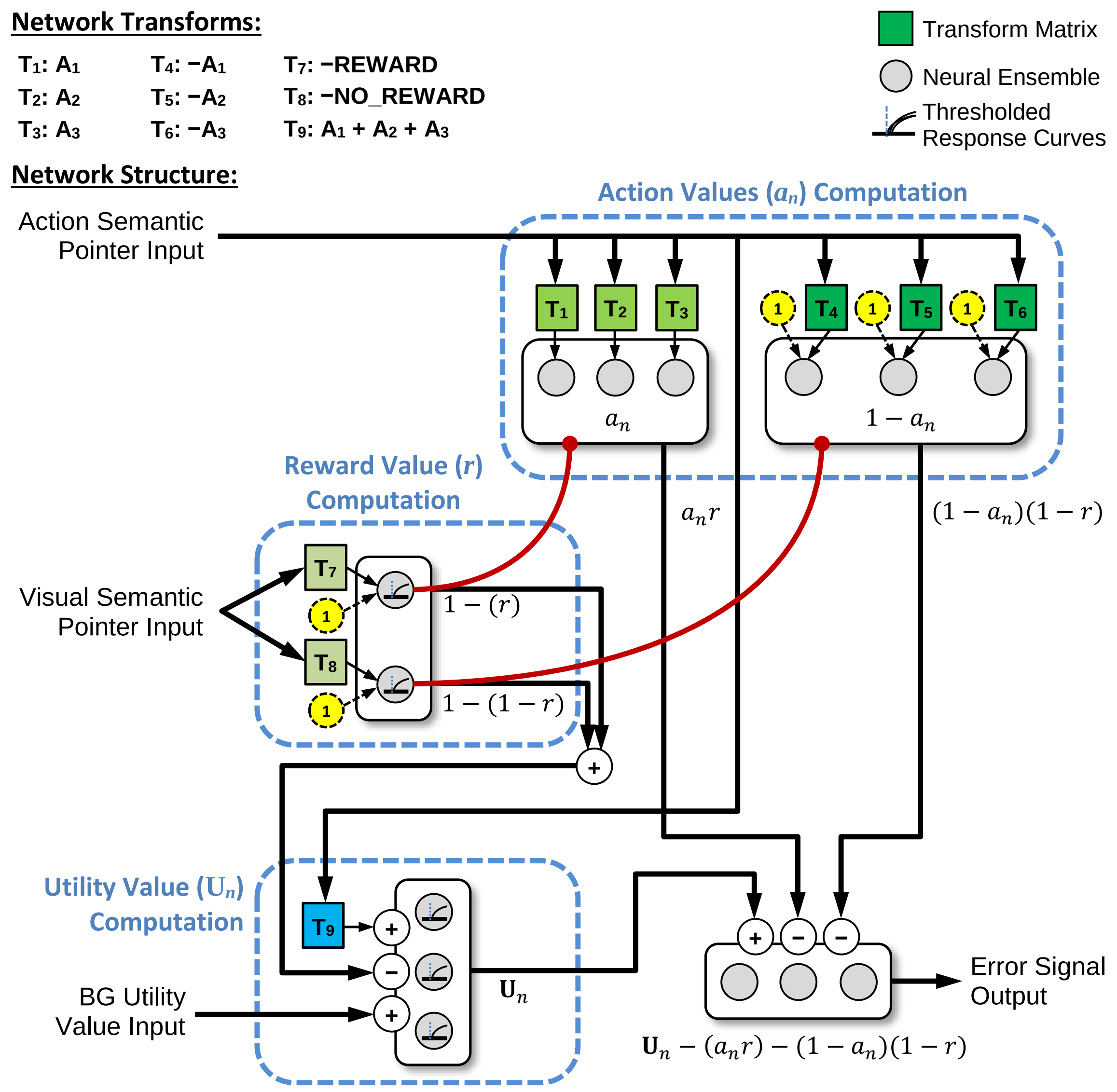


Figure 4.18: Network schematic of Spaun 2.0's updated reward evaluation network, illustrating the various networks and transformation matrices used to compute the intermediary action and reward values necessary to generate the error signal value. Note the use of inhibitory connections to implement the multiplication operator. Also note that the utility value computation network has additional input projections to suppress its output when no reward is provided to Spaun. Refer to the accompanying text for details.

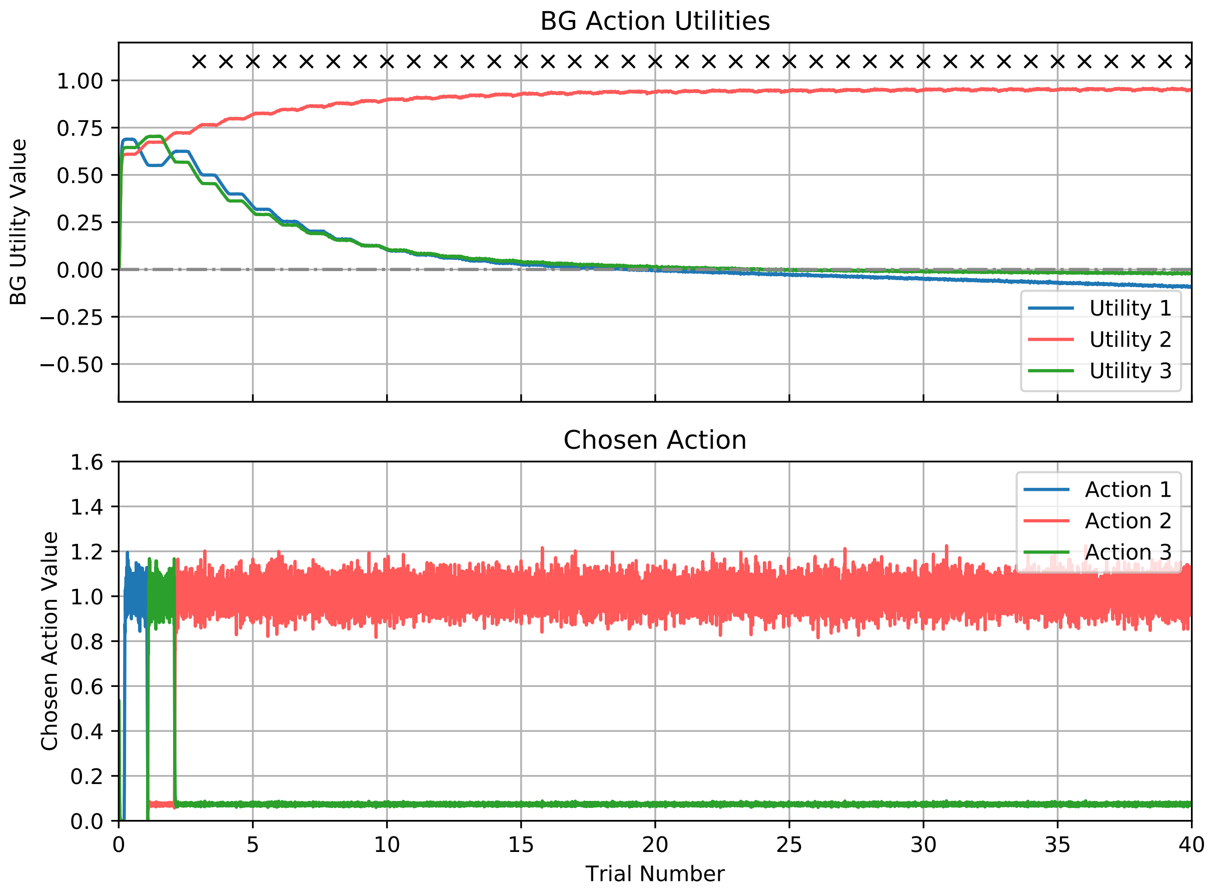


Figure 4.19: Basal ganglia utility value plots using Spaun 2.0's updated reward evaluation system. The graphs show plots of the basal ganglia utilities and Spaun's chosen action for the three-arm bandit task over 40 learning trials. The reward chances for the task are: Action 1 – 0%, Action 2 – 100%, Action 3 – 0%. (Top) Plot of the effect the error signals generated by the reward evaluation system have on the values of the input utilities to the basal ganglia network. Black 'x' marks indicate a trial in which Spaun received a reward. From the graph, it can be seen that despite receiving continual rewards, the utility values remain in the range [0, 1]. (Bottom) Graph of the Spaun's chosen action throughout the learning task, demonstrating that Spaun is able to correctly (and consistently) identify the task with the highest reward (Action 2).

Module Name	Spaun Model	Spaun 2.0 Model
Vision	52,021	186,400
Motor	7,620	99,850
Information Encoding	437,360	618,620
Information Decoding	155,070	533,820
Working Memory	1,060,620	1,540,950
Transformation	527,850	903,910
Reward Evaluation	710	600
Action Selection Hierarchy (Layer 1)	99,991	692,860
Total neuron count	2,341,242	4,577,010

Table 4.3: Comparison of the neuron counts for each major module in the Spaun and Spaun 2.0 models.

original visual working memory network was part of the general working memory module), and the modification of the change detect network from being a semantic pointer based network to a pixel based network.

- Motor Module: The implementation of the fully spiking motor hierarchy, and the fully spiking motor timing controller contributed heavily to the increase in neuron count for this module.
- Information Encoding Module: The architecture of this module remained unchanged, but the position working memory network was made more robust to longer list lengths. Additionally, networks were implemented to provide functionality for serial recall in the reverse direction.
- Information Decoding Module: While not discussed in the previous sections (as the changes did not specifically address a critical issue with the Spaun model), the information decoding module has been re-implemented to be more robust during the decoding process. This included a hierarchical cleanup memory to better discern between "known" outputs (Spaun is confident about the output value) and "unknown" outputs (Spaun is confident there is an output digit, but is unsure of the value of the digit).
- Working Memory Module: Architecturally identical to the original working memory, with the exception that the counting circuitry has been implemented within each integrator of the working memory module. This allows all memories to be incremented simultaneously, rather Spaun needing to route the information to the transformation system to do so.
- Transformation Module: The transformation module received additional selector networks to increase its general-use capability.

- Reward Evaluation Module: This is the only module that decreased in neuron count. This is due to the simplification of the multiplication function used in the error signal computation (see Section 4.5).
- Action Selection Hierarchy: To reduce the neuron count of the original (Spaun) action selection hierarchy, it used axis-aligned semantic pointers, rather than randomly generated semantic pointers. This reduced the dimensionality (and thus the size of the task memory networks) of the semantic pointers to 10. In Spaun 2.0, the task memory networks in the action selection hierarchy use the full 512-dimensional semantic pointers, thus increasing the neuron count for this module.

The increase in the number of neurons between the Spaun and Spaun 2.0 models has undoubtedly increased the simulation time for each model. To collect the data published in [Eliasmith et al., 2012], the original Spaun model was run on a compute cluster with 4-core, 2.4Ghz AMD Opteron processors, requiring about 2.5 hours to simulate 1 second of Spaun's simulation. Using more up-to-date hardware (6-core, 3.5Ghz Intel® Xeon® E5-1650v3 processor), the original Spaun model requires approximately 50 minutes per second of simulation time. In contrast, the Spaun 2.0 model requires approximately 2 hours per second of simulation time on the newer hardware. Fortunately, the time required to run the Spaun simulations can be reduced by leveraging the massively parallel processing architecture of modern GPUs. Using the `nengo_ocl` (Nengo simulator implemented with in OpenCL™) software package, and the Nvidia® Titan X graphics card, the Spaun 2.0 model required only 30s to process 1s of the simulation.

4.6.2 Code base

One of the primary objectives of the re-implementation of Spaun is the reorganization of the Spaun code base. As illustrated in Figure 4.20, the majority of Spaun's code was contained in one monolithic script file (`spaun_main.py`). Spaun 2.0 improves the organization of the code base by dividing each module's code into a separate file, and groups them in their own folder. Additionally, the SPA-specific network scripts, and the generic network scripts have been relocated in their respective folders. The changes to the code base make the Spaun 2.0 model easier to maintain and build expansions for, thereby working towards the goal of realizing Spaun as a test bed for experimental neural algorithms.

4.6.3 Results

Compared to the original Spaun model, the Spaun 2.0 model contains several substantial changes in each of its modules, most of which have been made to address concerns regarding the original

A

```
root
 │ │
 │ ├── /motor_data ...  Data files
 │ │                    for the
 │ │                    motor system
 │ └── /vis_data ...  Data files
 │                    for the vision
 │                    system
 ├── spaun_main.py ...  Main spaun
 │                      script,
 │                      contains code
 │                      for each Spaun
 │                      module
 ├── conf.py ...
 │   Configuration
 │   parameters for Spaun
 ├── cleanup_mem.py
 ├──   ⋮
 └── visual_hier.py
```

B

```
root
 ├── /_networks ...
 │   Generic
 │   networks (e.g.
 │   normalization
 │   network)
 ├── /_spa ...  SPA-specific networks
 │              (e.g.  associative
 │              memory)
 ├── /modules ...  Folder for module
 │   │             scripts and data
 │   │             files
 │   ├── info_decoding.py
 │   ├──   ⋮
 │   └── working_memory.py
 ├── spaun_main.py ...  Main spaun
 │                      network.
 │                      Imports module
 │                      specific code
 └── configurator.py ...  Configuration
                          parameters
                          for Spaun
```

Figure 4.20: Comparison of the organization of the Spaun and Spaun 2.0 code base. Note that only the major files and folders are included in each directory tree. (**A**) Directory tree for the Spaun code base. (**B**) Directory tree for the Spaun 2.0 code base. The full Spaun 2.0 code can be found at `https://github.com/xchoo/spaun2.0`.

Spaun model. To demonstrate the effects these changes have made to the behaviour of the updated Spaun model, it was run against the same battery of tasks (refer to Section 3.2) as the original Spaun model. This section compares and contrasts the results from the original and improved Spaun models.

4.6.3.1 Copy Drawing

The Spaun 2.0 model uses a completely different visual hierarchy than the original Spaun model. Thus, the results of the copy drawing task demonstrate that the visual semantic pointer generated by the updated visual hierarchy retains sufficient feature information for the methods described in Section 3.2.2 to use in generating the motor semantic pointer.

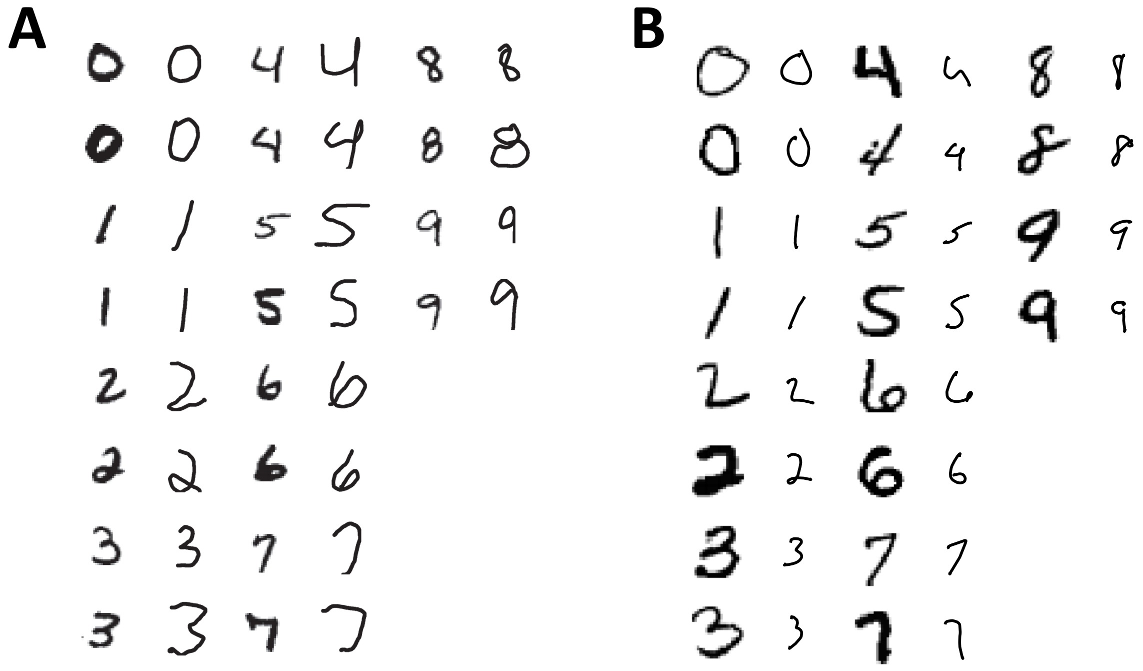


Figure 4.21: Comparison of Spaun and Spaun 2.0 results for the copy drawing task. For each pair of digits, the MNIST stimulus image used is displayed on the left, and Spaun's reproduction is displayed on the right. Shown are two randomly selected examples for each digit from 0 to 9. (**A**) Copy drawing task results for the Spaun model. (Adapted from [Eliasmith et al., 2012] with permission) (**B**) Copy drawing task results for the Spaun 2.0 model.

As Figure 4.21 demonstrates, the semantic pointers generated by the updated hierarchy contain enough semantic information to be successfully used for this task, enabling Spaun 2.0 to (as with the original Spaun model) capture distinctive features (e.g. the slant of the digit, the appearance of loops, etc.) of the stimulus images. Additionally, the results demonstrate that using the updated (fully-spiking) motor arm controller in Spaun 2.0 results in "cleaner" (less ragged) digits compared to the original Spaun model.

4.6.3.2 Digit Recognition

For the purpose of comparing the original Spaun and Spaun 2.0 models, the digit recognition task serves as a sanity check to ensure that the updated visual system is able to operate correctly in conjunction with the rest of Spaun's modules.

The original Spaun model reported a digit recognition task accuracy of 94% [Eliasmith et al., 2012], while the Spaun 2.0 model achieves a performance accuracy of 97.76% (95% confidence interval of 96.76% to 98.66%) on the digit recognition task. Compared to the original Spaun model, Spaun 2.0's results more closely match human performance, which is reported to be approximately 98% [Chaaban and Scheessele, 2007].

4.6.3.3 *N*-arm Bandit Task

As one of the major changes in the Spaun 2.0 model involves the modification of the error signal computations, the *N*-arm bandit task serves to demonstrate that the updated Spaun 2.0 model is able to reproduce the results demonstrated by Spaun. Additionally, the bandit task can be run for extended trial counts to demonstrate that Spaun 2.0 is capable of performing the longer learning tasks without suffering from any unintended disruption in Spaun's behaviour.

Figure 4.22 compares the results of the bandit task for both the Spaun and Spaun 2.0 models. Both models were configured with similar learning rates[41] and run on a 3-arm bandit task configured with identical reward probabilities and number of trials (see figure caption for details). As the figure illustrates, for each set of 20 trials, both the Spaun and Spaun 2.0 models were able to correctly identify the action with the highest probability of reward.

In order to demonstrate that the modified reward evaluation system (combined with the action selection system) is capable of continual performance for prolonged learning trials, the Spaun 2.0 model was also run on the 2-arm bandit task for a total of 160 learning trials (compared to 60 in the Figure 4.22). The 2-arm bandit task consisted of 4 sets of 40 trials, with the reward probabilities modified for each set of trials. Figure 4.23 compares the results obtained from the Spaun 2.0 model with the results obtained from the adaptive basal ganglia model from [Stewart et al., 2012].

The results illustrated in Figure 4.23 demonstrate that with the changes to the reward evaluation system, Spaun 2.0 is capable of performing extended learning tasks, whereas the original Spaun model might have failed to do so correctly (see Section 4.5). Of importance to note, for the Spaun 2.0 results, the drop in the probability of choosing "Action 1" for the last couple of trials (trials 158 – 160) is an artifact of how the Spaun simulation is run. The Spaun simulation is run for a predetermined amount of time, which is estimated by combining the number of input stimuli Spaun is to receive with the number of responses Spaun is expected to provide. However,

[41] The neural learning rule implementation differs slightly between the Nengo 1.4 and Nengo 2.0 simulation software. The learning rate of Spaun 2.0 was configured to best match the effects of learning in both versions of the Nengo software.

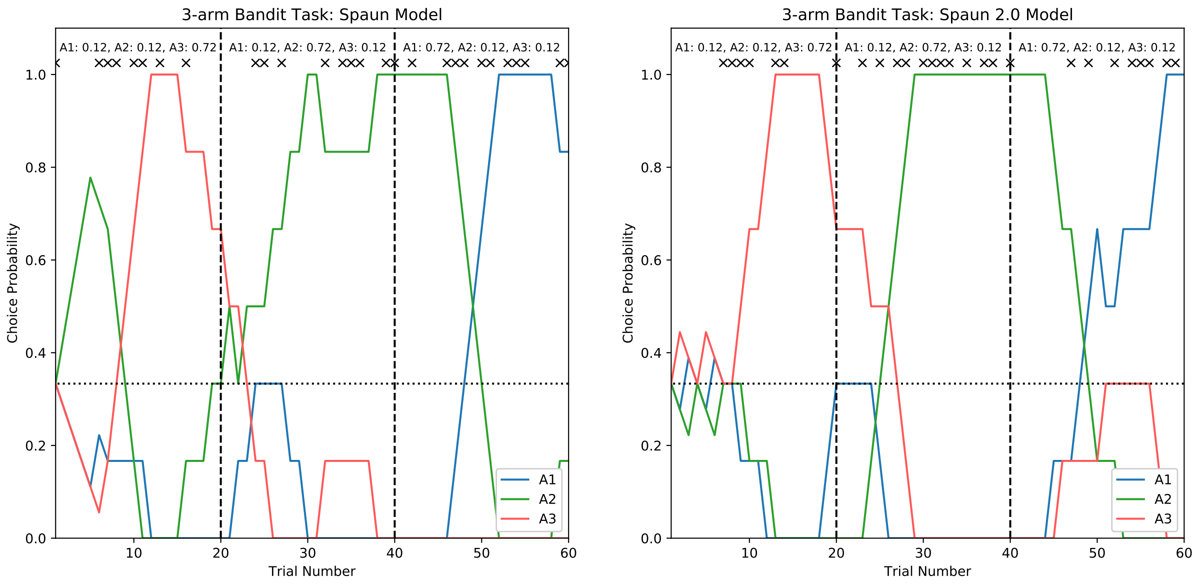
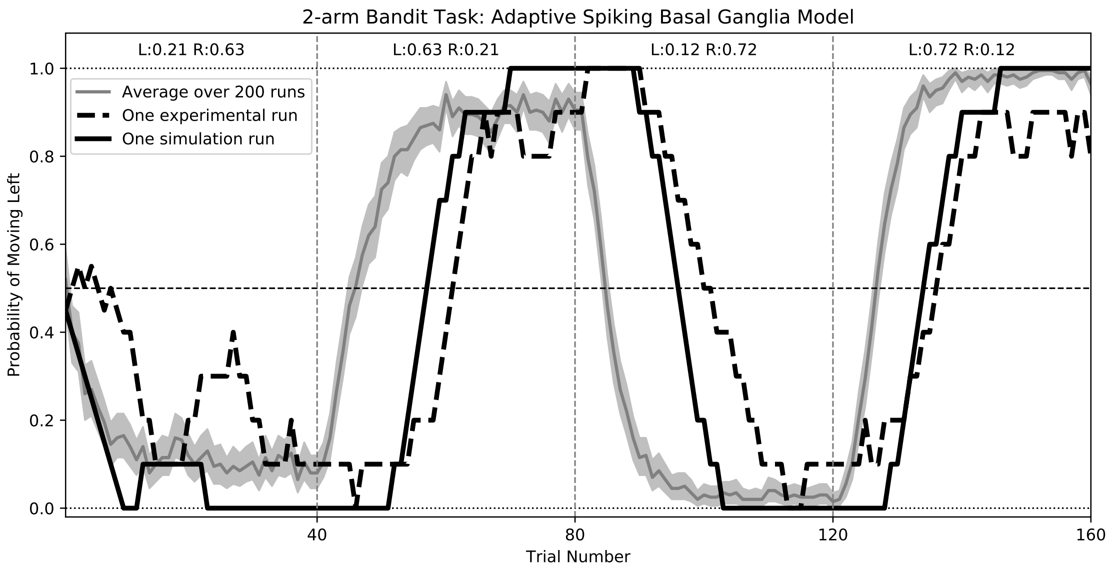


Figure 4.22: Comparison of Spaun and Spaun 2.0 results for the 3-arm bandit task. For this specific task, the reward chance for action 1 (A1) is 12% for the first 20 trials, 12% for the next 20 trials, and 72% for the last 20 trials. Likewise, the reward chance for the second action (A2) is 12%, then 72%, and lastly 12%. For the third action (A3), the reward chances are 72%, then 12%, and finally 12%. Rewarded trials are marked with a 'x'. Spaun's probability of choosing a particular action is computed over a 5-trial window. (Left) 3-arm bandit task results for the Spaun model. (Adapted from [Eliasmith et al., 2012] with permission) (Right) 3-arm bandit task results for the Spaun 2.0 model. The particular experimental run displayed was chosen out of a batch of 10 randomly instantiated Spaun 2.0 models. Out of the 10 experimental runs, the run shown above was chosen as the pattern of rewarded trials most closely matched the original Spaun data.

because the timing of Spaun's responses is non-deterministic, it is possible that, within the fixed simulation time, Spaun is unable to complete all 160 learning trials. Trials that Spaun is unable to complete are marked as having no action been taken, thus reducing the overall probability of choosing Action 1 for the last couple of learning trials.

4.6.3.4 List Memory

The Spaun 2.0 model significantly altered the design and behaviour of the integrator networks used in the memory system, reducing the decay rate of the integrators, and removing the consistent dip in stored representations when the gating signal is switched (see Section 4.3). As such, it is



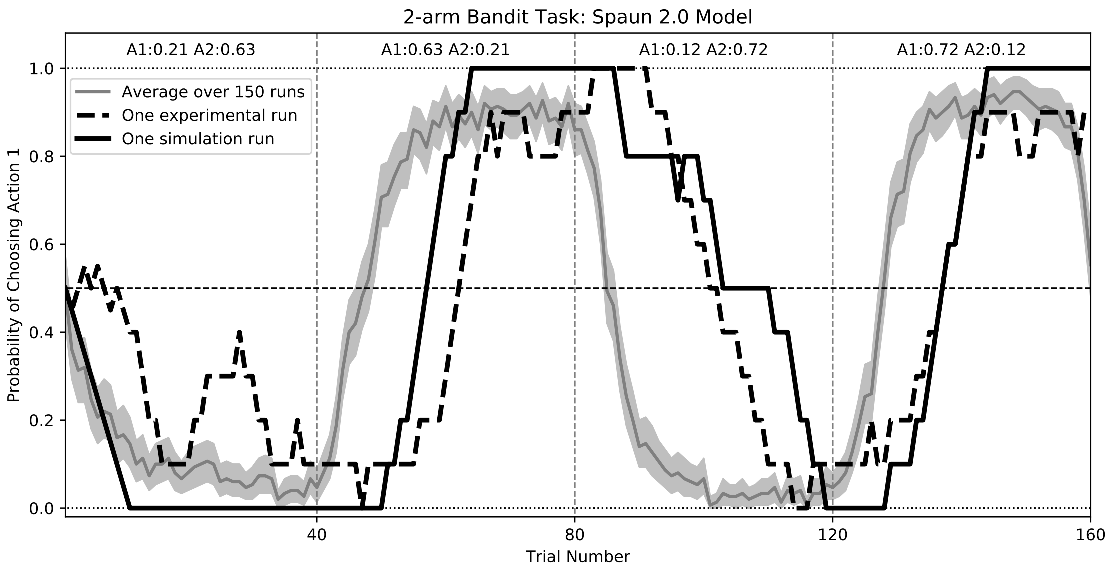


Figure 4.23: Comparison of results between the adaptive basal ganglia model [Stewart et al., 2012] and the Spaun 2.0 model for the 2-arm bandit task. The task consists of 4 sets of 40 learning trials, with the probabilities of rewarded actions listed above each set of trials. Note that the adaptive spiking model uses the terms "Left" and "Right" to denote the actions, while the Spaun model uses the terms "Action 1 (A1)" and "Action 2 (A2)". For each graph, the model's probability of choosing the first action is averaged over all of the runs and plotted in gray, with the shaded region indicating the 95% confidence interval. Also plotted (dashed line) is sample data from [Kim et al., 2009], where rats were trained to perform the 2-arm bandit task. Additionally, each graph plots a single run of the respective models, using an averaging window of 10 trials. (Top) Data obtained from the adaptive spiking basal ganglia model [Stewart et al., 2012]. (Adapted from [Stewart et al., 2012] with permission). (Bottom) Data obtained for the Spaun 2.0 model. Note that the mean choice performance is averaged over 150 runs, compared to the 200 in the previous graph.

expected that these changes will affect Spaun's performance on the list memory task.

Figure 4.24 compares the recall accuracy curves for the serial list memory task for lists 4 to 8 digits in length. Both data from the original Spaun model, and the Spaun 2.0 model are illustrated, as well as human recall accuracy data (for lists of numbers) as reported in [Dosher, 1999]. As the graphs indicate, addressing the shortcomings of Spaun's integrator networks has resulted in an improvement in Spaun's recall accuracy, resulting in data that more closely matches human performance data. Comparing the performance results of the Spaun 2.0 model, and human subjects, it can be seen that the general shape of the recall curves match, however the Spaun 2.0 model is more accurate at recalling longer lists. It is important to note that this may be because the Spaun 2.0 model was configured with the same memory parameters as the original Spaun model (which borrowed its parameters from the OSE memory model), and no effort was made to fit the performance of Spaun 2.0 to the human data. It is also hypothesized that the implementation of a more realistic episodic memory component (see Section 3.1.3) may result in a better match to human data, however, this is left for future work.

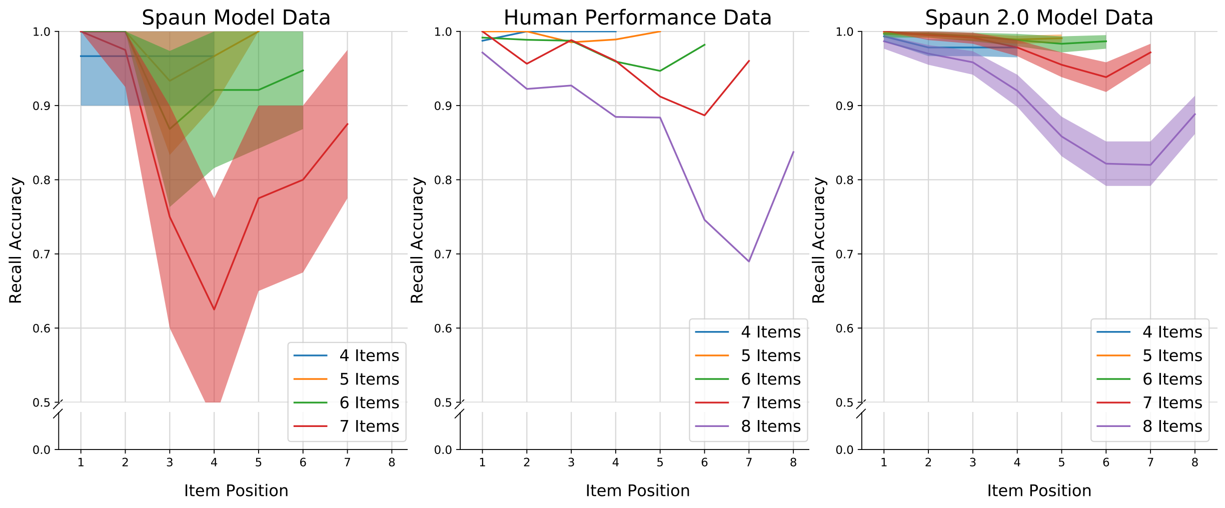


Figure 4.24: Comparison of results between the Spaun model, the Spaun 2.0 model and human performance for the serial list recall task. For both Spaun models, the shaded region indicates the 95% confidence intervals for the data collected. (Left) Results from the original Spaun model. (Adapted from [Eliasmith et al., 2012] with permission). (Center) Human serial recall performance data for digit lists, as reported in [Dosher, 1999] (Adapted from [Eliasmith et al., 2012] with permission). (Right) Results from the Spaun 2.0 model, which included a re-implementation of the core integrator networks within the memory system.

4.6.3.5 Counting

Unlike most of the mechanisms (e.g. inductive reasoning, question answering, etc.) within Spaun – which are primarily driven by the visual system – the core of Spaun's counting mechanism is driven by internal control signals generated within the motor system (see Section 4.2). As such, the updates to the motor timing controller (see Section 4.2.1) will alter Spaun's behaviour during the counting task. Figure 4.25 compares the response generation time for the Spaun model (as reported in [Eliasmith et al., 2012]) and the Spaun 2.0 model. From the graphs, several observations can be made. First, for both the original Spaun model, and the updated Spaun 2.0 model, the task response time increases linearly with respect to the count length. Second, the variability in the response timing increases as the number of counts increases, demonstrating that both models reproduce an effect known as "Weber's Law" [Krueger, 1989].

The differences between the data is also important to note. It is observed that the variance in response times for the Spaun 2.0 model is greater than for the original Spaun model. This is most likely due to the fully neural implementation of the motor timing controller, which introduces more noise (and thus more variance) in the timing of Spaun's responses.

Additionally, the count time per item for the Spaun 2.0 model is greater than for the Spaun model – measured to be 475 ± 63 ms and 419 ± 10 ms [Eliasmith et al., 2012], respectively. Compared to the subvocal count time (per item) for human subjects [Landauer, 1962], Spaun 2.0's results lies on the edge of the measured human range of 344 ± 135 ms. The increase in count time between the Spaun and Spaun 2.0 models can be attributed to the fully neural implementation of the motor timing controller. While the ramp timing (i.e., the time it takes the ramp to go from 0 to 1) remains unchanged between the two models, the Spaun 2.0 model does not reset the ramp instantaneously. Rather, the Spaun 2.0 model relies on the ramp reset circuitry to reset the ramp (see Section 4.2.1), thus adding to the count time for each item.

With regards to the difference between the count times reported for the human subjects and the Spaun models in general, this is likely due to the fact that the Spaun's method of "subvocalization" differs from the human subjects (Spaun "imagines" writing the digit instead). Implementing a proper subvocalization mechanism (i.e., one that is independent of the motor system) would probably result in a better match to human data, however, this is left to future work.

4.6.3.6 Question Answering

The original Spaun model made the prediction that the memory effects observed in the serial list recall task (i.e., the effects of primacy and recency) would also manifest in the recall accuracies

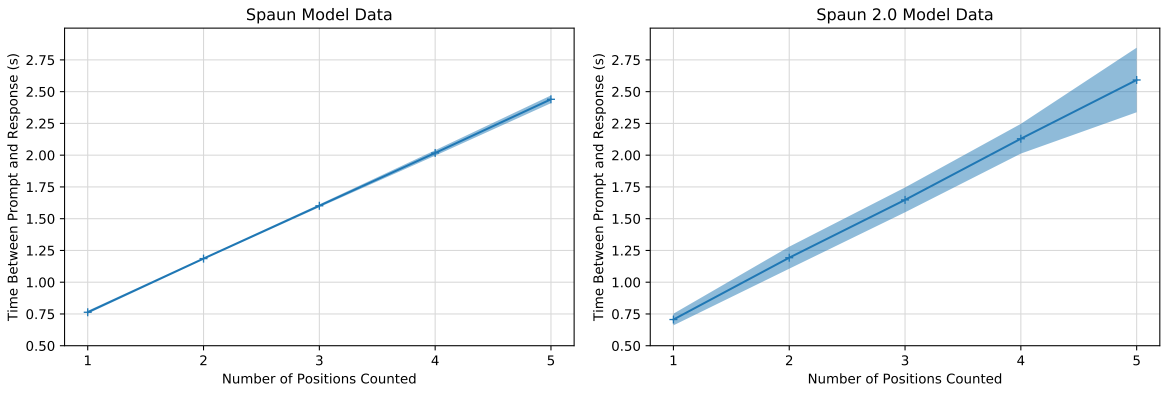


Figure 4.25: Comparison of Spaun and Spaun 2.0 response timings for the counting task. Shaded regions indicate +/- one standard deviation of the reported data. Refer to accompanying text for an analysis of the results. (Left) Results from the Spaun model (adapted from [Eliasmith et al., 2012] with permission). (Right) Results from the Spaun 2.0 model.

of the question answering task. Additionally, this effect is independent of the task type ("kind" versus "position"), resulting in similar recall curves between the two task types. Because the recency and primacy effects are a direct consequence of the memory implementation, the changes to Spaun 2.0's memory system should also affect the results obtained for the question answering task. As demonstrated by Figure 4.26, the improved integrator networks in Spaun 2.0 does result in higher recall accuracies for the question answering task, while retaining the characteristic U-shaped recall curves.

4.6.3.7 Rapid Variable Creation (RVC)

The Spaun 2.0 model implements changes to the memory and transformation system particularly to address the shortcomings of the original Spaun model's performance in the induction tasks. Comparing the performance of the original Spaun model and the Spaun 2.0 model on the RVC task demonstrates the effect these changes have on the behaviour of the Spaun model. For the plots discussed in this section, the Spaun 2.0 model was tasked with three variants of the RVC task. Each task variant consisted of three exemplar input-output pairs, followed by the query input list. The task variant descriptions are as follows:

- `aX` $\rightarrow$ `X`: The input list consists of one constant digit (constant across the three exemplar input-output pairs), followed by a variable digit (changes across the three exemplar pairs). The output list consists only of the variable digit.
- `aXb` $\rightarrow$ `Xb`: The input list consists of one constant digit, then by a variable digit, and finally

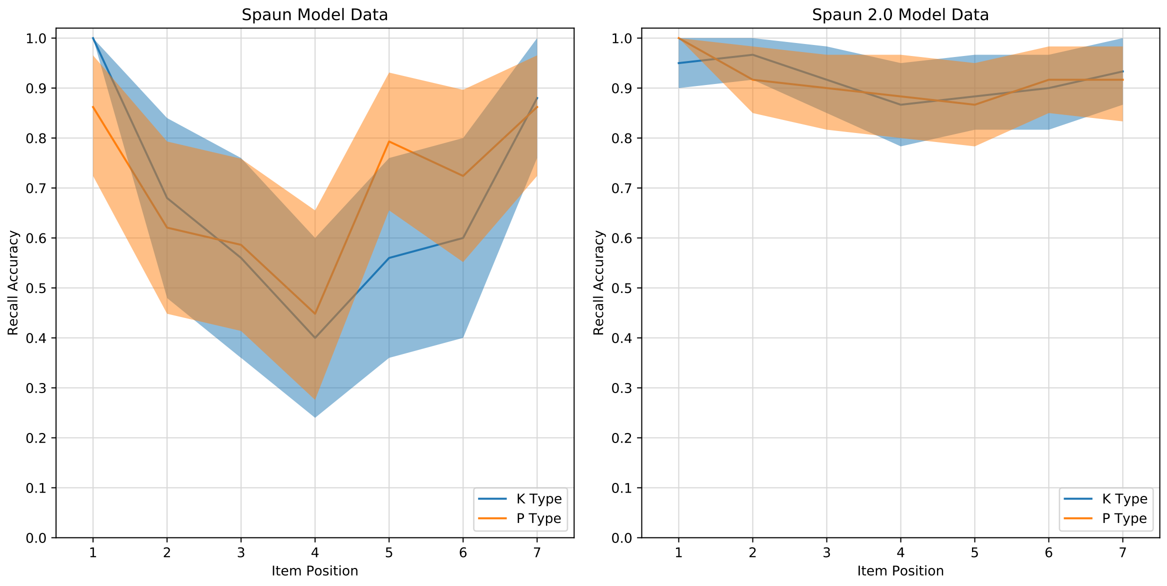


Figure 4.26: Comparison of recall accuracy for the question answering task between the Spaun and Spaun 2.0 models. For this task, list lengths of 7 items were used. Shaded regions indicate the 95% confidence intervals in the recall data. (Left) Recall data from the Spaun model as reported in [Eliasmith et al., 2012]. (Adapted from [Eliasmith et al., 2012] with permission). (Right) Recall data from the Spaun 2.0 model.

by another constant digit that differs in value to the first constant digit. The output list is composed of the variable digit and the last constant digit.

- `aaXb` $\rightarrow$ `Xb`: The input list consists of two identical constant digits, then by a variable digit, and finally by another constant digit that differs in value to the first set of constant digits. The output list is composed of the variable digit and the last constant digit. This variant is identical to the "biffle biffle *rose* zarple" example described in Section 3.2.8.

Figure 4.27 compares the original Spaun model and the Spaun 2.0 model's performance on the three variants of the RVC task mentioned above. It should be noted that while [Eliasmith et al., 2012] demonstrates Spaun performing both the "`aX` $\rightarrow$ `X`" and "`aaXb` $\rightarrow$ `Xb`" task variants, insufficient data was collected for the latter variant to construct a meaningful plot. For the data collected, two scoring criteria were used to generate the performance metrics. The first (relaxed) scoring criteria ignores extraneous digits in Spaun's response, and marks Spaun's answer as "correct" if the first N digits of Spaun's response fully match the expected response. I.e., if the expected response is the list `[45]`, the response `[4512]` would be considered correct, while the response `[4]` would be considered incorrect. The second (strict) scoring criteria only marks the response as correct if and only if all the digits in Spaun's response match the expected answer.

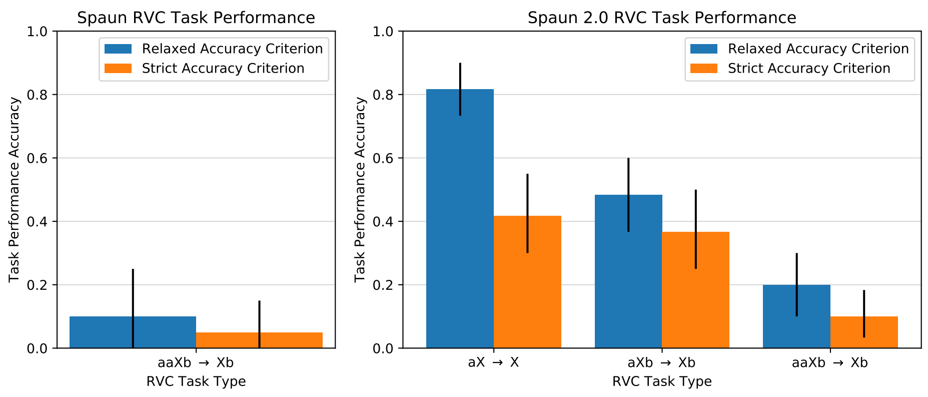


Figure 4.27: Comparison of task performance for the Spaun and Spaun 2.0 models on the rapid variable creation (RVC) task. Error bars indicate the 95% confidence interval of the reported data. For each task type, the model's task response is graded according to a relaxed performance metric, and a strict performance metric. See accompanying text for details on the performance metrics used, and for an analysis of the data collected. (Left) Performance of the Spaun model on the `aaXb` $\rightarrow$ `Xb` variant of the RVC task. (Right) Performance of the Spaun 2.0 model on 3 variants of the RVC task.

As illustrated in Figure 4.27, the changes made to the Spaun 2.0 model does result in an increase in Spaun's performance on the "`aaXb` $\rightarrow$ `Xb`" variant of the RVC task. However, a sharp decrease in performance is observed as the RVC task variants get more complex. Figures 4.28 and 4.29 show more detailed analysis of Spaun's responses to help enumerate the possible reasons for this performance drop.

Figure 4.28 compares Spaun's response to each of the 3 exemplar input-output pairs provided to it as part of the RVC task training set. From the graphs, it is observed that the responses of the original Spaun model favoured the third exemplar pair (i.e., the responses Spaun provided was for the third exemplar input list, and not the query list). The data also suggests that this issue is largely addressed in the Spaun 2.0 model, likely from the updates to the transform averaging network (specifically, the addition of the initialization condition – see Section 4.3.2), although other changes to the Spaun 2.0 model (e.g., the addition of the normalization network) may have contributed as well. Despite the reduction in the probability of Spaun answering with an exemplar response, the performance on the "`aaXb` $\rightarrow$ `Xb`" task variant has not significantly increased, indicating that some other factor is responsible for the poor performance on that variant of the task.

Figure 4.29 shows a breakdown of Spaun 2.0's responses compared to individual digits of the

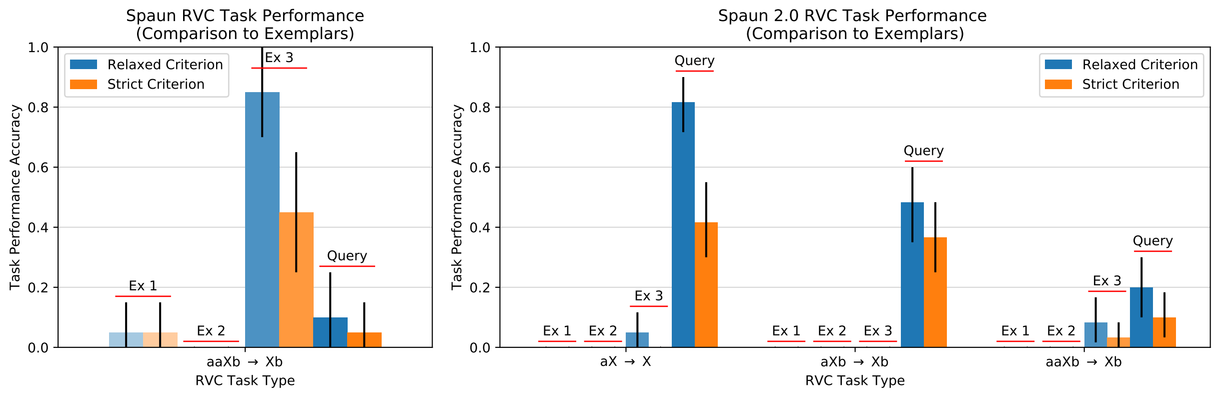


Figure 4.28: RVC task results for the Spaun and Spaun 2.0 models showing a breakdown of Spaun's response compared to the exemplars provided as part of the RVC task set up. For each task variant, the graph is divided into 4 sections, labelled "Ex 1" to "Query". These indicate the accuracy of Spaun's response when compared to each of the three exemplar input-output pairs, followed by the accuracy of Spaun's response to the query input list. (Left) Exemplar breakdown of results from the original Spaun model. (Right) Exemplar breakdown of results from the Spaun 2.0 model.

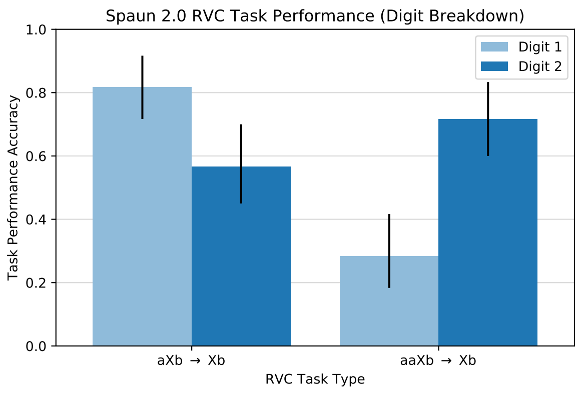


Figure 4.29: Comparison of Spaun 2.0's responses to individual digits of the expected task response. Note that an analysis of the "`Xb` $\rightarrow$ `X`" task variant results is not included in this figure as it would be identical to the "relaxed criterion" results (for Spaun 2.0, on the "`aX` $\rightarrow$ `X`" variant) illustrated in Figure 4.27.

"correct" task response. As the figure demonstrates, the Spaun 2.0 performs reasonably well at identifying (chance identification of an individual digit is 10%) both digits of the "`aXb` $\rightarrow$ `Xb`" variant, and the second digit of the "`aaXb` $\rightarrow$ `Xb`" variant. The reduction in the overall task performance is a consequence of the Spaun 2.0 model being unable to identify both digits correctly, simultaneously.

Using the analysis performed in Figures 4.27 through 4.29, several factors contributing to Spaun's performance on the RVC task can be inferred (see below). Further investigation to deduce the *actual* cause(s) of Spaun's performance issues with the RVC task is, however, left for future work.

- The weighting factor for the transform averaging factor was chosen to minimize the difference between the true average and the moving average of 5 transforms (see Section 3.2.8). Having a mismatched weighting factor for the RVC task manifests as Spaun's responses favouring the exemplar lists instead of the query list. Changing the weighting factor to favour the RVC task (as opposed to the fluid induction task) may increase Spaun's performance on the RVC task, however, doing so may negatively impact Spaun's performance on the fluid induction task. Additionally, from the data plotted in Figure 4.28, it will, at best, improve Spaun's performance by 10% on the "`aaXb` $\rightarrow$ `Xb`" variant of the RVC task.
- The list encoding schema (see Section 3.2.5) used for the information stored in memory might not be particularly suitable for the RVC task. As demonstrated by the examples detailed in Section 4.3.2, the results of the transformation calculation often result in superfluous digits being included in Spaun's final result, which makes it difficult for Spaun to effectively "remove" digits as part of the transform calculation. The effect the superfluous digits have on the task performance of Spaun can be inferred from the difference in accuracy numbers between the "relaxed" and "strict" criteria in Figure 4.27. However, this really only applies to the "`aX` $\rightarrow$ `X`" variant of the RVC task.
- The thresholds of the associative memories used in the decoding system might need to be adjusted to do a better job at filtering out the superfluous digits from Spaun's responses. However, care must be taken to ensure that changes to the decoding system appropriately affect Spaun's performance on its other tasks.
- The induction required to perform the RVC task correctly does not fall into the "sequence" task type that the original RPM sequence solver was designed for (see Section 3.1.5). As such, the RVC task should be tested with the full range of solvers implemented in the original RPM solver. The combination of outputs from the 3 solvers may boost Spaun's performance in the RVC task.
- Perhaps the most probable cause of Spaun's performance degradation (as the complexity of the RVC task increases) is the primacy and recency effects embedded within the representations stored in memory. These effects are achieved through the preferential weighting

of items at the start of the list (primacy), and the decay of items in memory (recency). While the semantic pointer as a whole can be normalized, this does not affect the individual weightings of each list item's contribution to the list memory representation, effectively resulting in non-unitary contributions to the list memory. As demonstrated in Section 4.3.2, the use of non-unitary semantic pointers in the transform calculations have a multiplicative effect on the final result. The list format of the RVC task is most susceptible to this effect, as the differences between the relative list positions of items in the input and output lists increase as the task variants get more complex. This is evidenced by the large drop in accuracy for the first digit of the "`aaXb` $\rightarrow$ `Xb`" task variant in Figure 4.29. Testing this hypothesis involves running Spaun on a RVC task variant where the relative list positions between the input and output is minimized (e.g., "`Xbaa` $\rightarrow$ `Xb`"). Additionally, tests can be conducted with a memory system that employs a different (non differentially weighted) list encoding to further verify this hypothesis.

4.6.3.8 Fluid Induction

While the changes to the memory and transformation system were directed at improving the Spaun's performance accuracy in both induction tasks, the changes were mostly focused at improving the fluid induction task in particular (as demonstrated in Section 4.3.2). Figure 4.30 compares the performance of the Spaun and Spaun 2.0 models on 9 different variants of the fluid induction task. These variants include:

- `[1][2][3]`: The digit sequence variant of the progressive matrix, where each cell contains a single digit that is one numerical value higher than the previous cell.
- `[3][2][1]`: The digit sequence variant of the progressive matrix, where each cell contains a single digit that is one numerical value lower than the previous cell.
- `[X][XX][XXX]`: The list length sequence variant of the progressive matrix, where each cell contains one more item than the previous cell. The first cell of each row always begins with 1 item in the list.
- `[X][XXX][XXXXX]`: The list length sequence variant of the progressive matrix, where each cell contains two more item than the previous cell. The first cell of each row always begins with 1 item in the list.
- `[XXX][XX][X]`: The list length sequence variant of the progressive matrix, where each cell contains one less item than the previous cell. The first cell of each row always begins with 3 items in the list.
- `[1][3][5]`: The digit sequence variant of the progressive matrix, where each cell contains a single digit that is two numerical values higher than the previous cell.
- `[5][3][1]`: The digit sequence variant of the progressive matrix, where each cell contains a single digit that is two numerical values lower than the previous cell.

- `[1][12][123]`: A combination of the digit and list length sequence variants of the progressive matrix. In addition to the lists growing by 1 item from the previous cell, the added item is one numerical value higher than the item added in the previous cell.
- `[3][32][321]`: A combination of the digit and list length sequence variants of the progressive matrix. In addition to the lists growing by 1 item from the previous cell, the added item is one numerical value lower than the item added in the previous cell.

A detailed analysis of the results is provided below.

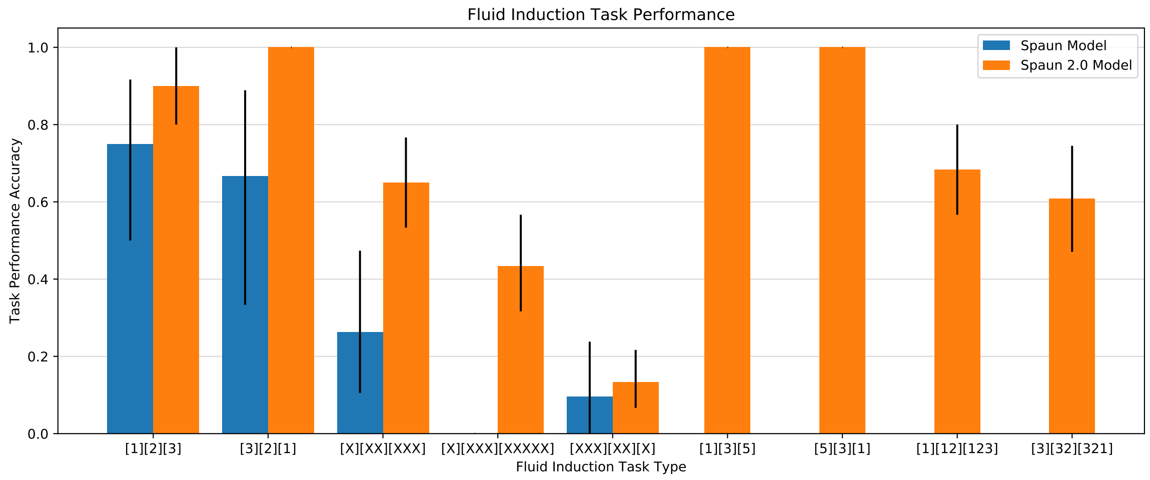


Figure 4.30: Comparison of task performance accuracy between the Spaun and Spaun 2.0 models for multiple variants of the fluid induction task. Shown are comparison between the Spaun and Spaun 2.0 models for 5 variants of the fluid induction task ("`[1][2][3]`" through "`[XXX][XX][X]`"). Also included are the answer generation accuracies for 4 fluid induction task variants not tested on the original Spaun model ("`[1][3][5]`" through "`[3][32][321]`"). The black error bars indicate the 95% confidence intervals of the reported data. See accompanying text for a detailed analysis of the data shown. Note that the size of the confidence intervals on the original Spaun data is due to the number of trials run (10 trials for Spaun, compared to 60 trials for Spaun 2.0).

In Eliasmith et al. [2012], the performance accuracy of the original Spaun model on the fluid induction task was reported to be 88%. This result was computed as the average accuracy for the "`[1][2][3]`", "`[3][2][1]`", and "`[X][XX][XXX]`" variants of the induction task. Additionally, it assumed Spaun had access to the 8 possible answers and was tasked to chose the correct answer. Using the same metric, the accuracy for the Spaun 2.0 model on the same set of matrices is 92.5% (raw accuracy of 80%, with a 95% confidence interval of 72.5% to 86.67%, with a match-adjusted rate of $80\% + 25\% \times 50\% = 92.5\%$ – see [Eliasmith et al., 2012] for a breakdown of the computation). As a comparison, human subjects averaged 89% on the Raven's progressive matrices that included only an induction rule (5 out of the possible 36 matrix types – see [Forbes, 1964]).

The results presented in Figure 4.30 use a different performance metric for a more direct comparison of the performance of the two Spaun models. First, there is no assumption that Spaun is provided with the possible answers to the matrix, rather Spaun is graded on the generation of the correct answer. Second, unlike the RVC induction task (see Section 4.6.3.7, responses are only marked correct if the entire answer matches the expected answer, and no partial matches are considered. An analysis of the results follows:

- `[1][2][3]` and `[3][2][1]`: Of the two changes to Spaun targeting the induction tasks, only the addition of the normalization network affects the performance accuracy on both of these matrix variants (see Section 4.3.2). From Figure 4.30, it can be seen that addition of the normalization network results in an increase of the mean performance accuracy of digit sequence variant of the progressive matrix (no statistically significant increase in the accuracy of the "`[1][2][3]`" variant, but a significant increase in accuracy of the "`[3][2][1]`" variant.)
- `[X][XX][XXX]` and `[X][XXX][XXXXX]`: The modification of Spaun 2.0's transform averaging memory was made particularly to address the performance accuracy on the list sequence variants of the progressive matrices. As the data illustrates, both the 1-digit-increase list sequence, and the 2-digit-increase list sequence variants received boosts in their performance for Spaun 2.0. The reduction in performance when compared to the digit sequence variants can be attributed to the list representation used by Spaun. Because Spaun is unable to discern patterns in the input stimulus, each list is treated as a list of distinct (and independent) items, and Spaun has to remember the entire list to perform the task correctly. Human subjects, however, would be more likely to encode the lists as "N items of X", simplifying the representation, and increasing performance on this task. Note that using the strict grading criteria described above, the original Spaun model was unable to correctly answer any of the trials of the "`[X][XXX][XXXXX]`" variant of the task.
- `[XXX][XX][X]`: Similar to the results of the RVC induction task, the Spaun and Spaun 2.0 models perform poorly on the list sequence variant that involves the removal of an item as part of the transformation. See Section 4.6.3.7 for a discussion on this phenomenon.
- `[1][3][5]` and `[5][3][1]`: From the perspective of the SPA (and the encoding of list information within Spaun), these task variants are identical to the "`[1][2][3]`" and "`[3][2][1]`", with the only difference being the result of the transform computation. As such, similar results are seen between all four task variants.
- `[1][12][123]` and `[3][32][321]`: The performance of this variant is dependent on Spaun's performance on the digit sequence variant, and the list sequence variant of the progressive

matrices. Since Spaun 2.0 performs perfectly on the digit sequence matrix variants, the results of the "`[1][12][123]`" and "`[3][32][321]`" should be similar to the results of the "`[X][XX][XXX]`". This is indeed the case, as demonstrated by the data.

4.7 Discussion

Overall, the modifications made to the original Spaun model to produce Spaun 2.0 have resulted in significant functional improvements across many of the original tasks. In particular, changes to the integrator networks in the working memory system have resulted in improvements in recall accuracies for both the working memory and question answering tasks. Additionally, the rework of the averaging memory network and the addition of the vector normalization network have increased performance accuracy for the Spaun's two induction tasks. Aside from the changes to the implementation details of Spaun's networks, Spaun's code base has also been significantly reorganized, focused on improving the modularity of the Spaun 2.0 network implementation. These improvements make Spaun 2.0 more robust and more extensible than the previous version. In the next chapter, this is taken advantage of, and a significant new behaviour is introduced to the model. Specifically, Spaun 2.0 is extended to have a more complex visual system, an adaptive motor system, and most importantly, the ability to process and execute customized instructions.

Chapter 5

Extensions to Spaun 2.0

The process of formalizing and reimplementing the Spaun model presented the opportunity to generalize the architectural design of each of Spaun's modules (Section 3.3), and to reorganize Spaun's code base (Section 4.6.2). This was done to enhance the modularity of the Spaun model, providing it with the theoretical ability to swap and test different individual (or multiple) module implementations without affecting the rest of the model.

This chapter explores the methods involved in implementing and integrating new module variants for the Spaun 2.0 model, and discusses the additional functionality they bring to the Spaun 2.0 model. This section also looks at adding new tasks, and modifying core functionality of the Spaun 2.0 model. Three extensions to the Spaun model, each increasing in complexity from the last, are described in this section. They are, in order:

1. Adaptive motor control – to demonstrate how the modularity of the Spaun 2.0 can be used to add functionality to only one of Spaun's modules.
2. ImageNet visual system – to demonstrate how the modularity of the Spaun 2.0 can be used to add functionality to only one of Spaun's modules, and to show how new tasks are added to the Spaun architecture.
3. Generalized instruction following – to show how new tasks are added to the Spaun architecture, and demonstrate how new modules can be added to the Spaun architecture to modify the behaviour of core Spaun modules.

5.1 Adaptive Motor Control

One feature omitted from the Spaun 2.0 implementation of the REACH model described in Section 4.2.2 is the ability of the controller to compensate for non-linear perturbations applied to the end effector of the arm it is controlling. In [DeWolf, 2014], this adaptation is demonstrated using the perturbed reaching task, whereby subjects are instructed to move their arm to one of eight points – equally distributed around a circle – while an unknown force is applied as they affect the movement. Figure 5.1 illustrates the recorded trajectories of a simulated arm with two different types of velocity force fields (the force applied is proportional to the velocity of the arm), and the result using the REACH model after adaptation.

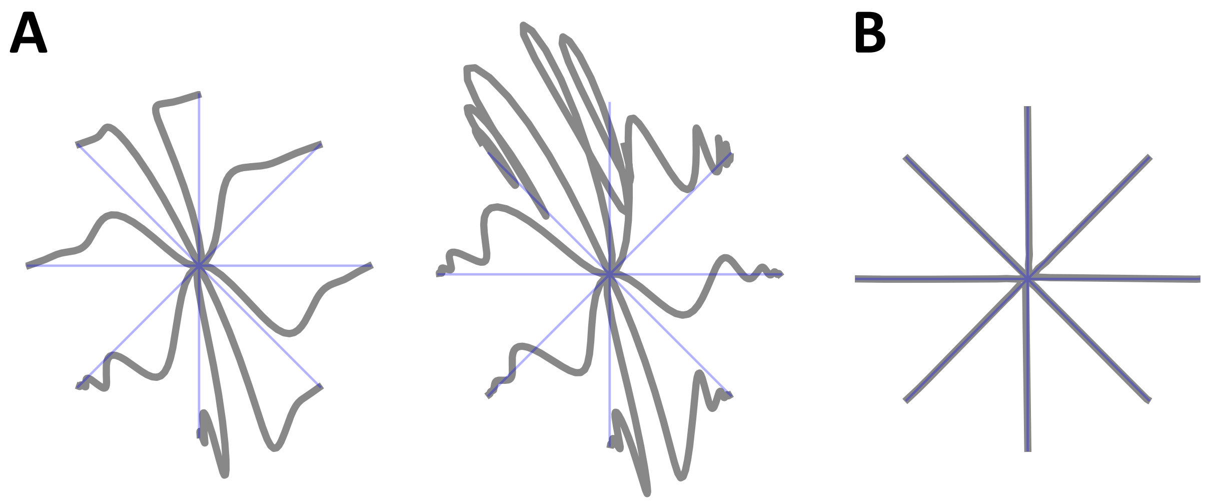


Figure 5.1: Reaching task trajectories recorded for a simulated arm showing the effects of various force fields, and the effect of adaptation using the REACH model. The thin lines indicate the ideal reach trajectory, while the thick lines indicate the trajectory of the simulated arm under the effects of the force field. (**A**) Recorded trajectories of the simulated arm in an end-effector based velocity force field (left) and a joint based velocity force field (right). (**B**) Recorded trajectory of the simulated arm when controlled by the REACH model, after having adapted to the joint velocity based force field.

5.1.1 Spaun 2.0 Implementation

In [DeWolf, 2014], the adaptation in the motor hierarchy is achieved by using the control output (u) of the M1 population as an error signal for an additional adaptive control signal (u_{adapt}) projected to the arm. The final control signal projected to the arm is then $u + u_{adapt}$. Modifying the Spaun 2.0 motor hierarchy discussed in Section 4.2 to include the adaptive functionality involves adding these same projections to the motor hierarchy, as illustrated by Figure 5.2. Moreover, as the inputs and outputs from the motor module remain unchanged, the different (non-adaptive, and adaptive) motor systems can be interchanged with no additional changes to the rest of the Spaun network.

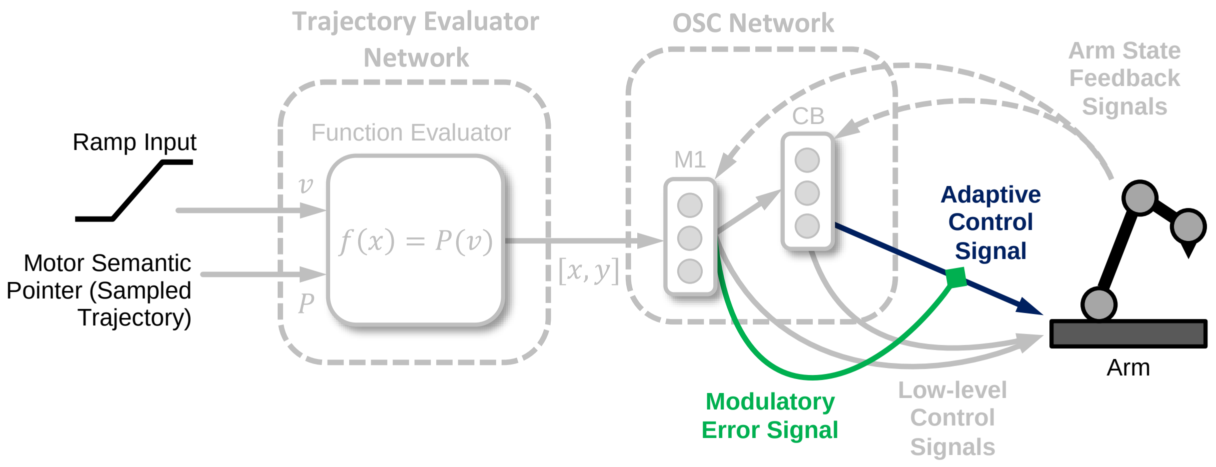

Trajectory Evaluator Network
OSC Network
Arm State Feedback Signals
Function Evaluator
Ramp Input
v
$f(x) = P(v)$
P
Motor Semantic Pointer (Sampled Trajectory)
$[x, y]$
M1
CB
Adaptive Control Signal
Arm
Modulatory Error Signal
Low-level Control Signals

Figure 5.2: Schematic of the Spaun 2.0 motor hierarchy modified to include the adaptive control signal. Note that the common functional components of the Spaun 2.0 motor hierarchy (from Figure 4.8) have been grayed out to highlight the changes made to the network to implement the adaptation algorithm.

5.1.2 Results

In the REACH model, motor control adaptation was demonstrated using the 8-point reaching task illustrated in Figure 5.1. The repertoire of outputs of the Spaun model is, however, constrained to the digits from 0 to 9. Thus, to demonstrate the adaptation of the motor hierarchy in action, the Spaun model was instructed to repeatedly perform the list recall task with its simulated arm placed in a velocity based force field.

Figures 5.3 and 5.4 show the recorded inputs and outputs of Spaun for eight sequentially performed 4-digit list memory tasks, with its simulated arm placed the presence of a joint velocity based force field. In Figure 5.3, the Spaun 2.0 model did not feature the adaptive motor hierarchy. As the recorded outputs illustrate, no improvement in its written outputs is observed regardless of the amount of time spent experiencing the effects of the force field.

In Figure 5.4, the Spaun 2.0 model included the adaptation in the motor hierarchy, and consequently, shows improvement in its written output as more time is spent experiencing the effects of the force field. Additionally, on the seventh list memory task, Spaun wrongly identifies the first digit as a “6” instead of a “4”, but is still able to adequately reproduce a “6”, despite having not previously experienced writing the “6” under the effects of the force field. This demonstrates that the adaptive motor hierarchy is generalizing to the force field, rather than to the individual digits.

For reference, Figure 5.5 provides sample outputs from Spaun 2.0’s non-adaptive motor system without the effect of the force field.

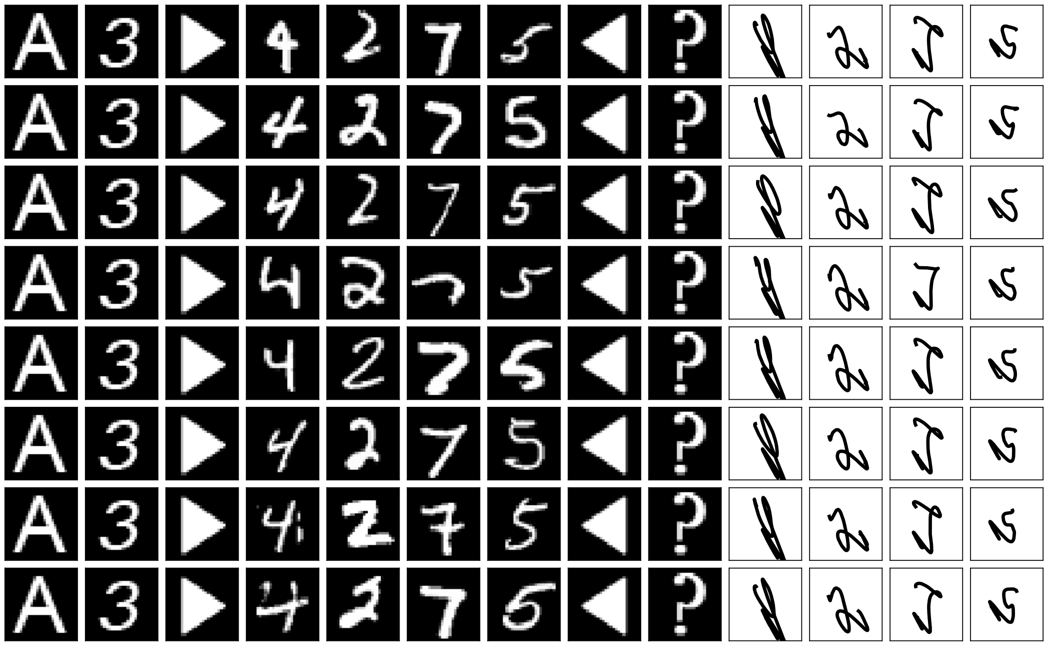


Figure 5.3: Recorded input stimuli and produced outputs for the Spaun 2.0 model with a non-adaptive motor hierarchy while its arm is subjected to a joint velocity modulated force field. To differentiate between the two, the input stimuli are presented against a black background, while Spaun’s written responses are against a white background. See accompanying text for an analysis of the results.

5.2 ImageNet

ImageNet [Russakovsky et al., 2015] is a dataset of over a million images used to test visual classification networks, similar in purpose to the MNIST dataset. However, while the images in the MNIST dataset are grayscale, 28 × 28 pixels in size, and only feature handwritten digits (i.e., there are 10 possible classification classes), the images from the ImageNet dataset are fully

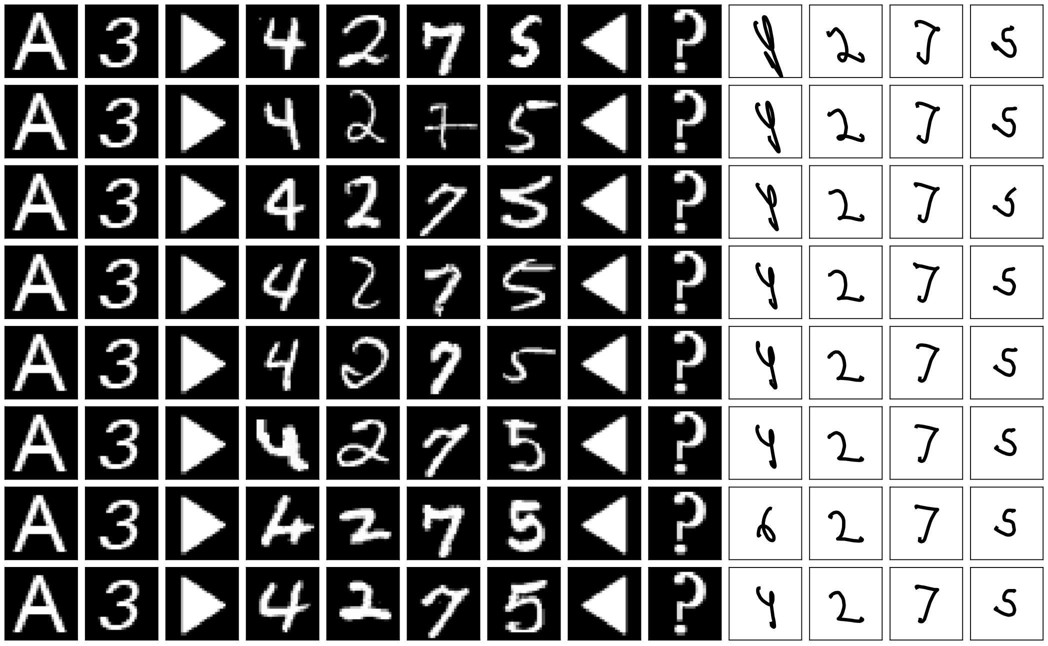


Figure 5.4: Recorded input stimuli and produced outputs for the Spaun 2.0 model with an adaptive motor hierarchy while its arm is subjected to a joint velocity modulated force field. See accompanying text for an analysis of the results.

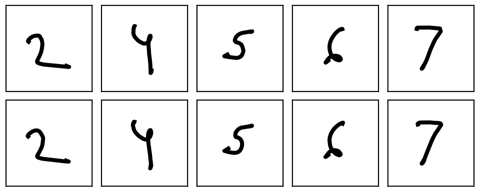


Figure 5.5: Reference outputs for the digits “2”, “4”, “5”, “6”, and “7” written by Spaun under normal (no external force field, no motor adaptation) conditions.

coloured (RGB), can vary in size (typically 256 $\times$ 256 pixels), and are classified into 1000 output classes ranging from specific animals (e.g. specific breeds of dogs) to mechanical devices (e.g. police cars, ambulances). Figure 5.6 shows several example images from the ImageNet dataset, along their assigned classification class(es).

In 2016, Hunsberger and Eliasmith [2016] combined the AlexNet visual system architecture [Krizhevsky et al., 2012] with the methods described in Section 4.1.1 to develop a spiking visual hierarchy capable of classifying the images from the ImageNet dataset. The spiking Ima-

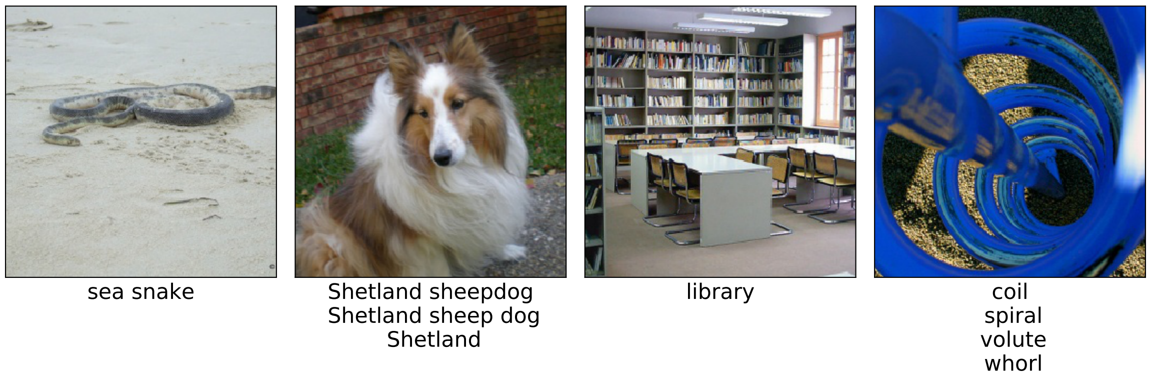


Figure 5.6: Example images from the ImageNet dataset [Russakovsky et al., 2015], and their associated class label.

geNet visual hierarchy consists of 6 layers of spiking neurons, containing 193,600, 139,968, 64,896, 43,264, and 43,264, 8,192 neurons respectively, for a total of 493,184 spiking LIF neurons. The network receives input as a full colour (RGB) 224 $\times$ 224 pixel image (cropped to account for the differing sizes of images found in the ImageNet dataset), and produces a 1000-dimensional vector as an output, with each vector element corresponding to 1 of the 1000 possible classes in the ImageNet dataset.

This section describes the process of integrating the ImageNet visual hierarchy with the Spaun 2.0 model. The discussion proceeds in three parts: first, Section 5.2.1 details the Spaun implementation of the ImageNet visual hierarchy, and discusses the modifications made to the network to facilitate the integration with Spaun. This is followed in Section 5.2.2 by a discussion regarding the addition of a new task to the Spaun collection of tasks, necessary because Spaun's output is limited to the range of digits from 0 to 9. Finally, Section 5.2.3 explores the results of data collected using the ImageNet capable Spaun 2.0 model.

5.2.1 Spaun Implementation

Adapting the ImageNet visual hierarchy to the Spaun network required several modifications to be made to the interface between the visual hierarchy and the rest of the Spaun 2.0 visual system. These modifications are the focus of this section.

5.2.1.1 Spaun Control Characters

The first and most obvious change is to the input stimulus provided to the visual hierarchy. The ImageNet dataset does not contain the Spaun specific control characters necessary for Spaun to operate correctly, so the existing 28 × 28 pixel control character images were up-scaled to match the input resolution (224×224 pixels) of the ImageNet visual hierarchy. A Gaussian filter was also applied to the images to smooth out the pixel boundaries introduced as a result of the up-scaling process. Lastly, the images were converted from grayscale to RBG to match the full colour input expected by the ImageNet visual hierarchy. Figure 5.7 compares a sample MNIST-based control character image with the equivalent ImageNet compatible image.

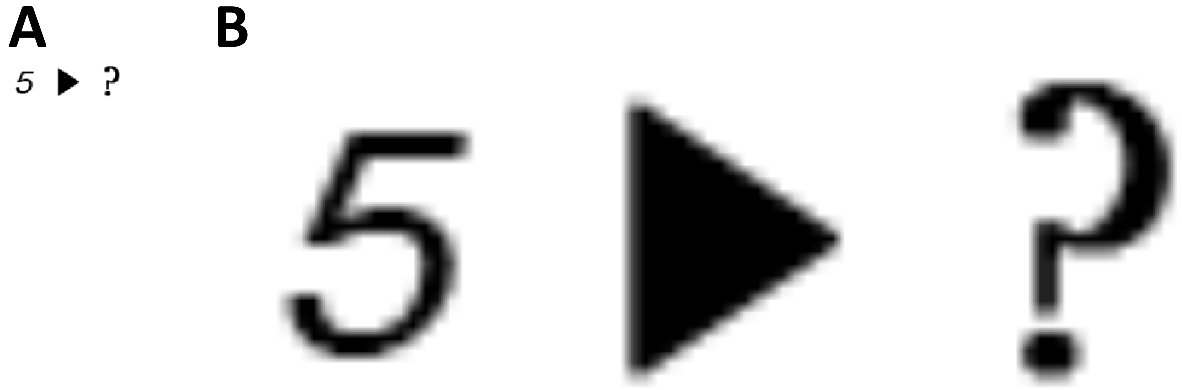


Figure 5.7: To-scale comparison of Spaun control characters used with the MNIST visual hierarchy (**A**), and the ImageNet visual hierarchy (**B**).

5.2.1.2 Change Detect Network

Section 4.1.2 describes the modifications made to the original Spaun visual change detection network to remove the artificial dependence on the "blank" input stimulus. The changes made to the change detection network shifted its operation from the output of the visual classifier (512-dimensional output) to working directly with the pixel-based input to the visual hierarchy. For the MNIST dataset, the images are 28 × 28 pixels, so the change detection network has an input dimensionality of 768 (28 × 28).

In contrast, the ImageNet visual hierarchy receives 224 × 224 pixel images. Additionally, the images are coloured, with one channel each for the red, green and blue components respectively. Thus the total dimensionality of the input to the ImageNet visual hierarchy is $224 \times 224 \times 3 = 150,528$. If every single pixel were used to generate the output of the change detection network, using 30 neurons per differentiator results in a total neuron count of 4,515,840 (more than 9 times larger than the visual hierarchy!).

To reduce the overall neuron count of the ImageNet visual system, instead of projecting every pixel to the change detection network, projections from a maximum of 5000 randomly selected pixels are used. This caps the neuron count for the change detection network to a maximum of 150,000 neurons.

5.2.1.3 Classifier Networks

As mentioned above, the output of the ImageNet visual hierarchy is a 1000-dimensional vector, with each element of the vector representing each class found in the ImageNet dataset. To convert the output of the visual hierarchy into the 512D semantic pointer format used by Spaun's other modules, the 1000D output of the visual hierarchy is projected into an 1000-element associative memory (see Section 2.5.5.5) that associates each of the 1000 output classes with a randomly generated 512-dimensional semantic pointer.

Unlike the MNIST visual hierarchy, the classification accuracy of the ImageNet visual hierarchy can either be reported as "Top-1" or "Top-5". "Top-1" refers to the classification accuracy when only the highest output class is considered for each image, whereas "Top-5" refers to the classification accuracy when the top 5 highest output classes are considered in the accuracy computation (i.e., the output result is marked "correct" if any one of the top 5 output classes match the expected class for the image).

For the Spaun implementation of the ImageNet-based visual system [Hunsberger and Eliasmith, 2016], classifying the inputs according to the "Top-1" criteria involves using a winner-take-all (WTA) network to restrict the output of the associative memory to only one of the 1000 possible classes. This is the method already in use for the MNIST-based visual system. Implementing the "Top-5" criteria is however, not possible. Instead, it is approximated by removing the WTA network, and adjusting the thresholds of the neural ensembles in the associative memory such that across the entire set of training images, the *maximum* number of output classes projected to the output of the associative memory is 5. Through trial and error, the threshold value of 0.21 was determined to produce this effect, with most images projecting 2 class semantic pointers to the output of the associative memory. To differentiate the difference in behaviour, this scoring criteria will be referred to as "Top-N". The results presented in Section 5.2.3 demonstrates the difference in classification accuracy when using the "Top-1" and "Top-N" criteria.

Because the Spaun control characters were not included in the set of training images, the output classes of the ImageNet visual hierarchy do not include a class for each of the control characters. Instead of repeating the lengthy process of training the ImageNet visual hierarchy specifically to include Spaun's control characters, it was hypothesized that because the images in the ImageNet dataset were sufficiently diverse, the semantic pointer output of the second layer of the visual hierarchy could be used to identify the Spaun control characters. This mandated the used of a separate associative memory network to classify *just* the control characters. However, by using the output of the "control character" associative memory to inhibit the "ImageNet" associative memory, this two-associative-memory approach offered the advantage of being able to give the identification of the control characters priority over the ImageNet images (i.e., if Spaun thinks the input image is a control character, it will not attempt to classify it under the 1000 ImageNet classes), making Spaun less prone to errors in classifying the control characters (which are crucial to Spaun's correct operation). This prioritization is not possible using the "standard" single associative memory approach.

5.2.1.4 Spaun 2.0 ImageNet Visual System

Using the modifications discussed in the previous sections, the full ImageNet visual system can be constructed. To summarize the architectural modifications:

- Instead of receiving projections from all of the pixels of the input stimulus, the change detection network receives projections from a randomly selected subset of pixels.
- The 1000-dimensional output of the visual hierarchy is projected into an associative memory that maps the 1000D vector into a 512D semantic pointer.
- A two associative memory configuration is used to prioritize the classification of Spaun's control characters over the classification of the ImageNet images.

Figure 5.8 shows the network schematic of the final Spaun 2.0 implementation of the visual system. Note that because the output of the associative memory networks (and thus the output of the visual system) remains a 512 dimensional semantic pointer, no additional changes to the rest of the Spaun 2.0 network are required to integrate the ImageNet functionality into Spaun.

5.2.2 Stimulus Matching Task

While the previous section details how the ImageNet visual hierarchy can be integrated into Spaun, one crucial concern remains to be addressed. Because Spaun's output is restricted to the digits from 0 to 9, and no mapping has been established between each of the possible ImageNet classes and a written response, all of Spaun's existing tasks cannot be used to demonstrate a successful ImageNet integration.[42] To address this issue, a new task – the stimulus matching

[42]Technically, probes can be used to record neural activity from the memory networks to demonstrate that Spaun is able to internally represent the ImageNet classes, but this is not in the spirit of Spaun's "input-to-output" encapsulation.

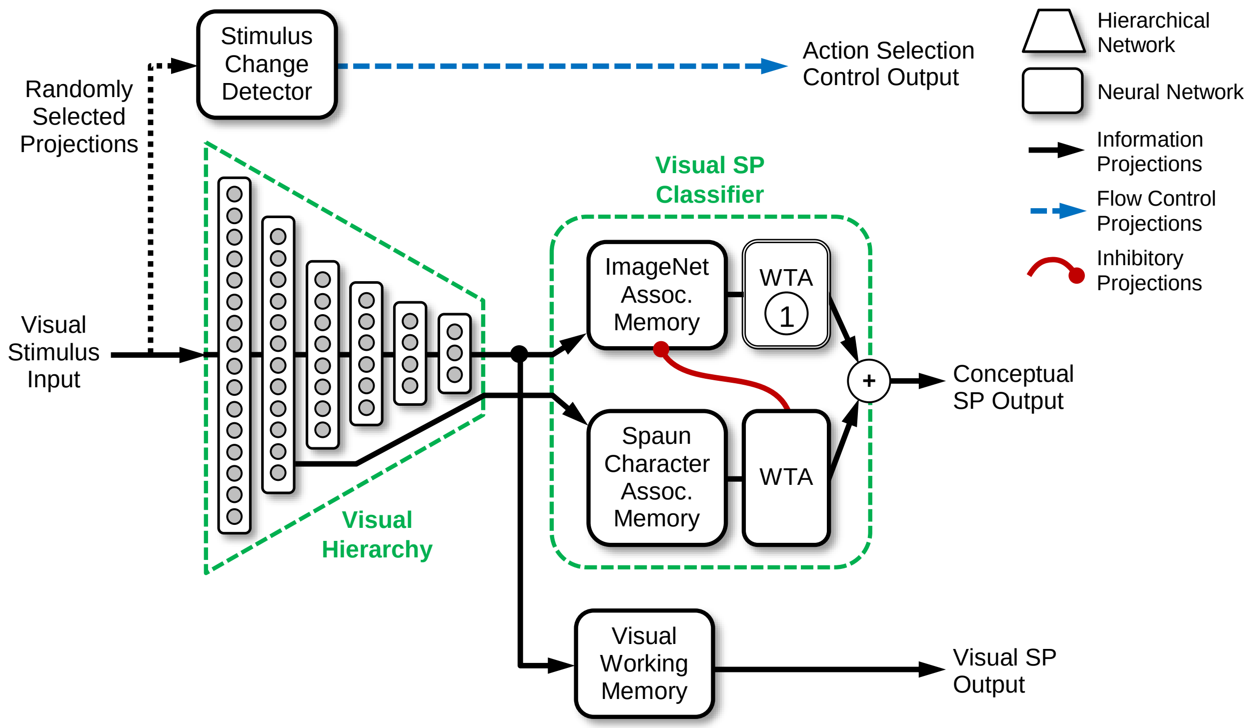


Figure 5.8: Network schematic of the Spaun 2.0 implementation of the ImageNet capable visual system. Note that the winner-take-all network (①) is included in the visual classifier network to implement the "Top-1" classification criteria, and is omitted from the visual classifier to implement the "Top-N" classification criteria. Refer to the accompanying text for details on the modifications made to the network shown in Figure 3.21 to allow Spaun to process images from the ImageNet dataset.

task – was added to Spaun's repertoire of tasks. This section discusses the specifics of the new task, and explores how existing Spaun networks can be used to implement this task with minimal additions to the Spaun 2.0 model.

Task Description: The stimulus matching task involves Spaun's indicating if there is a class-based match between the two presented stimuli. Spaun is to write a "1" if a match is identified, and a "0" otherwise.

Task Syntax: The input character sequence for the stimulus matching task follows the form:

$$\mathbf{A}\ \mathbf{C}\ \blacktriangleright\ x\ \blacktriangleleft\ \blacktriangleright\ y\ \blacktriangleleft\ \mathbf{?}$$

In the character sequence, "**AC**" informs Spaun that the task to follow is the stimulus match (comparison) task. After this, two single-digit lists (x and y) containing the items to be compared are presented. Finally, Spaun is prompted for its response with the question mark control character, after which Spaun is expected to respond with either a "0" (no match found) or a "1" (match found). It is important to note that the task is not limited to the ImageNet dataset, and should work with any variants of the visual system.

Conceptual Implementation: Using the SPA, the comparison of two semantic pointer representations is typically achieved by computing the dot product between the two representations. If the result of the dot product is above a pre-determined threshold, a match is declared. Once a match is declared (or a "non-match" outcome is decided), the outcome of the dot product operation is mapped onto their respective semantic pointer list representations to produce the appropriate "0" or "1" motor output (similar to the question answer task output mapping – see Section 3.2.7).

System Requirements: For this task, Spaun is required to have:

- At least two working memories, one for each of the two input lists.
- A neural network (dot product network) to perform the comparison between the stored representations of the two input lists.
- An method to map the output of the dot product network to the "0" and "1" responses Spaun is expected to provide.

Functional Implementation: Adding a new task to Spaun follows a straightforward process. First, the task-specific system requirements are determined (see above). From the system requirements, and after analyzing the existing Spaun model, the additional component(s) required to implement the new task are identified and implemented – prioritizing the reuse of existing networks. Finally, condition-consequence pairs are added to the action selection hierarchy to construct the task processing logic required for Spaun to perform the new task.

Looking at the system requirements for the stimulus matching task, Spaun already contains the necessary number of working memory networks for this task. Additionally, the dot product (compare) network used to determine the stopping condition in the counting task can be reused for the stimulus matching task. The only system requirement not met by the existing Spaun 2.0 network is the need for a mapping between the output of the dot product network and the list-format semantic pointer representation required by the motor system (to produce the "0" or "1" output). This functionality can be achieved by using an associative memory network to map the output of the comparison network to the desired list representation. This approach is similar to the method used to map the output of the binding network onto the list representations required for the output of the question answering task (see Figure 3.17). Figure 5.9 illustrates the modifications made to Spaun 2.0's transformation system to support the implementation of the stimulus matching task.

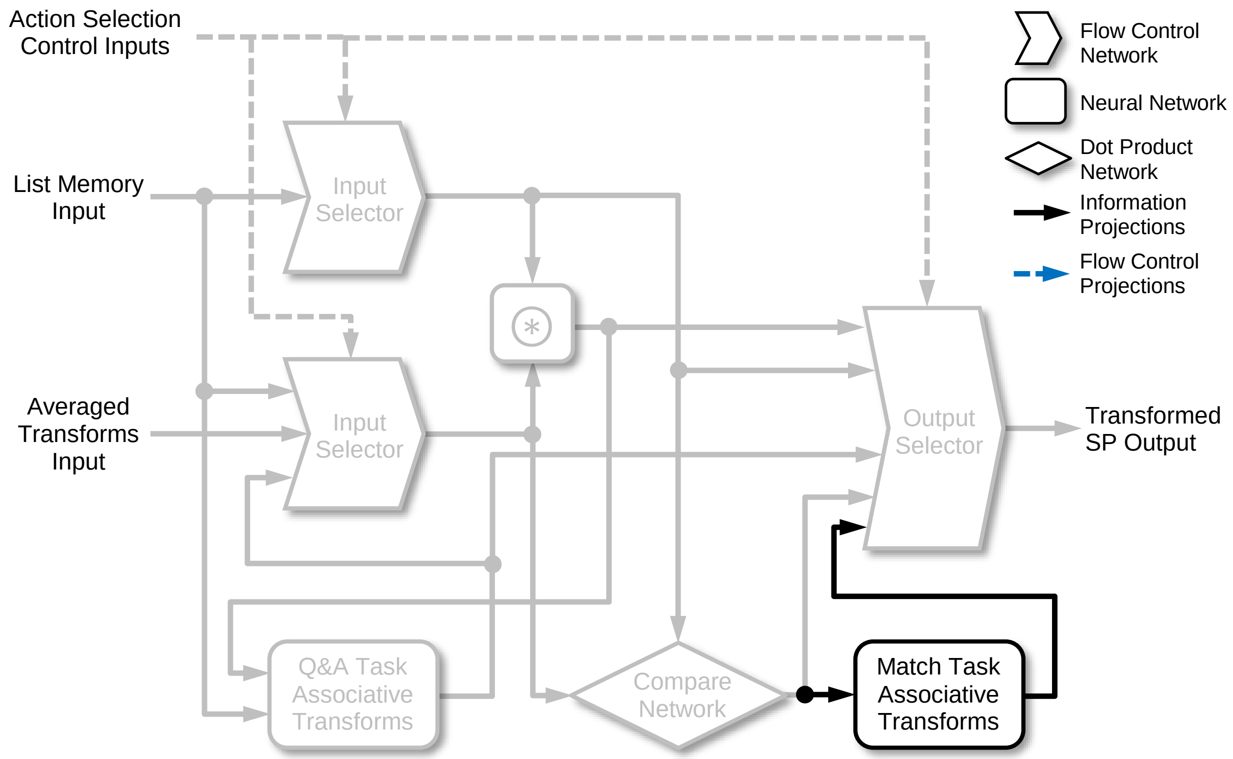


Figure 5.9: Schematic of Spaun 2.0's transformation system updated to support the stimulus matching task. Pre-existing networks and projections from the Spaun 2.0 implementation (see Figure 3.26) have been grayed out to better indicate the network changes added to support the stimulus matching task.

Modifying the action selection hierarchy is perhaps the most complex step to adding a task to the Spaun 2.0 model. Because Spaun's tasks are visually driven, the necessary modifications to the action selection hierarchy are systematically determined by stepping through the task steps determined by the control character sequence (similar to the steps taken in the RVC example from Section 3.3.8.1).

To start, so that Spaun keep track of the status of the matching task, a new semantic pointer (**COMPARE**) is added to the list of task representations (refer to Table 3.5). Next, it needs to be determined where Spaun is to store the representations of the two lists it is presented. As part of the requirements for the counting task, the transformation system (which houses the dot product network to be used for the matching task) is already configured to route information from the "Memory Block 2" and "Memory Block 3" components of the working memory system.[43] Additionally, since Spaun is configured to store the incoming input into "Memory Block 2" during the **TRANSFORM1** task processing stage, and into "Memory Block 3" during the **TRANSFORM2** task processing stage, these task processing stages can be reused for the matching task. The first two condition-consequence pairs for this task are then:

$$0.5 \times (task \bullet \mathbf{INIT} + vision \bullet \mathbf{C}) \longmapsto \mathbf{COMPARE} \Rightarrow task, \\ \mathbf{TRANSFORM1} \Rightarrow task\_stage \tag{5.1}$$

$$0.5 \times (task \bullet \mathbf{COMPARE} + task\_stage \bullet \mathbf{TRANSFORM1}) \longmapsto \mathbf{TRANSFORM2} \Rightarrow task\_stage \tag{5.2}$$

After remembering the list containing the second stimulus, Spaun is expected to produce a "match" or "no-match" response ("1" or "0", respectively). In order to do this, Spaun's updated transformation system (see Figure 5.9) has to be configured to route information from "Memory Block 2" and "Memory Block 3" to the dot product network, and from the output of the match-response associative memory to the output of the transformation system. Since no task processing stage fulfils this exact configuration requirement, a new task stage (**TRANSFORMC**) is added to the list of task processing stages (refer to Table 3.6). The appropriate action selection hierarchy transforms (see Section 3.3.8.2) are then added to set the correct routing paths within the transformation module during the **TRANSFORMC** task stage, as shown in Figure 5.10. The third condition-consequence pair is then defined as:

$$0.5 \times (task \bullet \mathbf{COMPARE} + task\_stage \bullet \mathbf{TRANSFORM2}) \longmapsto \mathbf{TRANSFORMC} \Rightarrow task\_stage \tag{5.3}$$

Finally, because the output of the transformation system is a list representation – which requires no further processing before the decoding (to handwritten outputs) stage – when Spaun is prompted with the question mark prompt, the task decoding state (see Table 3.7) needs to be set to **FORWARD**, concluding the last condition-consequence pair required to implement the

[43]During the counting task, "Memory Block 2" stores the **NUM_COUNT** representation, while "Memory Block 3" stores the **CURR_COUNT** representation. Both are routed to the dot product network to determine when the counting task stopping condition is met (see Section 3.2.6).

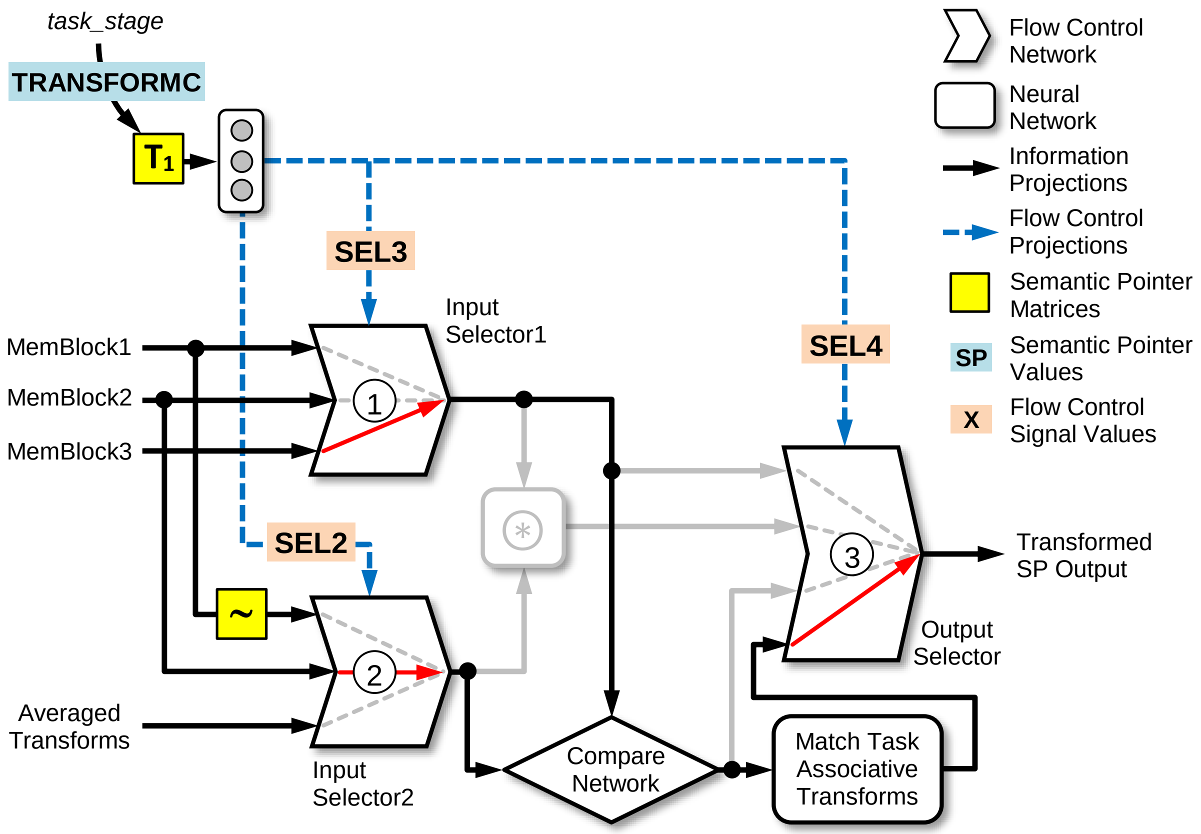


Figure 5.10: Illustration of the flow control signal configuration for the transformation module in the matching task specific "**TRANSFORMC**" task stage. Here, the actions selection hierarchy transform ($\mathbf{T}_1$) is configured such that when *task_stage* has a value of "**TRANSFORMC**", the "select3" signal is set for "input selector 1", and the "select2" signal is set for "input selector 2". This routes information from *mem*3 (①), and from *mem*2 (②) to the dot product network. Additionally, the "select4" signal is set for the "output selector" ensuring that the output of the match task associative transform (see accompanying text) is routed as the output of the transformation system (③). Note that the projections and networks unused by the matching task have been grayed out to enhance the clarity of the schematic.

matching task in Spaun:

$$0.5 \times (task \bullet \textsf{\textbf{COMPARE}} + vision \bullet \textsf{\textbf{?}}) \longmapsto task\_stage \Rightarrow task\_stage, \textsf{\textbf{FORWARD}} \Rightarrow task\_decode \tag{5.4}$$

To summarize, adding the stimulus matching task to the Spaun 2.0 model required the following additions to be made:

- The addition of an associative memory network to the transformation network to implement the matching task output mapping. This is shown in Figure 5.9.
- The addition of the **COMPARE** semantic pointer to the list of task representations, and the addition of the **TRANSFORMC** semantic pointer to the list of task stage representations.
- The addition of layer 2 (of the action selection hierarchy) transform matrices to configure the path of information flow in the transformation system during the **TRANSFORMC** task stage. This is demonstrated in Figure 5.10.
- The addition of four condition-consequence pairs to implement the task control logic for the stimulus matching task. These are Eqs. (5.1) through (5.4).

5.2.3 Results

In order to quantify the efficacy of the Spaun 2.0's ImageNet and stimulus-match task implementations, three types of tests were conducted. First, as with the implementations of the original Spaun and Spaun 2.0 MNIST visual hierarchies, the classification accuracy of the Spaun 2.0's non-integrated ImageNet visual hierarchy is compared against its non-spiking counterpart. Next, to test the correctness of the stimulus-match task, data is collected using Spaun 2.0's (verified-working) MNIST visual hierarchy. Finally, the fully integrated ImageNet visual hierarchy and the stimulus-match task implementation are combined to demonstrate the new ImageNet processing capabilities of Spaun.

Standalone Visual Hierarchy: Here, the results of Spaun's non-integrated ImageNet visual hierarchy is compared to the non-spiking, rectified-linear (ReLU) neuron-based ImageNet visual hierarchy of the same architectural design from [Hunsberger and Eliasmith, 2016]. Using the "Top-1" classification criteria, Spaun's ImageNet visual hierarchy achieved a 50.4% classification accuracy, while the non-spiking visual hierarchy achieved a 54.6% classification accuracy.

Using the "Top-5" classification criteria, Spaun's ImageNet visual hierarchy achieved a 62.2% classification accuracy, while the non-spiking visual hierarchy achieved a 79.1% classification accuracy. It is important to restate (see Section 5.2.1.3), however, that due to the limitations

MNIST	0	1	2	3	4	5	6	7	8	9
0	94.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
1		98.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
2			98.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
3				95.7%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
4					99.1%	0.0%	0.0%	0.0%	0.0%	0.0%
5						96.5%	0.0%	0.0%	11.1%	0.0%
6							93.9%	0.0%	0.0%	0.0%
7								96.5%	0.0%	0.0%
8									92.3%	0.0%
9										91.4%

Table 5.1: Stimulus matching task performance accuracy using the MNIST visual hierarchy, showing the probability of identifying one input stimulus digit against another input stimulus digit. The diagonal (shaded gray) indicates the accuracy of correctly identifying two matching stimuli, while the off-diagonals indicate the percentage of false positives in matching two non-matching stimuli.

of the neural implementation, Spaun's ImageNet visual hierarchy is incapable of achieving the "Top-5" criteria for all image classes. Thus, the comparison of the "Top-5" accuracies is difficult to interpret.

MNIST Stimulus-match Task: In order to verify the correct operation of the newly added stimulus matching task, data was collected running the task with Spaun 2.0's MNIST-based visual hierarchy – which, as demonstrated in Section 4.6.3, has been verified to be correctly integrated into the Spaun 2.0 model. Table 5.1 shows the percentage matches collected for 1,157 matching pairs of stimuli, and 404 non-matching pairs of stimuli, distributed roughly evenly across all 10 possible digit classes. For this task (roughly 100 trials for each matching pair, and 10 trials for each combination of non-matching pairs), the handwritten MNIST digits are presented to introduce the possibility that Spaun will incorrectly classify the digit's class (Spaun's control characters have an almost 100% classification accuracy).

From the data shown in Table 5.1, it can be seen that the average in-class match accuracy of 95.61% is in line with the classification accuracy of the digit recognition task (from Section 4.6.3.2), accounting for the fact that for a match to be determined, both stimuli have to be classified correctly (i.e., probability of correctly classifying one digit: 97.76%, probability of correctly classifying both digits: 97.76% $\times$ 97.76% = 95.57%). Additionally, using the MNIST dataset, only 1 false positive was reported across all of the collected data (mismatch between the 5 and 8 digits, 1 trial out of 9). The data thus demonstrates that the stimulus matching task has been correctly implemented within the Spaun 2.0 model.

ImageNet Stimulus-match Task: The previous sections demonstrated the effective implementation of Spaun 2.0's ImageNet visual hierarchy, as well as the stimulus matching task. Here, the integration of the ImageNet visual hierarchy with the stimulus matching task is discussed.

To begin, Figure 5.11 shows the recorded inputs and outputs of three instances where Spaun performed the ImageNet stimulus matching task correctly (showing both the true positive, and true negative cases). Next, Figure 5.12 demonstrates two instances where Spaun performs the ImageNet stimulus matching task incorrectly, in one instance, failing to identify a match between two images belonging to the same class (false negative), and in the other instance, incorrectly identifying two images from different classes as a match (false positive).

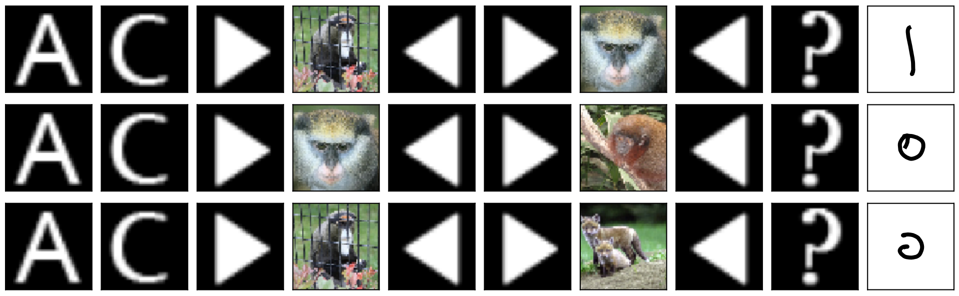


Figure 5.11: Recorded input stimuli and Spaun outputs for the ImageNet stimulus matching task. In this figure, Spaun responds with correct answers to the matching task. (Top) Here, the input stimuli are both guenons, and Spaun responds correctly with a "1" indicating it has identified images from the same class. (Middle) Here, the first input stimulus is a guenon (monkey), while the second stimulus is a titi (monkey). While both stimuli are monkeys, Spaun correctly identifies that they belong to different ImageNet classes, and responds with a "0" indicating so. (Bottom) Here, the first input stimulus is a guenon, and the second stimulus is a kit fox. Spaun correctly identifies that the images do not belong to the same image class, and responds accordingly.

In addition to the demonstration of Spaun performing the ImageNet stimulus matching task, overall performance data was also collected. To generate the overall performance data, the top 5 most populous images classes were identified within the set of test images. These were the: box turtle, sewing machine, guenon (monkey), Tibetan Terrier, and the Persian cat. Spaun was then instructed to perform the stimulus matching task using every combination of these five image classes. Table 5.2 presents the accuracy results with the ImageNet visual system configured to

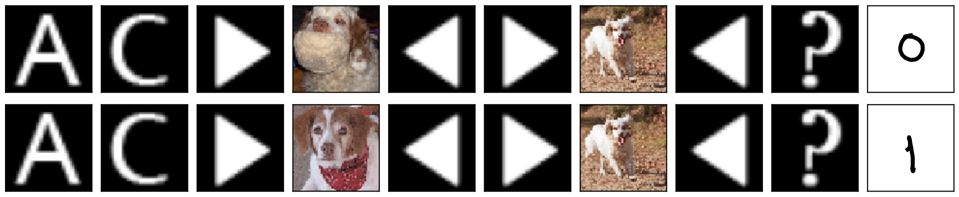


Figure 5.12: Recorded input stimuli and Spaun outputs for the ImageNet stimulus matching task demonstrating Spaun providing incorrect responses to the task. (Top) Here, both input stimuli are Clumber Spaniels, meaning Spaun should ideally respond with a "1". However, Spaun misidentifies one of the images as a different class, and thus responds to the task with a "0", indicating that it considers both images to belong to different ImageNet classes (i.e., a false negative). (Bottom) Here, the first stimulus is of a Brittany Spaniel, and the second image is of a Clumber Spaniel (identical to the image used previously). For these two stimuli, Spaun misidentifies the second image as a Brittany Spaniel (this can be deduced from the results in the top plot) and responds with a "1", as it believes both images belong to the same class (i.e., a false positive).

perform the "Top-1" classification criteria, while Table 5.3 presents the accuracy results with the visual system configured to perform the "Top-N" classification criteria. Each table presents:

- The raw classification accuracy as a ratio of the number of correctly classified images versus the total number of images in that class.
- The percentage of correctly identified matches for images belonging to the same class.
- The percentage of falsely identified matches (false positives) for images belonging to different classes, with one percentage accuracy value for each combination of image classes.

For each accuracy number, the Spaun model was run with the stimulus comparison task for an average of 80 trials.

From Table 5.2, it can be seen that Spaun has a 45.6% average task performance accuracy on the stimulus matching task using the "Top-1" classification criteria. This follows the trend of the raw classification accuracy. It is important to note that the chance accuracy for the raw image classification is $1/10^3$. However, for the stimulus matching task to be successful, Spaun has to correctly classify both stimuli, which has a chance success rate of $1/10^6$. From the data, it is also observed that across the 5 image classes, Spaun has an average 0.81% false positive rate for non-matching stimuli.

From Table 5.3, it is observed that using the "Top-N" classification criteria, the raw classification accuracies for each image class is increased. Additionally, Spaun's average overall performance on the stimulus matching task increases to 64.40%, with improvements across all but

ImageNet (Top-1)	Box Turtle	Sewing Machine	Guenon	Tibetan Terrier	Persian Cat
Raw classification accuracy	10/13 (76.92%)	4/13 (30.77%)	7/12 (58.33%)	8/11 (72.72%)	6/11 (54.54%)
Matched against:					
Box Turtle	54.00%	0.00%	0.00%	0.00%	1.00%
Sewing Machine		33.00%	0.00%	1.33%	0.00%
Guenon			52.00%	3.53%	0.00%
Tibetan Terrier				43.00%	2.22%
Persian Cat					46.00%

Table 5.2: Stimulus matching task performance accuracy results using the top 5 most populous image categories from the ImageNet test set. The table presents the results of the visual system configured to output the "Top-1" image class. The diagonals indicate correct performance, while the off-diagonals indicate false positives. See accompanying text for an analysis of the results.

ImageNet (Top-N)	Box Turtle	Sewing Machine	Guenon	Tibetan Terrier	Persian Cat
Raw classification accuracy	12/13 (92.31%)	5/13 (38.46%)	9/12 (75.00%)	10/11 (90.90%)	9/11 (81.81%)
Matched against:					
Box Turtle	78.00%	1.00%	3.08%	8.00%	10.53%
Sewing Machine		25.00%	3.85%	6.003%	5.26%
Guenon			65.00%	2.31%	6.40%
Tibetan Terrier				81.00%	8.42%
Persian Cat					73.00%

Table 5.3: Stimulus matching task performance accuracy results using the top 5 most populous image categories from the ImageNet test set used. The table presents the results of the visual system configured to output the "Top-N" image classes. The diagonals indicate correct performance, while the off-diagonals indicate false positives. See accompanying text for an analysis of the results.

one image classes. A decrease in accuracy is observed for the "sewing machine" class. However, the reported accuracy of 25% is still above the expected accuracy of 14.79% (classification accuracy for one "sewing machine" image is 38.46%, so the expected accuracy of identifying a match is 38.46% $\times$ 38.46% = 14.79%). As was the case with the "Top-1" configuration, the average matching-class accuracy follows the trend of the raw classification accuracy. However, the data also indicates that using the "Top-N" classification criteria leads to an increase in the false positive rate, to 5.49%.

The results from Table 5.2 and 5.3 demonstrate that the ImageNet visual system and the new stimulus matching task can be integrated into the Spaun 2.0 model simultaneously. Additionally, it is observed that the use of the ImageNet visual system for the stimulus matching task does not negatively impact the classification performance of the ImageNet visual hierarchy, showing that Spaun's visual system performance is independent of the task used (and vice versa). Finally, the demonstrated ability for Spaun to perform the stimulus matching task with either the MNIST visual system or the ImageNet visual system illustrates the modular capabilities of the Spaun 2.0 model.

5.3 Generalized Instruction Processing

The previous sections demonstrate that because of Spaun 2.0's updated modular design, both the motor system and the visual system can be interchanged with alternative implementations – as long as their input and output interfaces remain the same. While left for future work, this modularity can be demonstrated for the other Spaun 2.0 modules as well, allowing additional functionality to be introduced into the Spaun 2.0 model.

Despite the improved modularity of the Spaun 2.0 model, one aspect of Spaun remains invariant, and that is the set of tasks that Spaun is able to accomplish. As demonstrated in the previous sections, adding new tasks to Spaun requires, at minimum, making changes to the action selection hierarchy (particularly layer 1).

In 2013, Choo and Eliasmith demonstrated a proof-of-concept implementation of a modified cortico-basal ganglia-thalamic loop that was capable of processing and executing customized non-static condition-consequence pairs (i.e., the pairs could change during the course of the simulation run). This section explores the integration of this proof-of-concept network (hereafter referred to as the "instruction processing system") into the Spaun architecture, so that it may eventually serve as the foundation for a more flexible action selection hierarchy, thus removing Spaun's restriction to its eight original tasks.

Because the integration of the instruction processing system into Spaun affects its core functionality (and thus introduces complex behaviour), this section is divided into 6 sub-sections. First the general architecture of the instruction processing system is described (see Section 5.3.1). This is followed by one section (Sections 5.3.2 through 5.3.6) for each of the 5 stages in the implementation of the instruction processing system within the Spaun architecture, with each stage expanding upon the discussion of the previous sections and adding more complexity to Spaun.

The 5 stages begin with the implementation of the non-instructed stimulus-response task, which serves as a benchmark for the second stage. The second stage is the implementation of the instructed stimulus response task, which serves the demonstrate, for the first time, how the instruction processing system can be used to allow Spaun to perform a task where the task parameters are defined *after* the model has been constructed. The third and fourth stages describe how the instruction processing system can be applied to tasks more complex than just stimulus-response mappings, allowing it to affect the representations stored in memory (stage 3) and the representations within the action selection hierarchy (stage 4). Finally, the last stage demonstrates how the processing of sequential instructions can be implemented within the Spaun 2.0 architecture. This section concludes with a discussion of the instruction processing system as a whole.

5.3.1 The Instruction Processing System

The core of Spaun's action selection system is the set of condition-consequence pairs that dictate the logic behind how each task proceeds. Unfortunately, because the condition-consequence pairs are implemented as matrix transforms on the inputs to the basal ganglia network and the outputs of the thalamus network, they are essentially static after the model is built (with the exception of slow changes that can be made using the error signals and learning rules). This in turn implies that the set of tasks, and the processing steps within each task, are unchangeable after the Spaun model is built.

The primary idea of the instruction processing system is to have a method whereby the condition-consequence information is represented by neural activity (instead of a weight matrix), thus allowing the information to be modified after the model is constructed. Assuming that the information extracted from the decoding of the neural activity can be used to effect changes in the rest of the action selection hierarchy (methods for achieving this are discussed in later sections), the ability to change the neural representation of the condition-consequence pairs implies that logical processing steps of tasks can be altered, allowing Spaun to perform tasks it could not have performed before. Figure 5.13 compares the high-level architecture of the existing structure of the action selection hierarchy with one that includes the instruction processing system.

In the discussion that follows, the explanation of how the instruction processing system works is divided into two parts. First is an explanation of the schema used to encode each condition-consequences pair, as well as the schema used to encode collections of condition-consequence

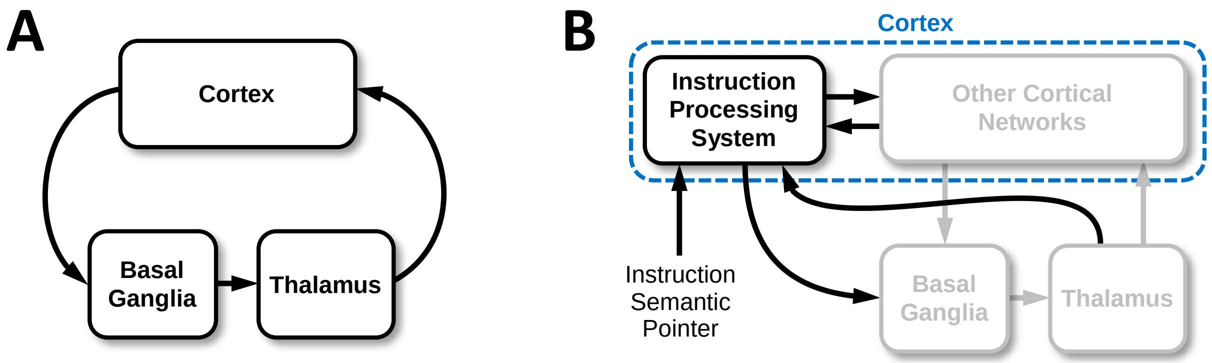


Figure 5.13: Comparison of schematic overviews of Spaun 2.0's current action selection hierarchy, and the instruction processing action selection hierarchy. (**A**) Simplified schematic of the current action selection hierarchy. As described in Section 3.3.8, state information in cortex affects the outputs of the basal ganglia and thalamic networks, resulting in changes in the representations stored in cortex. (**B**) Simplified schematic of the instruction processing action selection hierarchy, showing the additional mechanism (the cortical-instruction processing loop) by which cortical state information is combined with an instruction semantic pointer to effect changes in the representations stored in cortex, as well as the behaviour of the rest of the action selection hierarchy. Note that the existing action selection hierarchy networks and projections have been grayed out to better contrast the two network architectures.

pairs (referred to as an "instruction"). This is followed by an elaboration of how the encoding schema is procedurally reversed such that when the instruction processing system is provided with appropriate information, it can identify the consequence(s) that best match the state of the Spaun network, similar in functionality to the purpose of Spaun's action selection hierarchy.

Encoding Instructions: Spaun's implementation of the instruction processing system borrows concepts from the instruction processing network in [Choo and Eliasmith, 2013]. To facilitate the explanation of how Spaun's condition-consequence pairs can be encoded in neural activity, the following condition-consequence pair is used for the examples to follow:

$$\mathit{vision} \bullet \textsf{ZERO} \longmapsto \textsf{POS1} \circledast \textsf{ONE} \Rightarrow \mathit{motor} \tag{5.5}$$

To encode the conditionals and consequences, Spaun's instruction processing system uses bound pairs, with special "tag" semantic pointers to indicate the specific cortical area associated with the conditional or consequence. Thus, with the example conditional-consequence pair (see Eq. (5.5)), Spaun's instruction processing system would encode the conditional as:

$$\textsf{\textbf{CONDITION}} = \textsf{\textbf{VISION}} \circledast \textsf{\textbf{ZERO}}, \tag{5.6}$$

where "**VISION**" is the semantic pointer "tag" associated with the *vision* system. Likewise, the consequence of Eq. 5.5 would be encoded as:

$$\textsf{\textbf{CONSEQUENCE}} = \textsf{\textbf{MOTOR}} \circledast \textsf{\textbf{POS1}} \circledast \textsf{\textbf{ONE}}, \tag{5.7}$$

where "**MOTOR**" is the semantic pointer associated with the *motor* system.

To construct the representation of the entire condition-consequence pair, the collections operator is used with permuted versions of the condition and consequence representations, with one permutation matrix used to mark the conditional ($\mathbf{P}_{ant}$)[44] and a different permutation matrix used to mark the consequence ($\mathbf{P}_{cons}$). The reason for the use of the permutations will become obvious momentarily. Thus, the entire condition-consequence pair from Eq. (5.5) is encoded as:

$$\begin{aligned} \textsf{\textbf{CC\_PAIR}} &= (\textsf{\textbf{CONDITION}})\mathbf{P}_{ant} + (\textsf{\textbf{CONSEQUENCE}})\mathbf{P}_{cons} \\ &= (\textsf{\textbf{VISION}} \circledast \textsf{\textbf{ZERO}})\mathbf{P}_{ant} + (\textsf{\textbf{MOTOR}} \circledast \textsf{\textbf{POS1}} \circledast \textsf{\textbf{ONE}})\mathbf{P}_{cons} \end{aligned}$$

Instructions contain multiple condition-consequence pairs, and are constructed by are using the list representation used in Spaun's memory system. As an example, an instruction containing three condition-consequence pairs is represented as:

$$\textsf{\textbf{INSTRUCTION}} = \textsf{\textbf{POS1}} \circledast \textsf{\textbf{CC\_PAIR1}} + \textsf{\textbf{POS2}} \circledast \textsf{\textbf{CC\_PAIR2}} + \textsf{\textbf{POS3}} \circledast \textsf{\textbf{CC\_PAIR3}}$$

There are several important notes about the use of the list representation for the instruction. First, although the use of the list representation implicitly imparts an order to the condition-consequence pairs within the instruction, the order information is typically not a necessary property to the instruction as a whole (similar to how the condition-consequence pairs in the action selection hierarchy are unordered). Rather, the list representation is used because it provides a convenient method to "tag" each condition-consequence pair within the instruction. Second, the list representation is the reason for the use of the permutation matrices to mark the conditions and consequences within the instruction set.

As an example, by using "**ANT**" and "**CONS**" tags instead of the permutation matrices, the

[44]Here, the abbreviation "ant" is used to denote the condition (antecedent) of the condition-consequence pair. This is done to minimize possible confusion between the abbreviations for "conditional" ("cond") and "consequence" ("cons").

following two-condition-consequence-pair instruction:

$$\begin{aligned} vision \bullet \textsf{\textbf{ZERO}} &\longmapsto \textsf{\textbf{POS1}} \circledast \textsf{\textbf{ONE}} \Rightarrow motor \\ vision \bullet \textsf{\textbf{TWO}} &\longmapsto \textsf{\textbf{POS1}} \circledast \textsf{\textbf{TWO}} \Rightarrow motor, \end{aligned}$$

is represented as the following semantic pointer in the instruction processing system:

$$\begin{aligned} \textsf{\textbf{INSTRUCTION}} = {} & \textsf{\textbf{POS1}} \circledast ((\textsf{\textbf{ANT}} \circledast \textsf{\textbf{VISION}} \circledast \textsf{\textbf{ZERO}}) + (\textsf{\textbf{CONS}} \circledast \textsf{\textbf{MOTOR}} \circledast \textsf{\textbf{POS1}} \circledast \textsf{\textbf{ONE}})) + \\ & \textsf{\textbf{POS2}} \circledast ((\textsf{\textbf{ANT}} \circledast \textsf{\textbf{VISION}} \circledast \textsf{\textbf{TWO}}) + (\textsf{\textbf{CONS}} \circledast \textsf{\textbf{MOTOR}} \circledast \textsf{\textbf{POS1}} \circledast \textsf{\textbf{TWO}})) \end{aligned} \tag{5.8}$$

However, because the SPA binding operation is commutative (see Section 2.3.8), the semantic pointer representation of Eq. 5.8 is equivalent to:

$$\begin{aligned} \textsf{\textbf{INSTRUCTION}} = {} & \textsf{\textbf{POS1}} \circledast ((\textsf{\textbf{ANT}} \circledast \textsf{\textbf{VISION}} \circledast \textsf{\textbf{ZERO}}) + (\textsf{\textbf{CONS}} \circledast \textsf{\textbf{MOTOR}} \circledast \textsf{\textbf{POS1}} \circledast \textsf{\textbf{ONE}})) + \\ & \textsf{\textbf{POS1}} \circledast (\textsf{\textbf{CONS}} \circledast \textsf{\textbf{MOTOR}} \circledast \textsf{\textbf{POS2}} \circledast \textsf{\textbf{TWO}}) + \\ & \textsf{\textbf{POS2}} \circledast (\textsf{\textbf{ANT}} \circledast \textsf{\textbf{VISION}} \circledast \textsf{\textbf{TWO}}), \end{aligned} \tag{5.9}$$

which can be translated back into the following set of condition-consequence pairs:

$$\begin{aligned} vision \bullet \textsf{\textbf{ZERO}} &\longmapsto \textsf{\textbf{POS1}} \circledast \textsf{\textbf{ONE}} \Rightarrow motor, \textsf{\textbf{POS2}} \circledast \textsf{\textbf{TWO}} \Rightarrow motor \\ vision \bullet \textsf{\textbf{TWO}} &\longmapsto \emptyset \end{aligned}$$

Since the goal of the instruction schema is to have a unique mapping between the set of condition-consequence pairs and its SPA representation, the permutation matrices are used to enforce non-commutativity of the binding operation, thus avoiding the issues demonstrated by the example above.

Decoding Instructions: The primary purpose of the instruction processing system is to use information about the state of the Spaun network, along with the provided instruction semantic pointer representation (which contains information on the non-static condition-consequence pairs) to extract appropriate non-static consequence information which can then be used to augment the behaviour of the action selection hierarchy. In order to do this, the schema used to encode the instruction semantic pointer must essentially be reversed.

Extracting information from the instruction semantic pointer is a two step process, similar to the process outlined by Eq. (2.11). First, the information gathered about the state of the Spaun network is used to construct a "probe" semantic pointer. This "probe" is then used to identify which condition of each condition-consequence pair within the instruction best matches the given state of the system. Once the closest matching condition-consequence pair has been identified, that information can then be used to extract information about the consequence of that pair (which is then propagated to the rest of the Spaun network).

Using the following instruction semantic pointer as an example,

$$\begin{aligned}\textsf{\textbf{INSTRUCTION}} = \textsf{\textbf{POS1}} \circledast ((\textsf{\textbf{VISION}} \circledast \textsf{\textbf{ZERO}})\mathbf{P}_{ant} + (\textsf{\textbf{MOTOR}} \circledast \textsf{\textbf{POS1}} \circledast \textsf{\textbf{ONE}})\mathbf{P}_{cons}) +\\ \textsf{\textbf{POS2}} \circledast ((\textsf{\textbf{VISION}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{ant} + (\textsf{\textbf{MOTOR}} \circledast \textsf{\textbf{POS1}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{cons})\end{aligned} \tag{5.10}$$

the information extraction process can be demonstrated algebraically. The first step to the information decoding process is to tag information with the semantic pointer representations of their respective sources. Additionally, to indicate that the incoming state representations are to be used as the conditions for each condition-consequence pair within the instruction, the "ant" permutation is applied. As an example, if Spaun is visually presented with a "2", the "condition" probe semantic pointer (**PROBE_COND**) internal to the instruction processing system is constructed as:

$$\textsf{\textbf{PROBE\_COND}} = (\textsf{\textbf{VISION}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{ant} \tag{5.11}$$

Next, using this probe semantic pointer, the most appropriate condition-consequence pair is identified. This is achieved by binding the instruction semantic pointer with the inverse of the probe representation, resulting in a collection of list position semantic pointers. The list position semantic pointer which has the highest contribution in the result indicates the position of the most appropriate condition-consequence pair within the instruction. With the probe representation given by Eq. (5.11), and the instruction semantic pointer given by Eq. (5.10), the list position (**LIST_POS**) of the best matching condition-consequence pair is determined by:

$$\begin{aligned}\textsf{\textbf{LIST\_POS}} &= \textsf{\textbf{INSTRUCTION}} \circledast {\sim}\textsf{\textbf{PROBE\_COND}}\\ &= [\textsf{\textbf{POS1}} \circledast ((\textsf{\textbf{VISION}} \circledast \textsf{\textbf{ZERO}})\mathbf{P}_{ant} + (\textsf{\textbf{MOTOR}} \circledast \textsf{\textbf{POS1}} \circledast \textsf{\textbf{ONE}})\mathbf{P}_{cons}) +\\ &\quad \textsf{\textbf{POS2}} \circledast ((\textsf{\textbf{VISION}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{ant} + (\textsf{\textbf{MOTOR}} \circledast \textsf{\textbf{POS1}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{cons})] \circledast\\ &\quad {\sim}(\textsf{\textbf{VISION}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{ant}\\ &= [{\sim}(\textsf{\textbf{VISION}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{ant} \circledast \textsf{\textbf{POS1}} \circledast (\ldots) +\\ &\quad {\sim}(\textsf{\textbf{VISION}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{ant} \circledast \textsf{\textbf{POS2}} \circledast (\textsf{\textbf{VISION}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{ant} +\\ &\quad {\sim}(\textsf{\textbf{VISION}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{ant} \circledast \textsf{\textbf{POS2}} \circledast (\textsf{\textbf{MOTOR}} \circledast \textsf{\textbf{POS1}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{cons}]\\ &= [\ldots + \textsf{\textbf{POS2}} + \ldots]\end{aligned}$$

Once the list position representation is computed, it is fed through a cleanup memory to remove extraneous semantic pointer terms introduced by the binding operation:

$$\begin{aligned}\textsf{\textbf{LIST\_POS}}' &= \textit{cleanup}\{\textsf{\textbf{LIST\_POS}}\}\\ &= \textsf{\textbf{POS2}}\end{aligned}$$

Binding the instruction semantic pointer with the inverse of the cleaned list position semantic pointer results in the semantic pointer representation of the condition-consequence pair (**PROBED_CC_PAIR**) occupying the specified list position:

$$\begin{aligned}\textsf{\textbf{PROBED\_CC\_PAIR}} &= \textsf{\textbf{INSTRUCTION}} \circledast \sim\textsf{\textbf{LIST\_POS}}' \\ &= \textsf{\textbf{INSTRUCTION}} \circledast \sim\textsf{\textbf{POS2}} \\ &= [\textsf{\textbf{POS1}} \circledast \sim\textsf{\textbf{POS2}} \circledast (\ldots) + \\ &\quad\ \textsf{\textbf{POS2}} \circledast \sim\textsf{\textbf{POS2}} \circledast ((\textsf{\textbf{VISION}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{ant} + \\ &\qquad\qquad (\textsf{\textbf{MOTOR}} \circledast \textsf{\textbf{POS1}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{cons})] \\ &= [\ldots + (\textsf{\textbf{VISION}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{ant} + (\textsf{\textbf{MOTOR}} \circledast \textsf{\textbf{POS1}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{cons}]\end{aligned} \tag{5.12}$$

Using the probed condition-consequence pair, the desired semantic pointer representation of the consequence (**PROBED_CONS**) can be extracted by applying an inverse "cons" permutation matrix. From Section 2.3.8, the inverse "cons" permutation matrix is simply its matrix transpose ($\mathbf{P}^T_{cons}$):

$$\begin{aligned}\textsf{\textbf{PROBED\_CONS}} &= (\textsf{\textbf{PROBED\_CC\_PAIR}})\mathbf{P}^T_{cons} \\ &= [\ldots + (\textsf{\textbf{VISION}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{ant}\mathbf{P}^T_{cons} + (\textsf{\textbf{MOTOR}} \circledast \textsf{\textbf{POS1}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{cons}\mathbf{P}^T_{cons}] \\ &= [\ldots + (\textsf{\textbf{MOTOR}} \circledast \textsf{\textbf{POS1}} \circledast \textsf{\textbf{TWO}})]\end{aligned}$$

Finally, information (**DATA**) can be extracted from the consequence representation by binding it with the inverse of the tags representing each downstream module (i.e., each module that the conditions within the instruction target):

$$\begin{aligned}\textsf{\textbf{DATA}} &= \textsf{\textbf{PROBED\_CONS}} \circledast \sim\textsf{\textbf{MOTOR}} \\ &= [\ldots + (\textsf{\textbf{MOTOR}} \circledast \textsf{\textbf{POS1}} \circledast \textsf{\textbf{TWO}}) \circledast \sim\textsf{\textbf{MOTOR}}] \\ &\approx \textsf{\textbf{POS1}} \circledast \textsf{\textbf{TWO}}\end{aligned}$$

To summarize, the steps required to extract the semantic pointer representation of the consequence that best matches the state of the Spaun network are:

1. Tag the representations projected from the various Spaun modules with the semantic pointer

tags representing each module.

2. Apply the "ant" permutation matrix to the collection of all the tagged representations, creating the **PROBE_COND** representation.
3. Bind the instruction semantic pointer with the inverse of the probe semantic pointer to get the representation of the best matching list positions (**LIST_POS**).
4. Apply the cleanup operation to the list position representation (**LIST_POS**) to isolate only the top matching list position (**LIST_POS′**).
5. Bind the instruction semantic pointer with the inverse of the top matching list position semantic pointer (**LIST_POS′**) to get the best matching condition-consequence pair (**PROBED_CC_PAIR**).
6. Apply the inverse "cons" permutation matrix to the best matching condition-consequence pair (**PROBED_CC_PAIR**) to undo the "cons" permutation matrix, resulting in the consequence representation **PROBED_CONS**.
7. For each module that the conditions target, extract the relevant information by binding the probed consequence representation (**PROBED_CONS**) with the inverse of the semantic pointer tags representing each target module.

Figure 5.14 illustrates the schematic of the instruction decoding network, using the steps above to extract the relevant condition information from a given instruction semantic pointer.

5.3.2 Stage 1: The Stimulus-response Task

Stages 1 and 2 of integrating the instruction processing task into the Spaun 2.0 model involve the demonstration of non-instructed (Stage 1) and instructed (Stage 2) versions of a simple task – the stimulus-response task. For stage 1 of the integration process, the conditions and expected actions of the stimulus response task are hard-wired into the Spaun network (i.e., changing the task parameters require the Spaun network to be re-built), while in stage 2 of integration process involves adapting this task to use the instruction processing system, thus allowing the conditions and expected actions of the stimulus response task to be changed while the Spaun simulation is running.

Task Description: The stimulus-response task is straightforward. When Spaun is presented with a stimulus, it is expected to respond with a digit. Both the trigger stimulus and the expected digit response are predetermined before the Spaun simulation commences (i.e., they are pre-set as part of the model construction process).

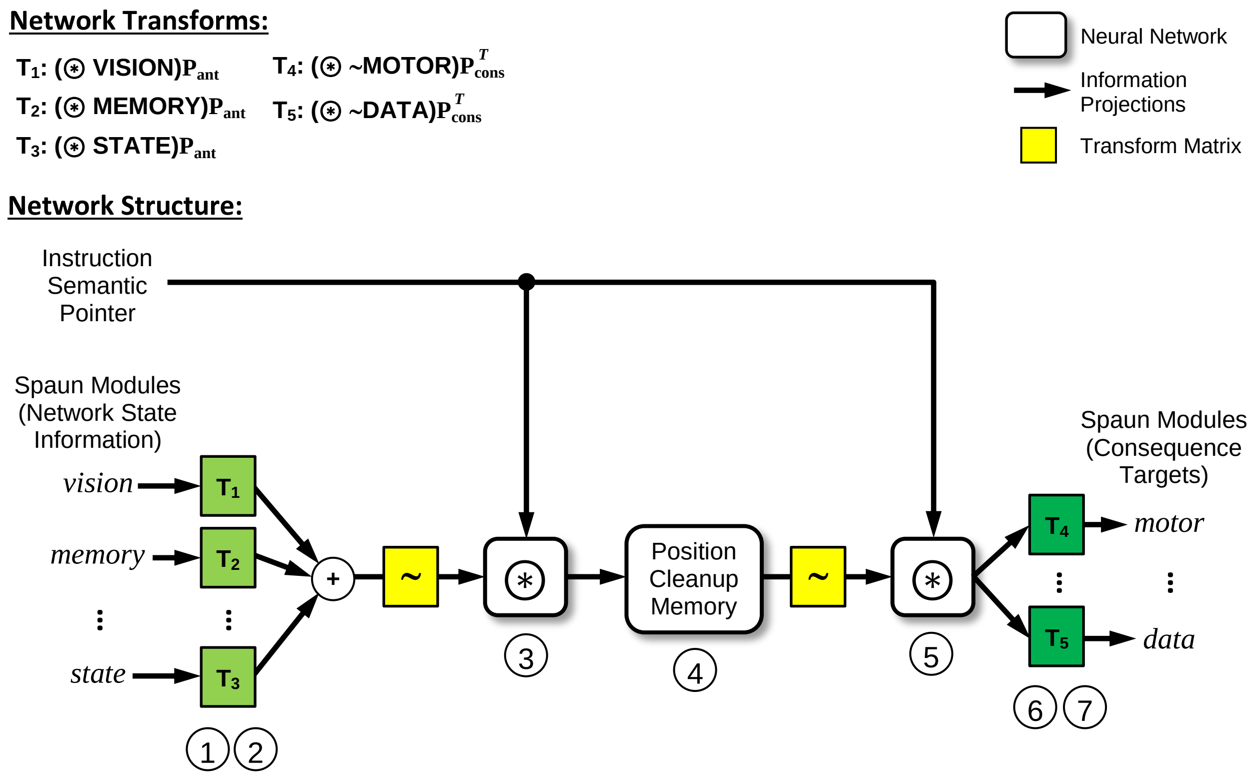


Figure 5.14: Network schematic of the instruction processing system detailing how the decoding steps (steps ① through ⑦) outlined in the accompanying text is achieved. There are several important details to note about the network implementation of the instruction decoding process. First, because the "tagging" of the Spaun network state information (①) is implemented using the SPA binding operator with an unchanging "tag" semantic pointer, this can be implemented in the network as a matrix transform. This applies to the "un-tagging" process (⑦) as well. Additionally, because both the binding and permutation operations are linear, they can be combined into one matrix (see "Network Transforms"). This applies to both the application of the "ant" permutation (②), and the removal of the "cons" permutation (⑥).

Task Syntax: The input character sequence for the stimulus-response task follows the form:

$$\textbf{A 8 ?}\ x\ r\ \ldots,$$

where the character sequence "**A8**" indicates that Spaun is to perform the stimulus-response task, x is the stimuli provided to Spaun (may not necessarily be the trigger stimulus), and r is the response Spaun provides to the provided stimulus image.

Conceptual Implementation: Conceptually, this task is physical manifestation of the condition-consequence pairs, as they both share the same "if-then" reasoning. As an example, is Spaun were needed to respond with a "1" upon the presentation of the "8" stimulus image, this can be implemented using the condition-consequence pair:

$$vision \bullet \textsf{\textbf{EIGHT}} \longmapsto \textsf{\textbf{POS1}} \circledast \textsf{\textbf{ONE}} \Rightarrow motor$$

System Requirements: For this task, Spaun is required to have no additional components, as it already contains the action selection hierarchy (which is used to implement the condition-consequence pairs).

Functional Implementation: While the stimulus-response task can be (conceptually) implemented using one condition-consequence pair per stimulus-response pair, the actual functional Spaun 2.0 implementation requires additional supporting infrastructure. First, as with the addition of the stimulus matching task, an additional task representation (**RESPONSE**) is added.

Next, in order to configure the Spaun network to correctly handle incoming information, the appropriate task stage representation needs to be determined. To generate a response for the stimulus-response task, information needs to be routed to the motor system. While a new task stage and supporting flow control networks can be added to achieve this, the Spaun 2.0 implementation takes advantage of the existing projections (and selector networks) from the transformation network to the decoding system (which already contain the necessary logic to convert the internal representations into the appropriate motor semantic pointers). Thus, the output of the thalamus is projected to the transformation network, and a new task stage (**DIRECT**) is added to configure the transformation network to route this thalamic projection to the output of the transformation system.

With this network configuration, the condition-consequence pairs to initialize the stimulus-response task is then:

$$0.5 \times (task \bullet \textsf{\textbf{INIT}} + vision \bullet \textsf{\textbf{EIGHT}}) \longmapsto \textsf{\textbf{RESPONSE}} \Rightarrow task, \textsf{\textbf{DIRECT}} \Rightarrow task\_stage$$

The condition-consequence pairs for each stimulus-response mapping is then of the form:

$$0.5 \times (task \bullet \textsf{RESPONSE} + vision \bullet \mathbf{x}) \longmapsto \mathbf{y} \Rightarrow transform,$$

where $\mathbf{x}$ is the semantic pointer representation of the trigger stimulus, and $\mathbf{y}$ is the semantic pointer representation of the expected response. As an example, using the stimulus-response mapping provided above (see "8", write "1"), the corresponding Spaun 2.0 condition-consequence pair would be:

$$0.5 \times (task \bullet \textsf{RESPONSE} + vision \bullet \textsf{EIGHT}) \longmapsto \textsf{POS1} \circledast \textsf{ONE} \Rightarrow transform$$

Figure 5.15 illustrates the changes necessary to the transformation system needed to implement the stimulus-response task.

5.3.2.1 Results

To demonstrate Spaun's efficacy at the stimulus-response task, two metrics were collected. For both metrics, the action selection hierarchy of the Spaun model was configured to respond with a "3" when shown a "1", and respond with a "4" when shown a "2". Figure 5.16 shows the stimulus inputs and corresponding Spaun outputs for the task, demonstrating that Spaun is capable of performing the task. Over 390 trials, Spaun reported a task performance accuracy of 98.97%.

Additionally, response timing data were collected for the task. For the 390 trials, Spaun took 290.55ms (95% confidence interval of 278.59ms to 303.16ms) to successfully complete the task. This data compares well to human response times measured for this task, which is reported as 274 ± 43ms in [Grice et al., 1982]. Figure 5.17 compares the mean response times for this task for the instruction processing model from [Choo and Eliasmith, 2013], human data from [Grice et al., 1982], and the Spaun 2.0 model. It should be noted that two values are reported for the original instruction processing model – the raw response times reported for the model, and the response times of the model adjusted to include average visual processing times (reported in [Anderson and Lebiere, 2014] as 85ms) and average motor processing times (reported in [Meyer and Kieras, 1997] as 150ms). This was done as the original instruction processing model did not include a visual network implementation nor a motor network implementation.

5.3.3 Stage 2: The Instructed Stimulus-response Task

Stage 2 of the integration process involves the implementation of the instructed stimulus-response task. At the high level, this task is behaviourally identical to the stimulus-response task from

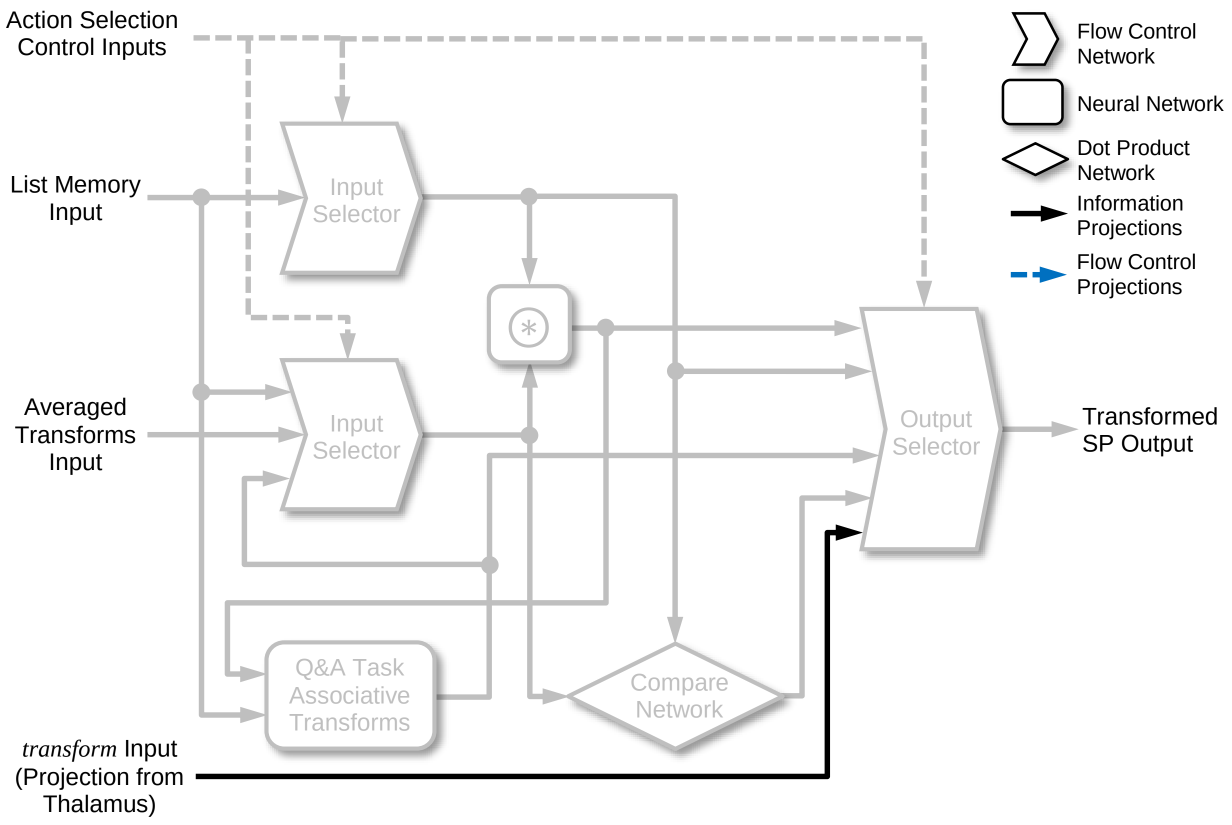


Figure 5.15: Schematic of Spaun 2.0's transformation system (see Figure 3.26) updated to support the stimulus-response task. Existing networks and projections from the Spaun 2.0 implementation have been grayed out to highlight the changes made to the network architecture.

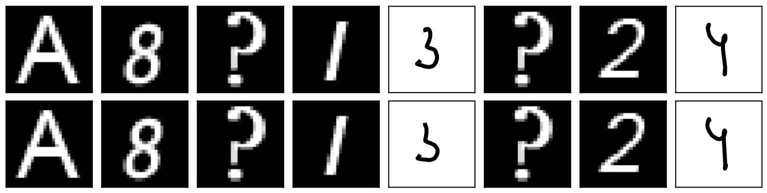


Figure 5.16: Recorded input stimuli and Spaun outputs for the stimulus-response task. Here, two iterations of the task (for a total of 4 trials) are shown. As the figure illustrates, because the stimulus-response mapping is "hard-coded" into the condition-consequence pairs of the action selection hierarchy, both iterations of the task produce the same stimulus-response mappings.

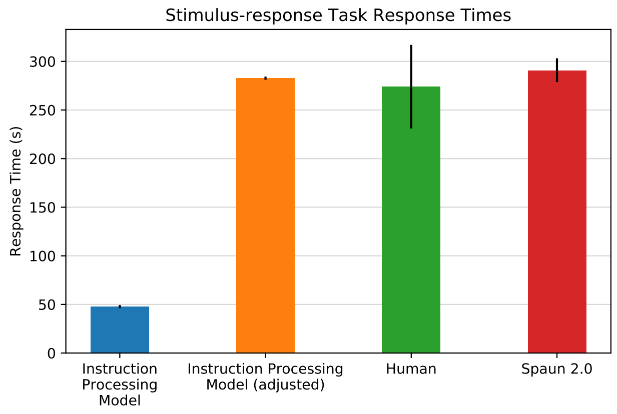


Figure 5.17: Comparison of response times for the stimulus-response task. Shown are data from the original instruction processing model [Choo and Eliasmith, 2013], human data [Grice et al., 1982], and the Spaun 2.0 model. Refer to accompanying text for additional details.

Stage 1. However, the unique property of the instructed stimulus-response task is that it is the first demonstration of how the instruction processing system can be used to modify the conditions and consequences of Spaun's task *during* the simulation run (i.e., after the Spaun model is created), without requiring the Spaun model to be reconstructed.

Task Description: The instructed stimulus-response task is identical its non-instructed counterpart. That is to say, the task defines a stimulus-response mapping, and when presented with the trigger stimulus, Spaun is expected to provide the appropriately mapped response. The difference between the two task lies in how the stimulus-response mapping is provided to Spaun. In Stage 1, it is presumed that Spaun has intrinsic knowledge of the desired stimulus-response mapping. For the instructed stimulus-response task however, Spaun is provided with the desired mapping (encoded as the instruction semantic pointer) as the task is being processed.

Task Syntax: The input character sequence for the stimulus matching task follows the form:

$$\mathbf{A\ 9\ ?}\ x\ r\ \ \ldots,$$

where the character sequence "**A9**" indicates that Spaun is to perform the instructed stimulus-response task, and as before, x is the stimuli provided to Spaun, and r is the response Spaun provides to the provided stimulus image.

Task Instruction Format: In the static stimulus-response task, the stimulus-response mappings are "encoded" in the condition-consequence pairs of the action selection hierarchy. As an example, if the desired stimulus-response mappings are:

- See "eight", write "one";
- See "two", write "three";

the static stimulus-response condition-consequence pairs (for Stage 1) would be:

$$0.5 \times (task \bullet \textsf{\textbf{RESPONSE}} + vision \bullet \textsf{\textbf{EIGHT}}) \longmapsto \textsf{\textbf{POS1}} \circledast \textsf{\textbf{ONE}} \Rightarrow transform$$
$$0.5 \times (task \bullet \textsf{\textbf{RESPONSE}} + vision \bullet \textsf{\textbf{TWO}}) \longmapsto \textsf{\textbf{POS1}} \circledast \textsf{\textbf{THREE}} \Rightarrow transform,$$

For the instructed stimulus-response task, the stimulus-response mappings are encoded in the instruction semantic pointer. The encoding schema used is identical to the schema introduced by Eq. (5.10), with the exception that the "payload" of the consequence statements are tagged with "**DATA**" instead of "**MOTOR**". While it is the case that the consequence statements for the stimulus-response task directly affects the motor system, the "**DATA**" tag is used increase the generalizability of the network used to implement this task (see Figure 5.18 and the Stage 3 integration process). With this change, the general encoding format of the instruction semantic pointer for this task is:

$$\textsf{\textbf{INSTRUCTION}} = \textsf{\textbf{POS}}N \circledast ((\textsf{\textbf{VISION}} \circledast \mathbf{x})\mathbf{P}_{ant} + (\textsf{\textbf{DATA}} \circledast \mathbf{y})\mathbf{P}_{cons}) + \ldots, \tag{5.13}$$

where $\mathbf{x}$ is the semantic pointer representation of the trigger stimuli, and $\mathbf{y}$ is the semantic pointer representation of the expected responses. Using this encoding format, the equivalent instruction semantic pointer for the example 2-set stimulus-response mapping described above would be:

$$\begin{aligned}\textsf{\textbf{INSTRUCTION}} = {} & \textsf{\textbf{POS1}} \circledast ((\textsf{\textbf{VISION}} \circledast \textsf{\textbf{EIGHT}})\mathbf{P}_{ant} + (\textsf{\textbf{DATA}} \circledast \textsf{\textbf{POS1}} \circledast \textsf{\textbf{ONE}})\mathbf{P}_{cons}) + \\ & \textsf{\textbf{POS2}} \circledast ((\textsf{\textbf{VISION}} \circledast \textsf{\textbf{TWO}})\mathbf{P}_{ant} + (\textsf{\textbf{DATA}} \circledast \textsf{\textbf{POS1}} \circledast \textsf{\textbf{THREE}})\mathbf{P}_{cons})\end{aligned}$$

Conceptual Implementation: While the static stimulus-response task is implemented through the construction of condition-consequence pairs, the implementation of the instructed stimulus-response task leverages the operation of the instruction processing system to achieve the same (condition-consequence) functionality. Section 5.3.1 demonstrates how the instruction processing system network can be used to extract consequence information from a given instruction semantic pointer. Thus, if the instruction processing system is provided with visual information, and with the appropriate instruction semantic pointer, all that is required is to ensure the extracted consequence information is properly routed to the motor system to achieve the same functionality that the condition-consequence pairs provide in Stage 1.

System Requirements: For this task, Spaun is required to have:

- The instruction processing system.

Functional Implementation: Much of the supporting framework for the instructed stimulus-response task has already been added through the implementation of its non-instructed counterpart. The only necessary change to the Spaun 2.0 network to support the instructed stimulus-response task is the addition of the instruction processing system (see Figure 5.14).

Adding the instructed stimulus-response task to Spaun follows the same basic steps as before. First, a new task representation ($\mathsf{\mathbf{INSTR\_RESP}}$) is added so that Spaun can keep track of the progress of the new task. Next, as with the static stimulus-response task, information needs to be routed to the motor system. However, as a task stage representation ($\mathsf{\mathbf{DIRECT}}$) already exists for that, it is re-used. This means that the first condition-consequence pair for this task can be written as:

$$0.5 \times (task \bullet \mathsf{\mathbf{INIT}} + vision \bullet \mathsf{\mathbf{NINE}}) \longmapsto \mathsf{\mathbf{INSTR\_RESP}} \Rightarrow task, \mathsf{\mathbf{DIRECT}} \Rightarrow task\_stage$$

The next condition-consequence pair for this task dictates what information is routed to the transformation system (and subsequently the motor system) when the trigger stimuli are presented. In the static stimulus-response task, the information to be routed was determined by the consequence set in the condition-consequence pair. For the instructed stimulus-response task, however, the flexibility of the instruction processing system is taken advantage of.

If an instruction semantic pointer encoded with the appropriate information (see Section 5.3.1) is projected into the instruction processing system, it can use information from the visual system to determine the appropriate task response. The condition-consequence pair for this task only ensures that the output of the instruction processing system is routed to the appropriate input of the transformation module. Referring to the instruction encoding format described above (see "Task Instruction Format"), the *data* output of the instruction processing system (*instr_data*) needs to be projected to the $transform$ input of the transformation module. This is accomplished using the following condition-consequence pair:

$$task \bullet \mathsf{\mathbf{INSTR\_RESP}} \longmapsto instr\_data \Rightarrow transform \tag{5.14}$$

It is important to note that in the case of the static stimulus-response task, one condition-consequence pair is needed for each desired stimulus-response mapping. However, because the instruction processing system handles the job of extracting the appropriate responses from the instruction semantic pointer, only one condition-consequence pair (Eq. (5.14)) is needed for the instructed stimulus-response task.

Figure 5.18 illustrates the networks and projections necessary to implement the instructed stimulus-response task. In the figure, the inputs and outputs of the instruction processing system (see Figure 5.13) and transformation system (modified to support the stimulus-response tasks in general – see Figure 5.15) have been included for clarity.

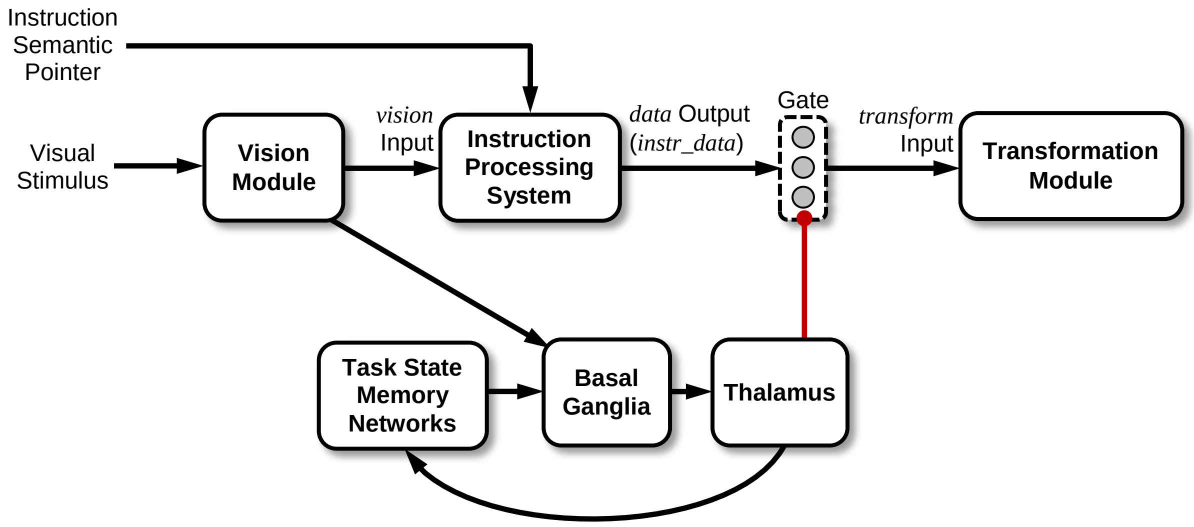

Instruction Semantic Pointer
Visual Stimulus
Vision Module
vision Input
Instruction Processing System
data Output (*instr_data*)
Gate
transform Input
Transformation Module
Task State Memory Networks
Basal Ganglia
Thalamus

Figure 5.18: Schematic diagram of the networks and projections necessary to implement the instructed stimulus-response task. Note that only the networks involved in the stimulus-response task are shown – the rest of Spaun's other networks are omitted to simplify the network diagram. Also note that the "gate" network used to control the flow of information between the output of the instruction processing system and the input of the transformation system is a result of the implementation of the dynamic consequence (see Section 3.1.4.3 and Figure 3.10) in Eq. (5.14).

5.3.3.1 Results

Similar to the stimulus-response task, two metrics were used to demonstrate Spaun's efficacy on the instructed stimulus-response task. For the recorded data, Spaun was provided with two instructions, distributed equally between the trials:

- See "1", write "8"; see "2", write "1";
- See "1", write "2"; see "2", write "8".

Figure 5.19 demonstrates Spaun performing the instructed stimulus-response task during one simulation run in which the instruction provided is changed halfway through the simulation. As the figure demonstrates, Spaun is able to perform the task correctly, without needing any permanent changes to the Spaun model (i.e., needing to re-construct the model). Over 320 trials, the Spaun model reported a mean task performance accuracy of 91.56%.

By comparing the task performance accuracy of this task to the non-instructed variant, it can be inferred that the addition of the instruction processing system decreased the performance accuracy by roughly 7.5%. This is probably due to the fact that the extraction of information from the instruction processing system introduces noise into the system, especially since a cleanup network is not used, thus decreasing the probability of successful recall from the decoding system, resulting in Spaun responding incorrectly (or not at all) to the provided stimulus.

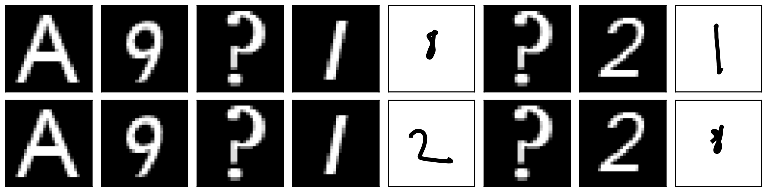


Figure 5.19: Recorded input stimuli and Spaun outputs for the instructed stimulus-response task. In the figure, two iterations of the task are shown. During the first iteration (top row) Spaun is provided with the instruction "see '1', write '8'; see '2', write '1"', and during the second task iteration (bottom row), Spaun is provided with the instruction "see '1', write '2'; see '2', write '8"'. It is observed that for both task iterations, Spaun is able to produce the correct responses, demonstrating that it is able to correctly modify its behaviour when the instruction changes. It is important to note that the data shown is collected from *one* simulation of the Spaun model, with no changes made to the model between the two task iterations.

As with the non-instructed stimulus-response task, response timing data were also collected. The Spaun model reported an average task response time of 373.31ms (95% confidence interval of 360.01ms to 383.63ms). This also compares well to the human response times on a similar task, reported in [Grice et al., 1982] to be 355 ± 58ms. Figure 5.20 compares the response time data collected using the original instruction processing model from [Choo and Eliasmith, 2013], human data reported in [Grice et al., 1982], and the Spaun 2.0 model.

In addition to the "raw" response times, Figure 5.20 also compares the response time differences between the instructed and non-instructed variants of the stimulus-response task. As the graphs show, for the Spaun 2.0 model, the extra processing time introduced by instruction processing system (81.76ms) matches the difference in response times reported for the human data (81ms), and is a more favourable match when compared to the response time difference reported for the original instruction processing model (73.2ms).

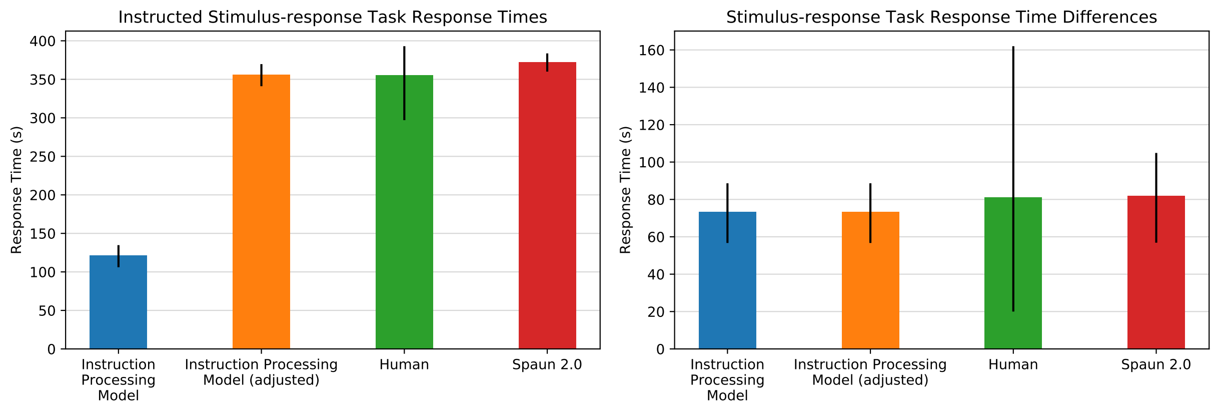


Figure 5.20: Comparison of response times for the instructed stimulus-response task. (Left) Comparison of the response times for the instructed stimulus-response task between the original instruction processing model from [Choo and Eliasmith, 2013], human data reported in [Grice et al., 1982], and the Spaun 2.0 model. As with Figure 5.17, two response times are reported for the original instruction processing model, the "raw" response times, and the response times taking into account averaged visual and motor processing times. (Right) Comparison of response time differences between the instructed and non-instructed variants of the stimulus-response task. Note that the error bars for this graph are the absolute minimum and maximum differences in response times between the two task variants, using the error bar data reported in the previous graphs.

5.3.4 Stage 3: Delayed Instructed Stimulus-Response Task

Stage 2 of the instruction processing system integration process allowed Spaun to, for the first time, adjust its responses based of the information provided as an instruction. However, for Spaun to be able to use the instruction processing system to augment the action selection hierarchy, the instruction processing system needs to be able to change the values stored within the memory networks of Spaun (in particular, the task state memory networks).

In the Stage 2 integration of the instruction processing system, for Spaun to generate its responses for the instructed stimulus-response task, the visual stimulus is required to be present. If it is absent, the instruction processing system will not have the required conditional information to generate the appropriate consequence output (and thus, no response will be generated by Spaun). As a consequence, this implies that the Stage 2 integration is insufficient for providing the instruction processing system with the ability to affect Spaun's memory system.

To address this issue, Stage 3 of the integration process focuses on how the instruction processing system is integrated with the memory systems in Spaun. This is achieved by implementing the delayed instructed stimulus-response task.

Task Description: The delayed instructed stimulus-response task is similar to the stimulus-response task, with the exception that a delay is introduced between the presentation of the stimulus, and the prompt for Spaun's response.

Task Syntax: The input character sequence for the stimulus matching task follows the form:

$$\mathbf{M}\ x\ \mathbf{?}\ r,$$

where the character "**M**" indicates that Spaun is to perform the delayed instructed stimulus-response task, and as with the previous two stages, x is the stimulus provided to Spaun, and r is the response Spaun provides to the stimulus image.

Task Instruction Format: The task instruction format for the delayed instructed stimulus-response task is identical to the format for the instructed stimulus-response task (see Eq. (5.13)).

Conceptual Implementation: Stage 2 of the integration process demonstrated that the instruction processing system can be used to extract meaningful data from the instruction semantic pointer. In addition, Stage 2 demonstrated that this data can be appropriately routed to the motor system so that Spaun can generate the correct responses. Conceptually, the implementation of the delayed and instantaneous variants of the instructed stimulus-response task is similar, with the primary difference being where the information extracted by the instruction processing system is routed. While this information is routed to the motor system (via the decoding system) in the instantaneous task variant, for the delayed task variant, the information needs to be routed to Spaun's memory system (which is then routed to the motor system when the "**?**" prompt is given).

System Requirements: The system requirements for this task are identical to those of Stage 2.

Functional Implementation: As was the case with the previous integration stages, the first step is to assemble the task initialization condition-consequence pair. For this instructed task (and the tasks in Stage 4 and 5), a new task semantic pointer representation ("INSTRUCT") is used. Additionally, the use of the "**M**" control character (in contrast to the "A" control character in previous tasks) to indicate the start of the task means that the condition-consequence pair for this task is slightly different than for Spaun's previous tasks. The condition-consequence pair to initialize the delayed instructed stimulus-response task is:

$$vision \bullet \mathrm{M} \longmapsto \textsf{\textbf{INSTRUCT}} \Rightarrow task, \textsf{\textbf{STORE}} \Rightarrow task\_stage, \textsf{\textbf{FORWARD}} \Rightarrow task\_decode \quad (5.15)$$

Note that for this task, the initialization condition-consequence pair sets initial values for the *task_stage* and *task_decode* representations. This is to ensure that unless altered by the instruction provided, Spaun will default to storing the incoming data into "Memory Block 1", and will also default to performing the "forward" list decoding when prompted for a response.

With the task initialization complete, the next step is to set up the routing logic for this task. Since the transformation network already projects to the memory system, the **DIRECT** task stage can be used to route information from the output of the instruction processing system to the inputs of the memory system, just as it was done for the previous integration stages. However, the task stage representations also determine which memory block (within the memory system) information gets stored in (see Table 3.6). This would imply that a separate "**DIRECT**" task stage would be needed for each of the memory blocks in the memory network.

To avoid this overhead, a different routing solution has been implemented: a direct projection is created from the output of the instruction processing system to the input of the memory system. However, since the memory system receives direct projections (i.e., projections not controlled by the action selection hierarchy) from the information encoding system, a selector circuit (see Figure 3.29) is used to be able to switch between the two sources of information. The "select" flow control signal to this input selector is then configured to route information from the instruction processing system when the *task* representation is **INSTRUCT**, and to route information from the encoding system otherwise. It is important to note that the use of the input selector networks also lays the foundation for how other sources of information can be routed to the memory networks while minimizing conflicts (e.g., accidental addition of representations) between the different sources.

In addition to the routed information, the memory networks also require a control signal to direct when they are to store their input values (see Figure 3.31). For information projected from the encoding system, the memory control signal is generated from the output of the visual change detector network (see Figure 4.4) – this is because Spaun is driven mostly by changes in visual stimuli. As for information being routed from the instruction processing system, the same concept can be applied – the memory control signal is generated based on changes detected in the output of the extracted *data* semantic pointer. As with the information inputs to the memory network, a selector network is used to ensure the memory control signal being used is matched with the source of the information being fed into the memory networks.

Figure 5.21 illustrates the changes required to the instruction processing system (i.e., the addition of the change detect network). Likewise, Figure 5.22 illustrates the updated memory system with the changes (additional selector networks) required to support the delayed instructed stimulus-response task. Finally, Figure 5.23 shows how the output of the instruction processing system is projected to the memory system. Note that because the memory network receives a direct projection from the output of the instruction processing system, no other condition-consequence pair is required to implement the delayed instructed stimulus-response task.

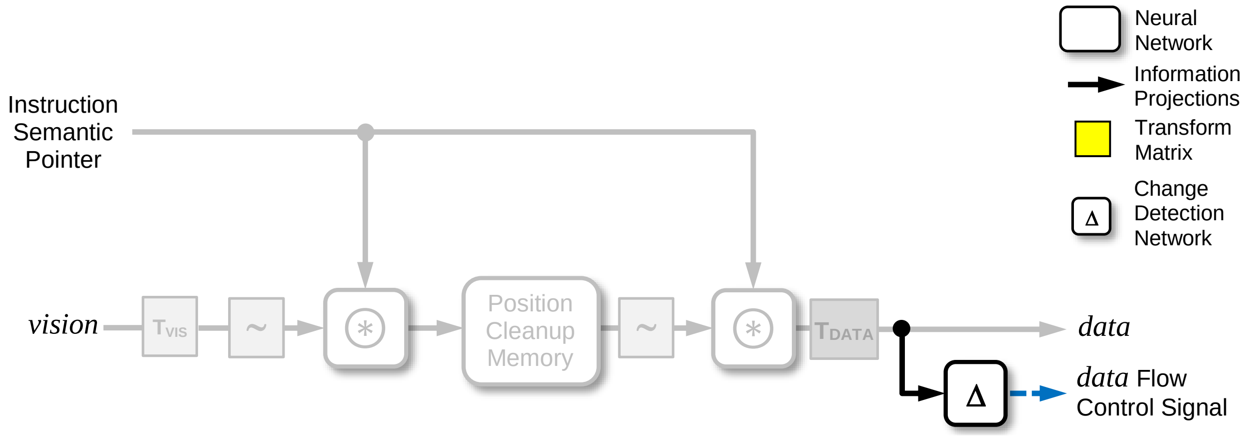


Figure 5.21: Schematic diagram of the instruction processing system updated to support the delayed instructed stimulus-response task. Existing networks and projections from the initial implementation (see Figure 5.14) have been grayed out to better indicate the modifications made.

5.3.4.1 Results

Figure 5.24 demonstrates Spaun performing the delayed instructed stimulus-response task using the same instructions as before, with the exception that instead of immediately producing its response, Spaun is instructed to remember the desired output (which Spaun is prompted for at a later time).

- See “1”, remember “8”; see “2”, remember “1”;
- See “1”, remember “2”; see “2”, remember “8”.

The key difference between this task and the non-delayed variant of the task is that the stimulus is no longer present when Spaun is prompted for the stimulus-matched response. In Figure 5.24, this is illustrated by the presentation of the “**?**” prompt, which removes the presence of the digit stimulus as Spaun is prompted for its response – and just as in the 8 original tasks, Spaun only responds after the “**?**” is removed. As the figure demonstrates, the changes introduced in Stage 3 of the integration process allows Spaun to correctly complete this task.

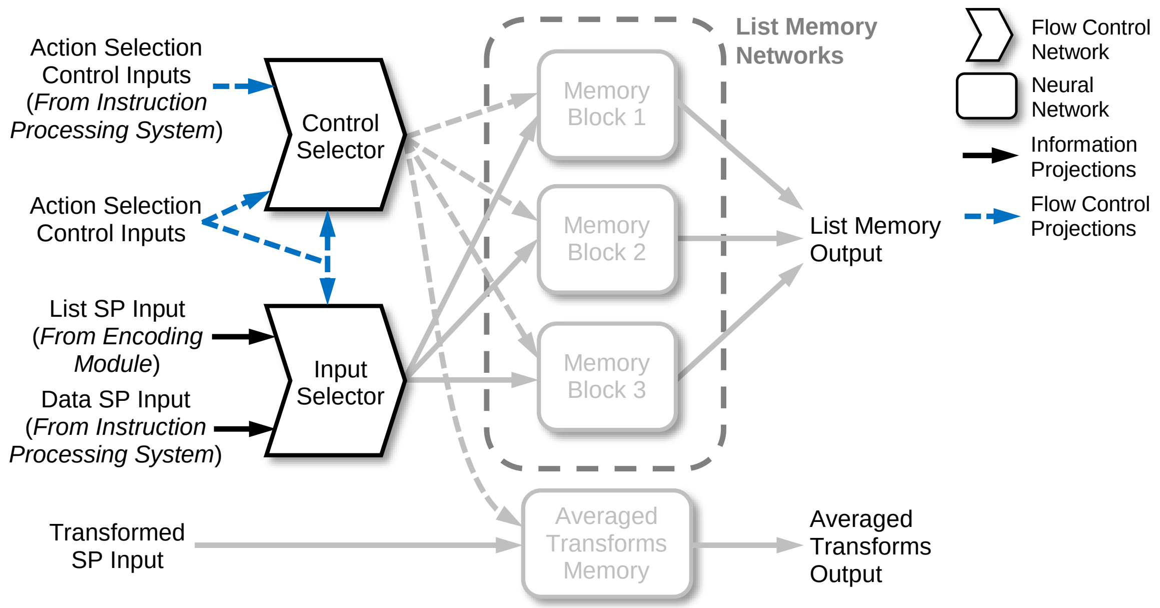


Figure 5.22: Schematic diagram of Spaun 2.0's memory system updated to support the delayed instructed stimulus-response task. Pre-existing networks and projections from the Spaun 2.0 implementation (see Figure 3.25) have been grayed out to better indicate the modifications made.

Population data was also collected for the delayed instructed stimulus-response task. Figure 5.25 plots Spaun's performance accuracy as the number of tasks within the provided instruction is increased from 2 to 6. As the figure demonstrates, Spaun's ability to successfully extract information from the instruction semantic pointer decreases as the complexity of the semantic pointer increases, dropping to about 50% when the instruction semantic pointer contains 6 instructed tasks. This behaviour is to be expected as adding instructed tasks increases the number of superfluous semantic pointer terms introduced during the information extraction process. This decreases the likelihood of Spaun's ability to either determine the appropriate task position semantic pointer, or the data that is to be stored in working memory (resulting in a failed list recall when prompted).

Figure 5.26 plots Spaun's performance on a slight variant of the delayed instructed stimulus-response task. In the figure, Spaun has been tasked to respond with a list of varying length when provided with a visual stimulus. This plot is shown to demonstrate the difference between the representations presented as a visual list, versus as a list encoded in the instruction semantic pointer. As the plot shows, the recency effect is missing from the recall data of the instructed list. This is to be expected as the primacy and recency effects are a consequence of how lists are encoded into memory as each list item is presented to Spaun (i.e., because of the rehearsal and the

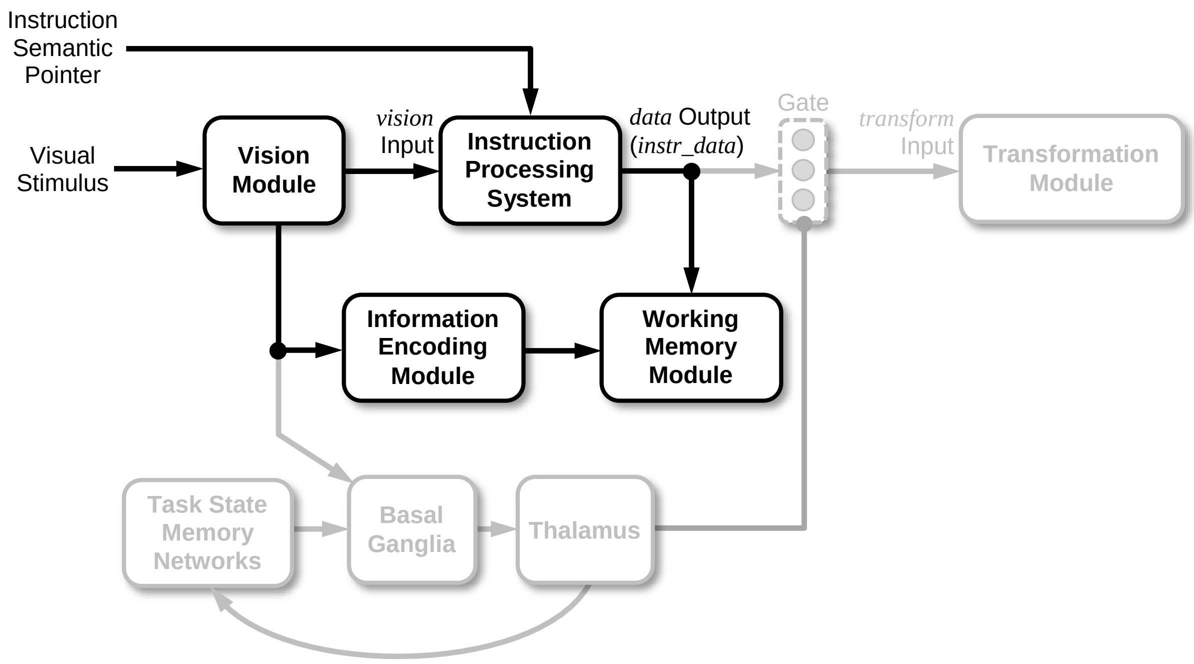


Figure 5.23: Schematic diagram of the networks and projections necessary to implement the delayed instructed stimulus-response task. Note that only the networks involved with the instructed stimulus-response tasks (Stage 2 and Stage 3) are shown – the rest of Spaun's networks are omitted to simplify the network diagram. Additionally, the networks and projections involved solely with the non-delayed instructed stimulus-response task has been grayed out to provide a comparison between the two implementations.

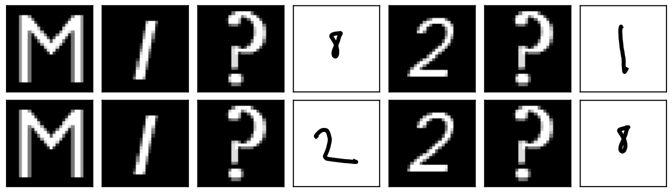


Figure 5.24: Recorded input stimuli and Spaun outputs for the delayed instructed stimulus-response task. In the figure, two iterations of the task are shown. During the first iteration (top row) Spaun is provided with the instruction "see '1', remember '8'; see '2', remember '1"', and during the second task iteration (bottom row), Spaun is provided with the instruction "see '1', remember '2'; see '2', remember '8"'. As in Figure 5.19, the data shown was collected in one simulation run of the Spaun model.

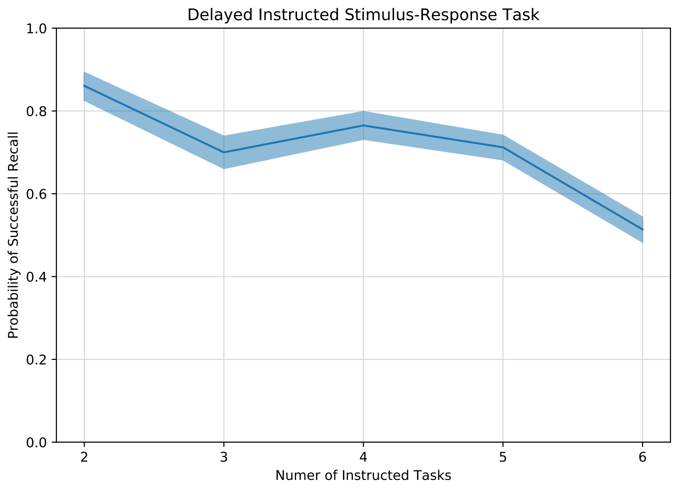


Figure 5.25: Delayed instructed stimulus-response task performance for increasing number of instructed tasks included in the instruction semantic pointer. See accompanying text for an analysis of the results.

decay parameters, each list item contributes a different amount to the overall list representation). This is in contrast to the list representations encoded in the instruction semantic pointer, where every item in the list contribute the same amount to the overall list representation.

Figure 5.26 also plots Spaun's performance on a 3-list variant of the delayed instructed stimulus-response task. Similar to Figure 5.25, data has been collected for an increasing number of instructed task contained within the instruction semantic pointer (from 1 to 3 instructed tasks). This plot serves to demonstrate that the probability of successfully extracting information from the instruction semantic pointer decreases dramatically if the "payload" of the instruction is complex. In the case of the 3-list variant of the task, each additional instructed task added to the instruction adds an 4 bound semantic pointer products (1 double-bound product, and 3 triple-bound products) to the instruction semantic pointer. In contrast, in Figure 5.25, each instructed task added to the instruction only adds 2 bound products (1 double-bound product, and 1 triple-bound product) to the instruction semantic pointer.

5.3.5 Stage 4: Instructed Stimulus-Task Task

Stage 3 of the integration process demonstrated that with the appropriate modifications to the instruction processing system and working memory module, the instruction processing system is capable of making changes to the representations stored in working memory. The purpose of Stage 4 is to apply the methods described in Stage 3 to the "task state" memory networks

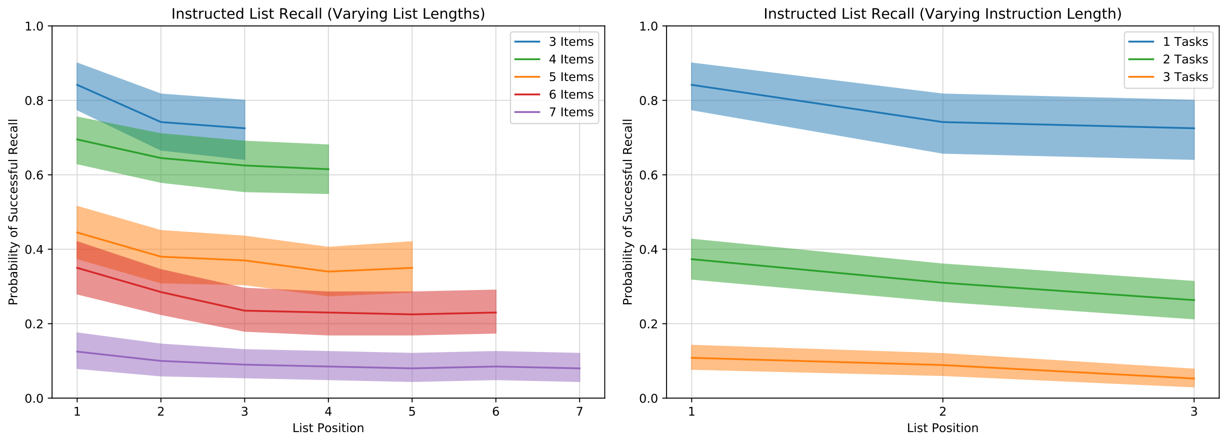


Figure 5.26: Spaun performance accuracy on list variants of the delayed instructed stimulus-response task. (Left) Recall curves for instructions containing one instructed list recall task, for lists increasing in length from 3 to 7. (Right) Recall curves for instructions containing 1 to 3 instructed list recall tasks, with each list being a fixed length of 3. See accompanying text for an analysis of the results.

contained within the action selection hierarchy, which should theoretically allow the instruction processing system to directly affect the task progression logic controlled by the action selection hierarchy.

Task Description: The instructed stimulus-task task is similar to the instructed stimulus-response task, with the only difference being that instead of a response being mapped to a stimulus, one of Spaun's original 8 tasks is mapped to a stimulus image.

Task Syntax: The input character sequence for the stimulus matching task follows the form:

$$\mathbf{M}\ x\ s_1\ \ldots\ s_n\ \mathbf{?}\ r_1\ \ldots\ r_n,$$

where the character "**M**" indicates that Spaun is to perform the delayed instructed stimulus-response task, and as with the previous two stages, x is the stimulus provided to Spaun, s_1 to s_n are the instructed task's specific stimulus inputs (control characters and digits), and r_1 to r_n are the response Spaun provides to answer the instructed task. As an example, if the stimulus-task mapping mapped the "1" stimulus to the counting task, s_1 to s_n would be "▶ x ◀ ▶ y ◀", and r_1 would be the result of the counting task (i.e., $x + y$).

Task Instruction Format: The task instruction format for the instructed stimulus-task task is identical to the instruction format of the instructed stimulus-response task (see Eq. (5.13)), with the exception that the "payload" is tagged with "**TASK**", instead of "**DATA**" (since the mapping is to a specific task, and not to working memory). For this task, the general encoding format for the instruction semantic pointer is then:

$$\textbf{INSTRUCTION} = \textbf{POS}N \circledast ((\textbf{VISION} \circledast \mathbf{x})\mathbf{P}_{ant} + (\textbf{TASK} \circledast \mathbf{y})\mathbf{P}_{cons}) + \ldots,$$

where $\mathbf{x}$ is the semantic pointer representation of the trigger stimuli, and $\mathbf{y}$ is the semantic pointer representation (from Table 3.5) of the associated task. The instruction semantic pointer may also include information to modify the *task_stage* and *task_decode* representations stored in memory. These are tagged with the representations "**TASK_STAGE**" and "**TASK_DECODE**" respectively.

Conceptual Implementation: The memory networks contained within the working memory module, and the memory networks used in the action selection hierarchy have similar operational dynamics. Thus, it should be straightforward to apply the techniques used in Stage 3 to the Stage 4 integration process in order to integrate the instruction processing system with the memory networks within the action selection hierarchy.

System Requirements: The system requirements for this task are identical to those of Stages 2 and 3.

Functional Implementation: To initialize the instructed stimulus-task task, Spaun needs to internally register that it is performing an instructed task. Stage 3 introduced the "**INSTRUCT**" task representation, that indicates to Spaun to route the outputs of the instruction processing system to the rest of Spaun's modules. As this is the desired routing state for the instructed stimulus-task task, the task initialization condition-consequence pair determined in Stage 3 (see Eq. (5.15)) can be re-used to initialize the stimulus-task task as well.

The next step to the integration process involves ensuring that the task (and possibly task stage or task decode states) is extracted from the instruction semantic pointer in a robust fashion. The encoding schema of the instruction semantic pointer does not place any restrictions on the type of data it can hold, and any data can be encoded, as long as the appropriate "tags" are used. This being the case, extracting the task representations from the encoded instruction conditions is no different than extracting the "data" representation.

However, because the task semantic representations came from a predefined list of semantic pointers (see Tables 3.5, 3.6 and 3.7), an associative memory network can be used to make the information extraction process more robust. The output of the associative memory networks can also be projected into the change detect networks to generate the flow control signals needed for the correct operation of the task state memory networks. These changes to the instruction processing system are illustrated in Figure 5.27.

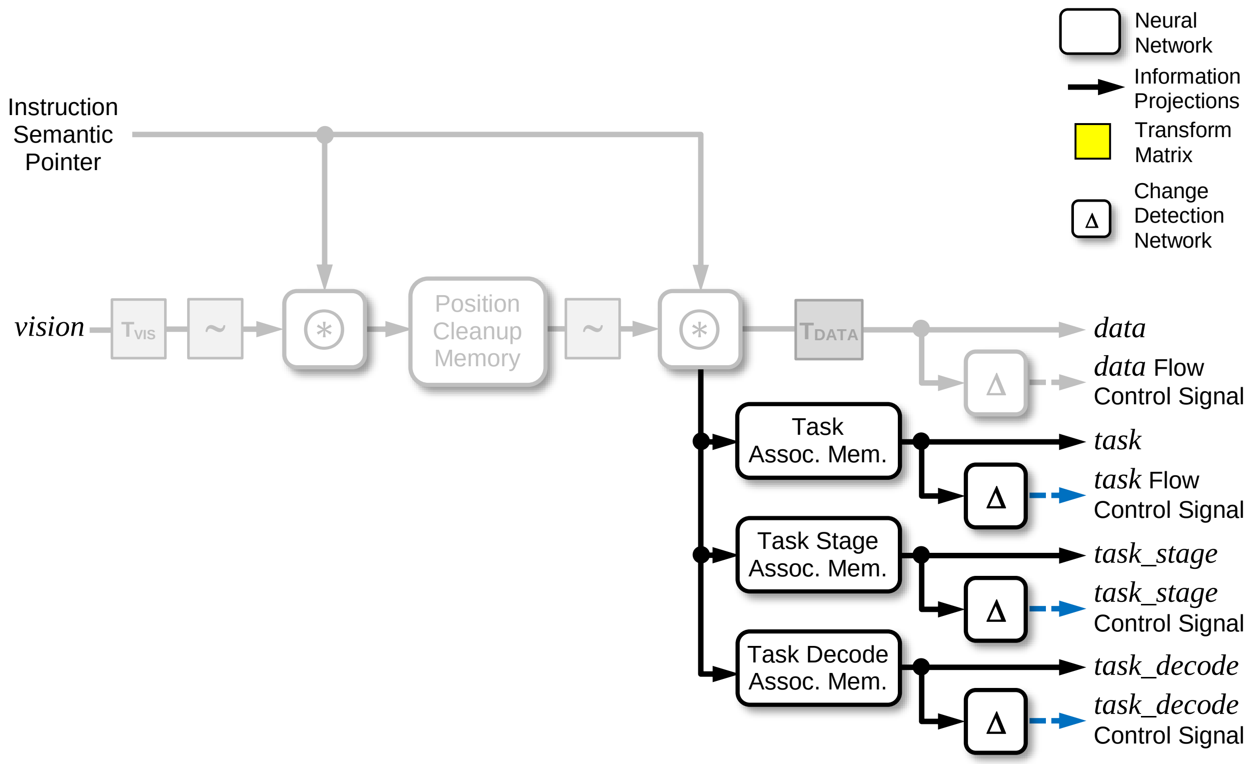


Figure 5.27: Schematic diagram of the instruction processing system updated to support the delayed instructed stimulus-task task. Existing networks and projections from the Stage 3 implementation (see Figure 5.21) have been grayed out to better indicate the modifications made.

Finally, the task information extracted from the instruction processing system needs to be routed to the task state memory networks. Because the representations in the task state memory networks are modified only by the action selection hierarchy (in particular, the condition-consequence pairs that dictate the logic of each task), the inputs to the task state memory are controlled solely by the thalamus network. That being the case, a condition-consequence pair can be used to route the task information to the task state memory networks (just like in Stage

2). The condition-consequence pair used is:

$$\begin{aligned} task \bullet \textsf{\textbf{INSTRUCT}} \longmapsto\ & instr\_task \Rightarrow task, \\ & instr\_task\_stage \Rightarrow task\_stage, \\ & inst\_task\_decode \Rightarrow task\_decode \end{aligned}$$

5.3.5.1 Results

Figure 5.28 demonstrates Spaun performing the instructed stimulus-task task. In the figure, Spaun has been instructed to perform the following tasks for the first two rows:

- See “1”, perform the working memory task,
- See “2”, perform the question answering task,

followed by the following tasks for the last row of images:

- See “1”, perform the counting task,

As the figure shows, Spaun is able to perform the tasks correctly, and as with the results from Stage 2 and 3 of the integration process, without stopping the simulation to reconstruct the Spaun model. The results from this figure also demonstrates that the changes made in the Stage 4 integration process have provided the instruction processing system with the ability to modify Spaun's internal task representations, which is a necessary step to allowing Spaun to perform tasks outside of its 8 original tasks.

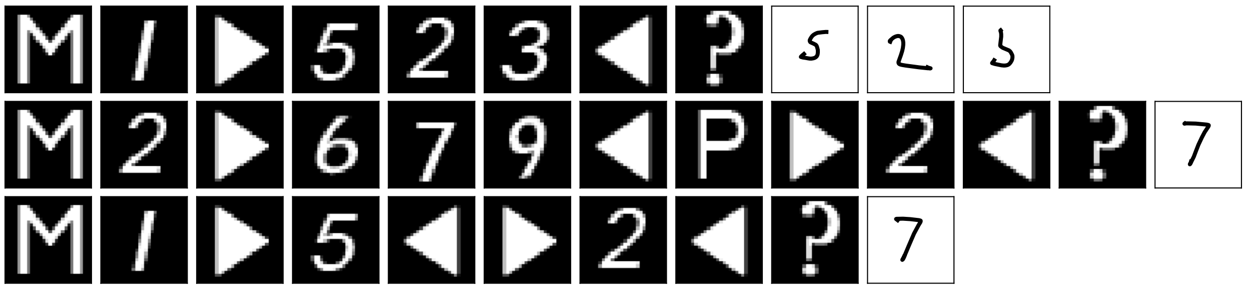


Figure 5.28: Stimulus inputs and Spaun outputs for the instructed stimulus-task task. See accompanying text for details.

For this task, data was also collected measuring Spaun's probability of successfully performing an instructed task as the number of instructed tasks within an instruction is increased from 2 to 6. The results of the analysis of the data is shown in Figure 5.29. As the plot in Figure 5.29

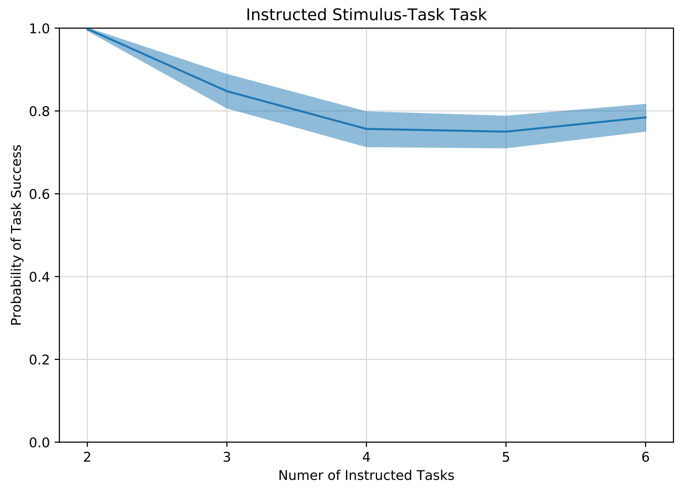


Figure 5.29: Instructed stimulus-task task performance for increasing number of instructed tasks included in the instruction semantic pointer. See accompanying text for an analysis of the results.

demonstrates, the same general trend of a decrease in the probability of task success is observed. It is, however, observed that compared to Figure 5.25, the performance accuracies are higher in general. It is hypothesized that this is because the payload for the stimulus-task task contains, on average, a single-bound semantic pointer (e.g., "**TASK** $\circledast$ **MEMORY**"). In contrast, the payload for the delayed stimulus-response task contains a double-bound semantic pointer (e.g., "**DATA** $\circledast$ **POS1** $\circledast$ **TWO**"), which introduces more noise (and lowering the task performance success rate) than the latter payload. It should be noted that the slight increase in the task performance accuracy for the last data point (6 instructed tasks in the instruction) is unexpected. Additional data collection is necessary to elucidate the exact cause of this result.

5.3.6 Stage 5: Sequentially Instructed Tasks

Stage 4 of the integration process demonstrated that, with the appropriate modifications, the instruction processing system can be used to alter the representations stored in the task state memory networks. This allows the instruction processing system to emulate the functions of the first layer of the action selection hierarchy (i.e., the basal ganglia-thalamus network). However, one aspect of the task progression logic remains to be implemented, and that is the focus of Stage 5 of the integration process.

Each of Spaun's original 8 tasks can be decomposed into the sequential activation of condition-consequence pairs[45], and it is this sequential activation that Stage 5 aims to integrate. To do

[45]Note that while the condition-consequence pairs for the induction tasks essentially forms a loop, a fixed length

this, Stage 5 implements the ability to process sequentially instructed tasks.

Task Description: The sequentially instructed task is a task where Spaun is provided with an ordered (but not necessarily in order) list of tasks. Spaun can then either be instructed to perform a task corresponding to a specific ordinal number, or to step through the list of tasks sequentially (following the ordinal numbering of each task).

Task Syntax: The input character sequence to sequentially process instructions follow the form:

$$\mathbf{M}\ \mathbf{P}\ p\ s_1\ \ldots\ s_n\ \mathbf{?}\ r_1\ \ldots\ r_n,$$

where the character sequence **MP** indicates to Spaun that it is to perform the task associated with the ordinal number p. Similar to Stage 4, s_1 to s_n are the instructed task's specific stimulus inputs, and r_1 to r_n are the response Spaun provides to answer the instructed task.

For the sequentially instructed tasks, Spaun can also be presented with the control character **V** to inform it to advance to the next task (following the order of the numbering) in the task list. The input character sequence for this scenario is:

$$\mathbf{V}\ s_1\ \ldots\ s_n\ \mathbf{?}\ r_1\ \ldots\ r_n,$$

It is important to note that the **V** control character is meant to only be used *after* a starting task number (using the **MP**p character sequence) has been provided.

Conceptual Implementation: The encoding schema of the instruction semantic pointer intrinsically includes the list position for each of the condition-consequence pairs included in the instruction. It should be straightforward to take advantage of this intrinsic positional encoding to implement the sequential instruction processing task. [Choo and Eliasmith, 2013] demonstrated that using the built-in list position tags, it is possible to build an instruction processing system to perform a counting task, which involved the sequential execution of each step of the count.

However, the use of the intrinsic list position tags has a major disadvantage – the method can only be applied to condition-less condition-consequence pairs. Since the list position tags are used to determine which consequence information to extract from the instruction semantic pointer (see Eq. (5.12)), using the list position tags as a positional indicator would negate the need for the conditions as part of the information extraction process (since it would "skip" the first step of

loop can still be "unravelled" into a sequence of condition-consequence pairs.

the two step decoding processing). This implies that this method of sequential processing ignores any conditions set in the instruction semantic pointer, which may not be desired.

Thus, in the Spaun 2.0 implementation, a different approach is used. Instead of re-using the list position tags to encode the task order, the task order is encoded as part of the condition for each condition-consequence pair. In addition to supporting both condition-less and "standard" condition-consequence pairs, this method of encoding the task order information also allows out-of-order tasks to be encoded without needing any additional logic to handle the non-sequential ordering.

Task Instruction Format: The task instruction format for ordered tasks follows the encoding schema used in Stages 2 through 4. For this task, the novel addition is the inclusion of positional information within the condition of each condition-consequence pair. The general instruction encoding format is:

$$\textsf{\textbf{INSTRUCTION}} = \textsf{\textbf{POS}}N \circledast ((\textsf{\textbf{POS}}M + \ldots)\mathbf{P}_{ant} + (\textsf{\textbf{TASK}} \circledast \mathbf{y} + \ldots)\mathbf{P}_{cons}) + \ldots,$$

where $\textsf{\textbf{POS}}M$ is the order information associated with the specific condition-consequence pair, and $\mathbf{y}$ is the semantic pointer representation of the task associated with the specific condition-consequence pair. As an example, the following list of tasks:

1. Task 2: See "3", perform the digit recognition task
2. Task 1: Perform the working memory task

is encoded as:

$$\begin{aligned}\textsf{\textbf{INSTRUCTION}} = \textsf{\textbf{POS1}} &\circledast (0.5 \times (\textsf{\textbf{POS2}} + \textsf{\textbf{VISION}} \circledast \textsf{\textbf{THREE}})\mathbf{P}_{ant} + (\textsf{\textbf{TASK}} \circledast \textsf{\textbf{RECOG}})\mathbf{P}_{cons}) + \\ \textsf{\textbf{POS2}} &\circledast ((\textsf{\textbf{POS1}})\mathbf{P}_{ant} + (\textsf{\textbf{TASK}} \circledast \textsf{\textbf{WM}})\mathbf{P}_{cons})\end{aligned}$$

System Requirements: The system requirements for this task are identical to those of Stages 2,3 and 4.

Functional Implementation: Because the task order information is encoded within the conditions of the instruction semantic pointer, the infrastructure developed for the previous integration stages can be reused to support the sequential execution of tasks. The condition-consequence pairs to initialize and route the instruction processing outputs are then:

$$\begin{aligned}
vision \bullet \mathbf{M} &\longmapsto \textsf{\textbf{INSTRUCT}} \Rightarrow task, \textsf{\textbf{STORE}} \Rightarrow task\_stage, \textsf{\textbf{FORWARD}} \Rightarrow task\_decode \\
task \bullet \textsf{\textbf{INSTRUCT}} &\longmapsto instr\_task \Rightarrow task, instr\_task\_stage \Rightarrow task\_stage, \\
&\qquad inst\_task\_decode \Rightarrow task\_decode
\end{aligned}$$

To support the use of the position semantic pointers as conditional arguments, a position memory network is added to the instruction processing system. The inputs to the position memory network are controlled by the task's "**P**" and "**V**" control characters. When presented with the "**P**" character (and only in the "instruction" task state), Spaun is required to store the next incoming digit and use it as a condition to drive the extraction of information from the instruction semantic pointer. To achieve this, a new task stage representation (**INSTRUCT_POS**) is added, and the following two condition-consequence pairs are used to shift Spaun into and out of this new task stage:

$$\begin{aligned}
0.5 \times (task \bullet \textsf{\textbf{INSTRUCT}} + vision \bullet \mathbf{P}) &\longmapsto \textsf{\textbf{INSTRUCT\_POS}} \Rightarrow task\_stage \\
0.5 \times (task \bullet \textsf{\textbf{INSTRUCT}} + task\_stage \bullet \textsf{\textbf{INSTRUCT\_POS}}) &\longmapsto \textsf{\textbf{STORE}} \Rightarrow task\_stage
\end{aligned}$$

In the **INSTRUCT_POS** task stage, the instruction processing system is configured to convert any visually presented digits into their corresponding position semantic pointer (using an associative memory network, similar to the question answering task – see Figure 3.17). The position representation is then stored in the position memory network within the instruction processing system, where it is treated no differently than any of the other conditional arguments (e.g., *vision*).

To advance the position memory by one increment following the presentation of the "**V**" control character, the "**V**" representation is used to create a flow control signal that routes the output of the position memory through a position increment circuit, and then back into the position memory. This flow control signal generation is similar to the method used to dictate the flow of information between memory circuits for the RVC task example shown in Figure 3.32.

The addition of the position memory, the associative digit-to-position mapping, and the increment circuit to the instruction processing system are illustrated in Figure 5.30. As the position semantic pointers are used as conditional arguments – which does not alter the remainder of the consequence information extraction network – no additional changes to the instruction processing system, nor the need for additional condition-consequence pairs, are necessary to implement Stage 5 of the integration process.

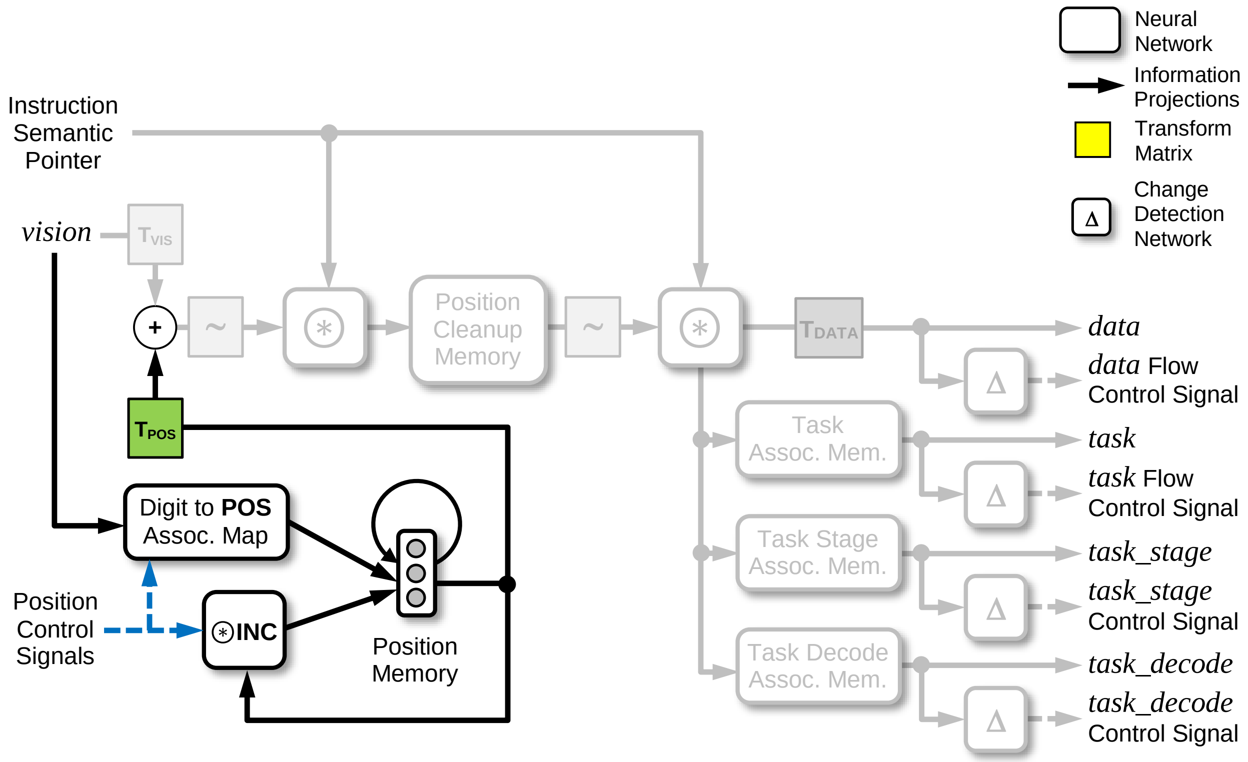


Figure 5.30: Schematic diagram of the instruction processing system updated to support the sequential processing of instructed tasks. Existing networks and projections from the initial implementation (see Figure 5.27) have been grayed out to better indicate the modifications made. Refer to the accompanying text for details.

5.3.6.1 Results

Figure 5.31 demonstrates Spaun performing a positional and sequential processing of instructed tasks. The instruction provided to Spaun for this task was:

1. Task 1: Perform the counting task,
2. Task 2: Perform the working memory task.

In the first row of the figure, Spaun is directed to perform the second task in the instruction (using the control character sequence "**MP2**"). This is followed by a direction for Spaun to perform the first task in the instruction (using the control character sequence "**MP1**"). Last, Spaun is directed to perform the "next" task in the instruction. Since the previous task performed was the first task, Spaun proceeds to perform the second task in the instruction. Looking at the instruction provided, Spaun is expected to perform the working memory task, followed by the counting task, and finally the working memory task again. As the figure demonstrates, the modifications made for the Stage 5 integration process allow Spaun to accurately perform this task. As with Stages 2 through 4, the data shown in the figure was collected over one simulation run, without requiring any offline modifications to be made to the Spaun network.

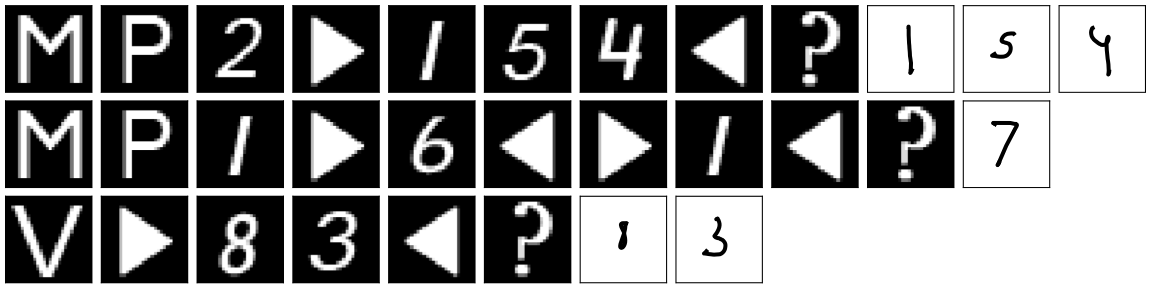


Figure 5.31: Stimulus inputs and Spaun outputs recorded from Spaun performing a sequential instruction of tasks. Refer to the accompanying text for details on the tasks provided to Spaun.

As before, data was collected for this task over an increasing number of instructed tasks contained within the provided instruction. This data is plotted in Figure 5.32, and once again, the general trend of a decrease in task performance accuracy is observed. However, it should be noted that the overall task success rate is higher than in Figure 5.29. Unlike the comparison between Figure 5.25 and Figure 5.29, it is hypothesized that the reason for the increase in task performance accuracy in this case is due to the reduction of complexity in the task conditions (rather than in the task payload). As an example, the task conditions for the sequential tasks contain an unbound semantic pointer (e.g., "**POS2**"), compared to the task conditions for the stimulus-task task which contain a bound pair of semantic pointers (e.g., "**VIS** $\circledast$ **TWO**").

5.3.7 System Integration Results

The previous sections detailed how the instruction processing system is integrated into Spaun 2.0 in the effort of enhancing one of the core systems in the model, demonstrating the ability to: use information from the instruction semantic pointer to directly affect Spaun's output (Stage 2); extract and modify representations stored in working memory, so that it can be used at a later time (Stage 3); modify the task state representations stored in the action selection hierarchy (Stage 4); and sequentially process tasks contained within the instruction semantic pointer (Stage 5). While, using these changes, it is theoretically possible to direct Spaun to perform tasks that it was not originally designed to do, this has yet been demonstrated.

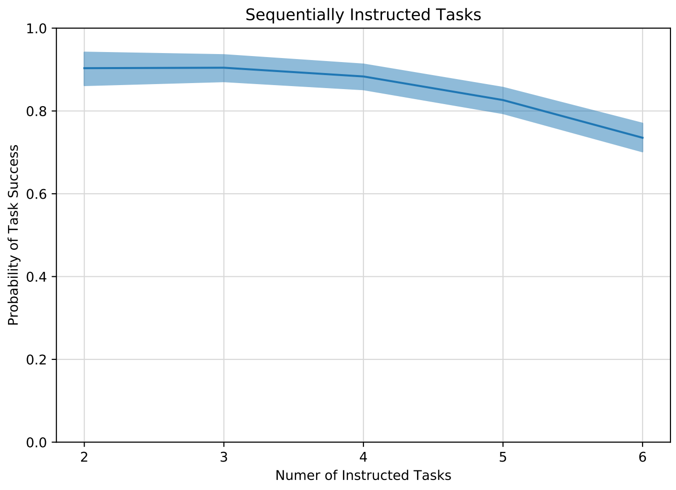


Figure 5.32: Task performance success rates for sequentially instructed tasks, for an increasing number of instructed tasks (from 2 to 6) within the instruction. Note that for the data plotted, in order to minimize complexity in analyzing the results, Spaun was directed only with positional task probes (i.e, "**V**" was not used). See accompanying text for an analysis of the results.

Before continuing, however, it is important to note that even with the changes made to the Spaun network, each step of any new task defined has to exist as a processing step of the existing 8 Spaun tasks. As an example, without the changes to the transformation module to implement the direct stimulus-response task, Spaun would have been unable to perform the task (even if instructed) because the neural projections necessary to complete the task would not have been present.

As a demonstration of the capabilities of the Spaun 2.0 model with the integrated instruction processing system, Figures 5.33 to 5.34 present customized instructed tasks, ordered by the complexity of each task. It should be noted that for the customized tasks, no population level task performance statistics were collected as they are expected to follow the trends observed in Figure 5.32.

Extended Question Answering Task: To start, Figure 5.33 shows Spaun instructed to perform a customized task where all of the task processing elements come from a single "original" Spaun task (in this case, each task processing element is part of the original question answering task). To generate the behaviour shown in the figure, the instruction provided to Spaun was:

1. Task 1: Perform the question answering task using the information to be provided.
2. Task 2: Using the list presented in Task 1, perform the "kind" question using the information to be provided.

3. Task 3: Using the list presented in Task 1, perform the "position" question using the information to be provided.

In essence, the instruction provided to Spaun performs the "standard" question answering task, with the important exception that the task does not have to be restarted every time a new question is asked (i.e., the input list can be reused). Converting the custom task into a format Spaun can process, the equivalent instruction semantic pointer is:

$$\begin{aligned}\textsf{\textbf{INSTRUCTION}} = {} & \textsf{\textbf{POS1}} \circledast ((\textsf{\textbf{POS1}})\mathbf{P}_{ant} + (\textsf{\textbf{TASK}} \circledast \textsf{\textbf{QA}})\mathbf{P}_{cons}) + \\ & \textsf{\textbf{POS2}} \circledast ((\textsf{\textbf{POS2}})\mathbf{P}_{ant} + (\textsf{\textbf{TASK}} \circledast \textsf{\textbf{QA}} + \textsf{\textbf{TASK\_STAGE}} \circledast \textsf{\textbf{QAK}})\mathbf{P}_{cons}) + \\ & \textsf{\textbf{POS3}} \circledast ((\textsf{\textbf{POS3}})\mathbf{P}_{ant} + (\textsf{\textbf{TASK}} \circledast \textsf{\textbf{QA}} + \textsf{\textbf{TASK\_STAGE}} \circledast \textsf{\textbf{QAP}})\mathbf{P}_{cons})\end{aligned}$$

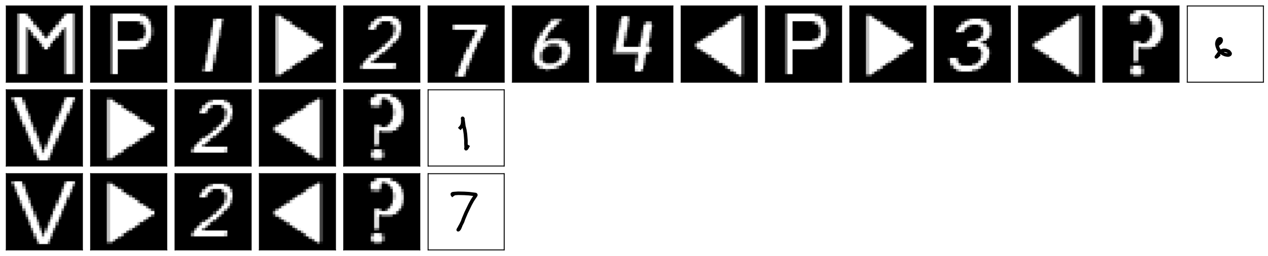


Figure 5.33: Stimulus inputs and Spaun outputs for the extended question answering task. See accompanying text for details.

In Figure 5.33, Spaun is first directed to perform the first task in the instruction (using the character sequence "**MP1**"). It is then presented the list "[**2764**]", and probed for the item in the third position, which Spaun correctly identifies as "6". Spaun is then directed to perform the next task in the instruction (using the control character "**V**"), and presented with the single digit list "[**2**]". Referring to the instruction provided to Spaun, Spaun is to interpret this list as a direction to identify the location of the "2" in the input list shown in the first task. When prompted, Spaun responds with "1", indicating correctly that the "2" is the first item of the original input list. Lastly, Spaun is directed to perform the third task, and once again provided with the list "[**2**]". However, this part of the task requires Spaun to interpret the "2" as a positional probe, and is expected to, when prompted, respond with the second item in the original list – which it does so correctly, identifying the "7" as the second item in the original input list of "[**2764**]".

Applied Counting Task: Figure 5.34 demonstrates the interaction of subcomponents from three different tasks: the working memory task, the counting task, and the question answering task. For this task, Spaun is provided with the instruction:

1. Task 1: Remember the list to be provided.
2. Task 2: Perform the counting task with the list provided in Task 1, with the number of counts that is to be provided.
3. Task 3: Using the results of the counting task, perform the question answering task using the question probe information that is to be provided.

This task (referred to as the "applied counting task") demonstrates that the instruction processing system can be used to create custom tasks where portions of the task use products generated as a result of stepping through the task. The equivalent instruction semantic pointer for applied counting task is:

$$\begin{aligned}\textsf{\textbf{INSTRUCTION}} = {} & \textsf{\textbf{POS1}} \circledast ((\textsf{\textbf{POS1}})\mathbf{P}_{ant} + (\textsf{\textbf{TASK}} \circledast \textsf{\textbf{WM}})\mathbf{P}_{cons}) + \\ & \textsf{\textbf{POS2}} \circledast ((\textsf{\textbf{POS2}})\mathbf{P}_{ant} + \\ & \qquad (\textsf{\textbf{TASK}} \circledast \textsf{\textbf{COUNT}} + \textsf{\textbf{TASK\_STAGE}} \circledast \textsf{\textbf{COUNT1}})\mathbf{P}_{cons}) + \\ & \textsf{\textbf{POS3}} \circledast ((\textsf{\textbf{POS3}})\mathbf{P}_{ant} + (\textsf{\textbf{TASK}} \circledast \textsf{\textbf{QA}})\mathbf{P}_{cons})\end{aligned}$$

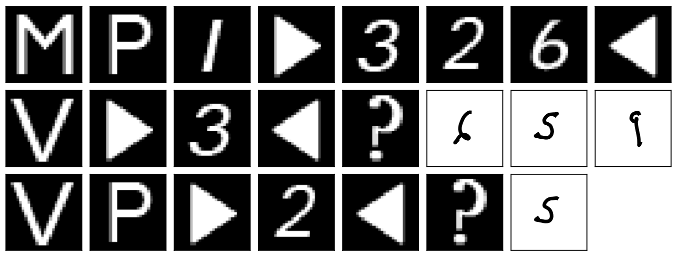


Figure 5.34: Stimulus inputs and Spaun outputs for the applied counting task. See accompanying text for details.

In Figure 5.34, as per the instruction provided to Spaun, it is first directed to perform the working memory task with the list "[**326**]". Note that because Spaun is not prompted for a response, it does not produce any written output. Rather, Spaun just holds the information it has been presented in working memory. Next, it is directed to perform the counting task, and is presented with the list "[**3**]". As per the instruction provided, Spaun is to use this information to modify its counting behaviour (i.e., since it was presented a "**3**", Spaun is to internally count 3 times). Since the internal counting process only happens when Spaun is prompted for a response, it is prompted for a response. Here, it is important to note that because Spaun treats a sequence of numbers as a list of numbers, directing Spaun to perform the counting task on a list of numbers implies that it should increment all of the numbers by the desired amount. That being the case, Spaun's responds correctly with the output list "[**659**]". Finally, Spaun is directed to perform the last part of the instruction, and is provided with the character "**P**", followed by the list "[**2**]".

Referring to the instruction provided to Spaun, these inputs dictate that Spaun is to perform answer the question: "what is the second item in the list?", where the "list" refers to the result of the counting task just prior. As the figure demonstrates, Spaun correctly responds with the answer "**5**".

Applied Induction Task: Finally, Figure 5.35 demonstrates a custom task where an inductive transformation representation is applied to an input that is not originally part of the induction task. For this custom task, the instruction provided to Spaun is:

1. Task 1: Deduce the pattern between pairs of input lists (presented using the RVC task format).
2. Task 2: Perform the question answering task.
3. Task 3: Apply the inductive pattern from Task 1 to the first argument of the question answering task (i.e., the input list).
4. Task 4: Apply the inductive pattern from Task 1 to the second argument of the question answering task (i.e., the probe list).

The instruction semantic pointer representation for this task is:

$$\begin{aligned}\textsf{\textbf{INSTRUCTION}} = \textsf{\textbf{POS1}} \circledast ((\textsf{\textbf{POS1}})\mathbf{P}_{ant} + (\textsf{\textbf{TASK}} \circledast \textsf{\textbf{RVC}})\mathbf{P}_{cons}) + \\ \textsf{\textbf{POS2}} \circledast ((\textsf{\textbf{POS2}})\mathbf{P}_{ant} + (\textsf{\textbf{TASK}} \circledast \textsf{\textbf{QA}})\mathbf{P}_{cons}) + \\ \textsf{\textbf{POS3}} \circledast ((\textsf{\textbf{POS3}})\mathbf{P}_{ant} + (\textsf{\textbf{TASK}} \circledast \textsf{\textbf{FLUID\_IND}} + \\ \textsf{\textbf{TASK\_STAGE}} \circledast \textsf{\textbf{TRANSFORM1}})\mathbf{P}_{cons}) + \\ \textsf{\textbf{POS4}} \circledast ((\textsf{\textbf{POS4}})\mathbf{P}_{ant} + (\textsf{\textbf{TASK}} \circledast \textsf{\textbf{FLUID\_IND}} + \\ \textsf{\textbf{TASK\_STAGE}} \circledast \textsf{\textbf{TRANSFORM2}})\mathbf{P}_{cons})\end{aligned}$$

It should be noted that in the instruction semantic pointer, Task 3 and 4 of the instruction are being achieved by essentially "tricking" Spaun into considering the arguments used for the question answering task as input cells for the fluid induction task. However, because Spaun does not apply any context to the list representations stored in working memory, this method is considered by Spaun to be perfectly valid.

Figure 5.35 shows inputs and outputs of a Spaun model that has been instructed to perform the applied induction task. First, Spaun is presented with a sequence of lists ("[**1**]", "[**3**]", "[**2**]", and "[**4**]") that it has been directed to parse using the RVC task format. While Spaun has not been instructed to provide a response, as part of the RVC task process, it still computes the inductive transform between the list pairs. Next, Spaun is directed to perform the question answering task. As part of the task setup, Spaun is provided with three inputs: the input list ("[**472**]"), the task

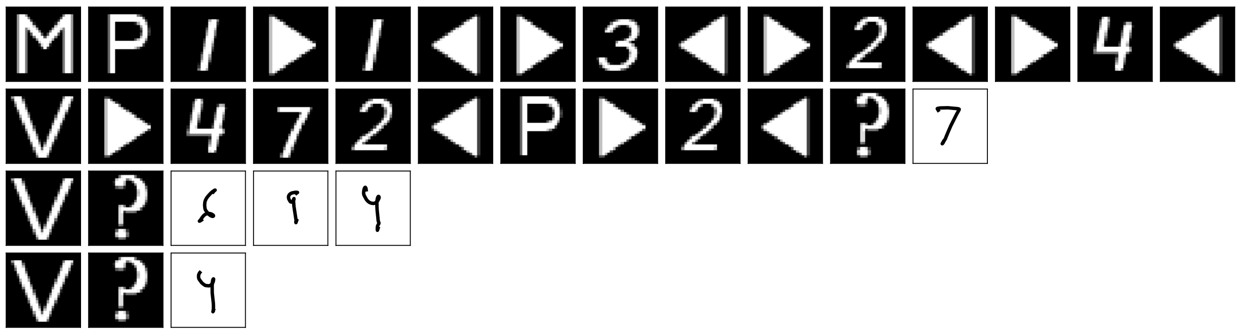


Figure 5.35: Stimulus inputs and Spaun outputs for the applied induction task. See accompanying text for details.

type ("**P**"), and a single digit query list ("[**2**]"). Using this information, Spaun correctly answers the prompt with "**7**". Next, it is directed to apply the inductive transform computed in Task 1 to the first argument of the question answering task (i.e., the list "[**472**]"). Looking at the list pairs from Task 1, the correct pattern to apply to the list is to increment each element by 2, which Spaun does correctly, responding with the list "[**694**]" when prompted. Last, Spaun is directed to apply the inductive transform to the second argument of the question answering task (i.e., the list "[**2**]"). Once again, it does so correctly, responding with the single digit output "**4**" when prompted to do so.

5.4 Discussion

This chapter explored three extensions to the Spaun project that brought new capabilities to the Spaun 2.0 model, providing it with an adaptive motor controller (Section 5.1), a visual system capable of processing 256×256 pixel full-colour images (Section 5.2), and an instruction processing system that allows Spaun to perform customized tasks (Section 5.3). While each extension has been successful in its own right, the generalized design of the re-organized Spaun 2.0 architecture makes it straightforward to interchange variants of functional Spaun modules, as long as the interface to these modules are not changed.

As a final demonstration of a fully integrated Spaun 2.0, Figures 5.36 through 5.38 show a Spaun model constructed with all three extensions described in this chapter performing a variety of tasks (both new and original). This feat is possible because the design of Spaun 2.0 allows variant implementations of Spaun modules to interact with one another as long as the variant implementations do not modify the input and output projections from each module (a requirement that each of the new extensions abide by). As a point of comparison, Table 5.4 shows the neuron counts of the original Spaun 2.0 model, and the neuron counts of a Spaun 2.0 model constructed with all three extensions described in this chapter.

Module Name	Spaun 2.0	Spaun 2.0 w/ Extensions
Vision (MNIST / ImageNet)	186,400	999,284 (+812,884)
Motor (Non-adaptive / Adaptive)	99,850	100,850 (+1,000)
Information Encoding	618,620	618,620
Information Decoding	533,820	533,820
Working Memory	1,540,950	1,540,950
Transformation	903,910	929,510 (+25,600)
Reward Evaluation	600	600
Action Selection Hierarchy	692,860	747,650 (+54,790)
Instruction Processing System	–	1,141,200
Total neuron count	4,577,010	6,637,284 (+2,060,274)

Table 5.4: Comparison of the neuron counts for the original Spaun 2.0 model, and a Spaun 2.0 model containing all three extensions described in this chapter. For modules that have increased in neuron count, the increase in neuron count is indicated in the parenthesis.

In Figures 5.36 through 5.38, Spaun is presented with 10 instances of the instructed stimulus-response task ("**A9**") using the ImageNet dataset as the image source, followed by 2 instances of the original working memory task using Spaun's control characters as the image source. For the first 5 instances of the instructed stimulus-response task, Spaun is provided with the instruction:

- Respond with a "7" when an image of a "police van" is presented,
- Respond with a "3" when an image of a "puck" is presented,
- Respond with a "9" when an image of a "grey whale" is presented.

For the next 5 instances of the instructed stimulus-response task, the instruction provided to Spaun is modified to:

- Respond with a "1" when an image of an "organ" is presented,
- Respond with a "8" when an image of a "grey whale" is presented,
- Respond with a "2" when an image of a "half track" is presented.

And finally, for the 2 instances of the working memory task, Spaun is first presented with the list "[**938**]", which is followed by the list "[**456**]". For Figure 5.36, all of the task instances are performed with Spaun's simulated arm placed in the presence of an unknown force field. Thus, the significance of the digits chosen for the working memory task is the demonstration that adaptive motor controller is able to learn and adapt to the unknown force field for digits it had prior training with (i.e., the digits “9”, “3”, and “8”, which are responses for the instructed stimulus-response task), and digits with which it had no prior training writing (i.e., the digits “4”, “5”, and “6”).

As already mentioned, Figure 5.36 demonstrates Spaun performing the task in the presence of the unknown force field. As a point of reference, Figure 5.37 shows the output of the tasks with the external force field applied, but, using a Spaun model that does not contain the adaptive motor controller. Lastly, Figure 5.38 provides a reference sample of Spaun’s “standard” handwriting when no external force field is applied.

In closing, it should be made clear that the Spaun model is by no means complete, and that there are plenty of opportunities for future work, which will be elaborated upon in the next chapter of this thesis. Some future work relevant to the extensions described in this chapter apply, in particular, to the instruction processing system. Of note is the fact that the instruction semantic pointer provided to Spaun has to be constructed using in-depth knowledge of the structure (and logical task flow) of the action selection hierarchy. This makes the instruction processing system somewhat artificial, deviating from the Spaun project goal of having a fully enclosed neural cognitive system.

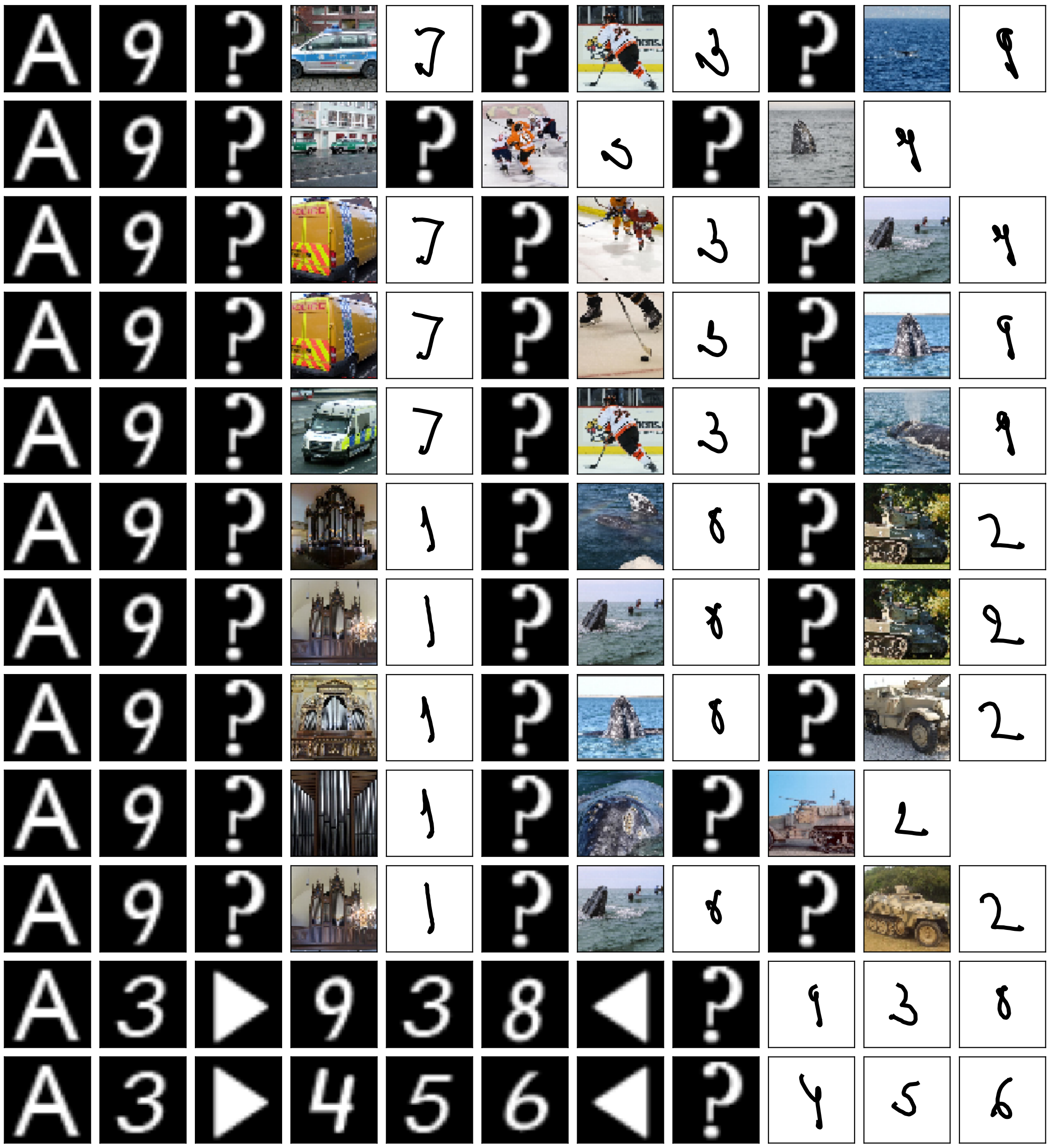


Figure 5.36: Stimulus inputs and Spaun outputs for the combined ImageNet instructed stimulus-response task, and working memory task – using a Spaun model containing all three Spaun 2.0 extensions. For the first 5 rows, Spaun is directed to perform an instructed response-stimulus task with the "police van", "puck" and "grey whale" images. For the next 5 rows, Spaun is asked to perform the instructed response-stimulus task with a different set of images ("organ", "grey whale", and "half-track"). Missing responses (images followed by a question mark) are a result of Spaun misclassifying the input images. Finally, Spaun is directed to perform two list memory tasks (last two rows). Spaun's adaptive motor system can be seen adapting to the unknown force field, being able to reproduce both digits it has had practice writing before, as well as digits which it had no prior practice. See accompanying text for additional details.

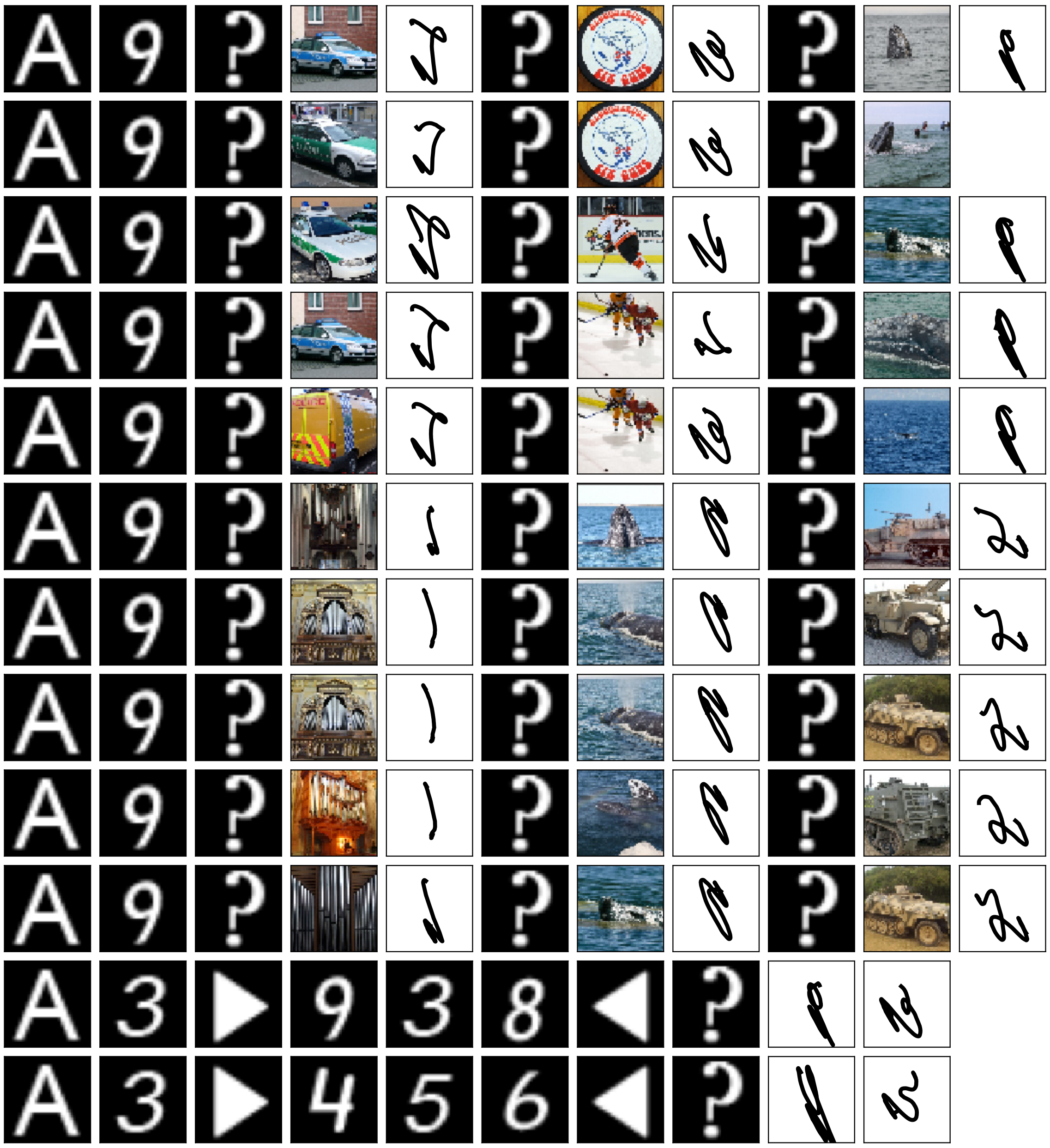


Figure 5.37: Stimulus inputs and Spaun outputs for the combined ImageNet instructed stimulus-response task, and working memory task – using a Spaun model containing only the ImageNet visual hierarchy and instruction processing system extensions. In this figure, an unknown force field has been applied to Spaun's simulated arm. However, without the adaptive motor system, it is observed that Spaun is unable to adapt to the force field, and in the case of the two working memory tasks, unable to reproduce all of the digits in the list in the time allotted.

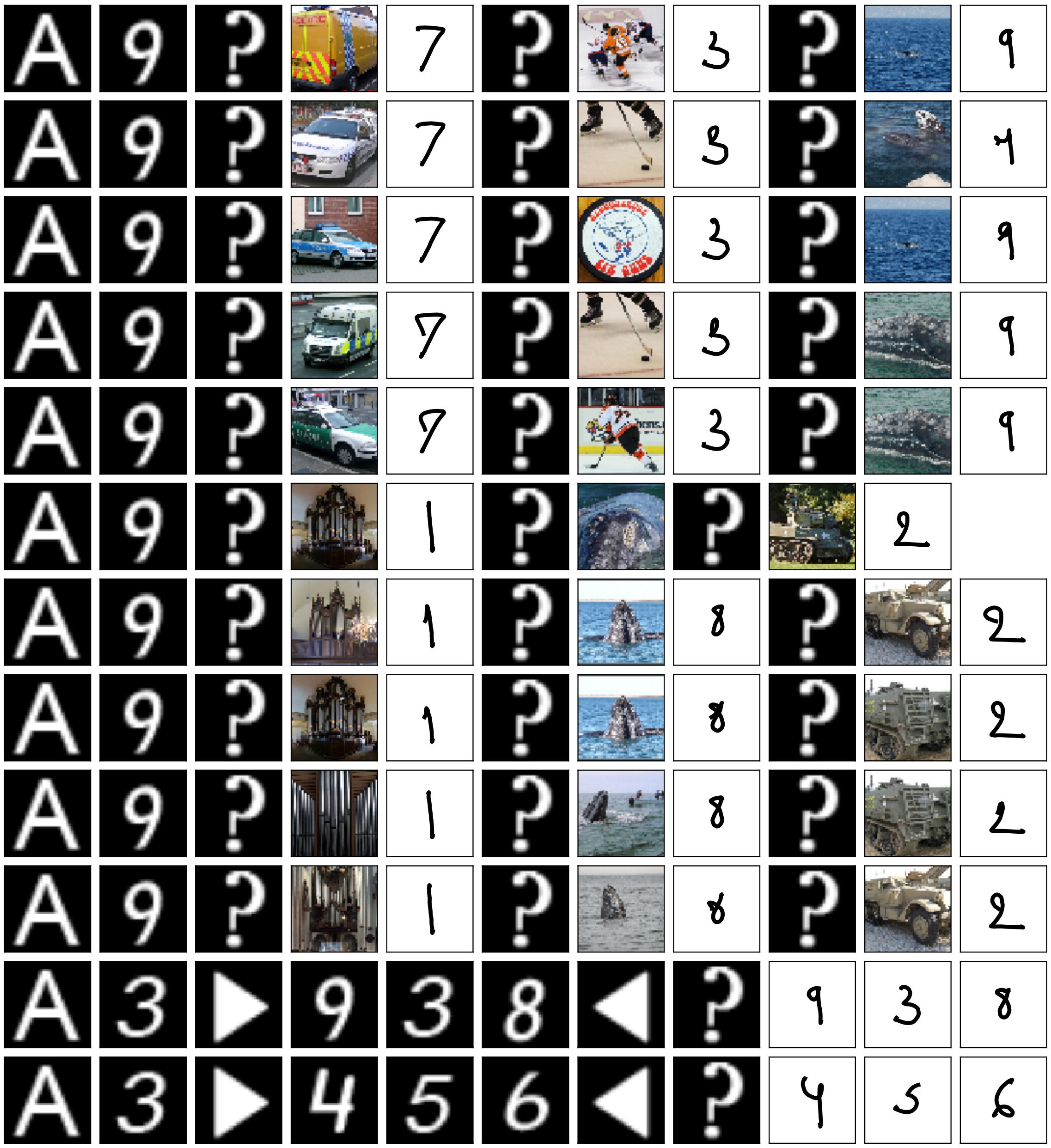


Figure 5.38: Stimulus inputs and Spaun outputs for the combined ImageNet instructed stimulus-response task, and working memory task – showing stereotypical Spaun outputs produced without the presence of an external force field. This figure serves as a reference for the output digits produced in Figure 5.36.

Chapter 6

Conclusion

This thesis described the design and implementation of the Spaun model, its evolution and re-implementation as Spaun 2.0, and three extensions to the Spaun 2.0 model. The original Spaun model was assembled from six pre-existing cognitive models, each originally designed to emulate different functional capabilities of the brain, ranging from visual processing to inductive reasoning. A framework known as the SPA was used to define a common foundation with which the various "precursor" models could be integrated, and the methods described by the NEF allowed the unified model (Spaun) to be implemented in a network of spiking neurons. The Spaun model itself is able to perform 8 cognitive tasks including digit recognition, list memory, self-guided counting, and inductive reasoning. The success of the Spaun model demonstrates several points:

- That the SPA is a scalable approach, and it can be used to design large-scale functional brain models.
- That the methods described by the NEF can be reliably used to implement SPA-based networks as spiking neural networks.
- That by combining the SPA and the NEF, it is possible to implement a fully-enclosed cognitive system, capable of internally generating all the control signals it needs to operate.

While a successful demonstrator of the SPA, the original Spaun model was not without issues. The Spaun 2.0 model was created to address these concerns, by analyzing, re-organizing, and re-implementing the original Spaun architecture. The modifications implemented in Spaun 2.0 included fully spiking visual and motor systems, improved list and averaging memory networks, the addition of a normalization network in the transformation system, and the addition of a bounded-limit mechanism to Spaun's learning system. These changes have resulted in an improvement in Spaun's task performance, particularly with the working memory, question answering, and both induction tasks. Additionally, through the analysis of the original Spaun model, and its re-implementation as Spaun 2.0, several lessons can be derived:

- The neural implementations of the visual and motor systems demonstrate that the design of the control signal generation networks is as equally important as the design of the function network.
- The modifications necessary for the working memory networks illustrate that minor changes to the behaviour of these "building block" networks can have a major change on the behaviour of the entire model. As such, changes to these components must be carefully evaluated.
- The addition of the normalization network to the transformation system, and the addition of the bound-limiting function to the reward evaluation system demonstrate that care must be taken when translating mathematical formulation into a neural implementation, as assumptions made in the mathematical formulation may not hold true in the neural implementation.

Lastly, to further demonstrate the flexibility of the Spaun 2.0 model, three extensions to the Spaun 2.0 model were presented. These extensions gave Spaun an adaptive motor system, the visual system capable of processing colour images, and the ability to process generalized task instructions. These extensions also serve to demonstrate that the re-organized and re-implemented Spaun 2.0 network is flexible enough to serve as a test bed for alternative implementations of functional cognitive subsystems.

6.1 Contributions

Given its complexity and design, the Spaun model sits at the unique intersection of multiple areas of research. Analyzing Spaun using a top-down approach and considering it a carefully constructed collection of functional modules, Spaun's architectural design contains many features of modern computing hardware. As an example, the working memory system can be thought of as analogous to the cache or registers within a typical processor, while the transformation system is Spaun's equivalent to a processor's arithmetic logic unit. Using this approach categorizes Spaun within the realm of computer science and computer engineering.

However, it is equally valid to analyze the Spaun model from a bottom-up perspective; in which case, Spaun is simply a collection of simulated spiking LIF neurons that are interconnected using specifically determined connection weight matrices. The consequence of this perspective is the categorization of Spaun as part of the computational neuroscience research field.

This section briefly discusses the contributions of the Spaun model to the areas of computer science and computational neuroscience.

6.1.1 Contributions to Computer Science and Computer Engineering

This project's contributions to the fields of computer science and computer engineering are primarily related to the development of neuromorphic hardware. Unlike the binary-based logic used in traditional digital hardware, neuromorphic systems perform computations using electronic circuits designed to emulate the electrical behaviour of biological neurons (e.g., [Mead, 1990; Indiveri et al., 2011]). While traditional programming techniques cannot be used to program neuromorphic hardware, the design of neuromorphic hardware can offer benefits with regards to the efficiency in the use of silicon, and the energy consumption while performing computations [Mead, 1990; Stöckel et al., 2017].

On the topic of programming neuromorphic hardware, one difficulty in doing so is determining the necessary configuration of connections and weights needed to perform a specific function with the hardware. While machine learning techniques can be used to address this issue, the Neural Engineering Framework (see Section 2.5.4) provides a concise and efficient method for solving this problem. Since the components of Spaun described in Section 3.3 can be functionally implemented using the NEF, these components can be easily transferred onto neuromorphic hardware and can be used to construct the neuromorphic analogues of the components found in microprocessor architectures. In particular, Spaun's action selection hierarchy (see Section 3.3.8) demonstrates how the flow of information within a neural network can be controlled using a set of logical constructs (the condition-consequence pairs), similar to how processor instructions (opcodes) are used to control the flow of information within a microprocessor.

6.1.2 Contributions to Computational Neuroscience

With regards to the field of computational neuroscience, Spaun aims to serve as a modular test platform on which various biologically based algorithms can be trialled. By leveraging the modularity of the Spaun 2.0 model, alternative implementations of Spaun's various systems can be tested within the context of Spaun's existing known behaviours. Two such examples have already been discussed in this thesis; namely, the adaptive motor system (Section 5.1) and the ImageNet visual system (Section 5.2). Additionally, the Spaun model has been used to test the effects of more realistic implementations of neurons.

Eliasmith et al. [2016] describe a Spaun model in which the LIF neurons in part of Spaun's working memory system have been replaced with conductance-based compartmental neuron models of a pyramidal cell. The results of the work show that the behaviour of Spaun remains consistent with the LIF-neuron-based version, as well as making predictions on how the recall accuracy of the working memory task is negatively affected when a neurotoxin (tetrodotoxin (TTX)) is applied to the neurons. Because the simulated neurotoxin affects the behaviour of the neuron's ion channels, such a prediction would not have been possible using the purely LIF-neuron-based model, as the LIF neuron model used in Spaun (see Section 2.5.3) does not model the effects changing the behaviour of the ion channels.

6.2 Future Work

The complexity of the Spaun architecture provides many avenues for future work, some of which have already been briefly mentioned in the previous sections. This section explores some of these areas of future work, grouped according to their application to the different functional modules within Spaun.

6.2.1 Vision Module

The implementation of Spaun's visual system imposes the largest constraint on Spaun's ability to process incoming information. Because the visual hierarchies implemented thus far have been trained to classify each visual scene as a single concept, complex ideas (e.g., number lists) have to be decomposed into their constituent parts before they can be presented to Spaun. This makes it difficult to "program" Spaun to parse compound visual constructs (e.g., words, number lists, etc.), and necessitated the use of Spaun's special control characters to convey the necessary task information to Spaun.

Fully addressing this issue is a research topic in its own right, and would probably require a neural network with a complexity that is on par (or exceeds) that of Spaun. However, a proposed "first-step" would be to reduce Spaun's dependence on its control characters. Using the MNIST-based visual system as an example, a possible approach to reducing the need for control characters would be to implement an attention routing circuit (e.g., [Bobier, 2011]) to physically "parse" the visual field. As a proof-of-concept implementation, the location and size of each shift of the foveal area can be fixed (see Figure 6.1A). Additionally, specific areas of the entire visual field can be dedicated to the various input arguments required by each of Spaun's tasks (see Figure 6.1B).

Integrating the attention routing circuit with the rest of the Spaun network would also be relatively straightforward. Assuming the output dimensionality of the attention routing circuit is matched with the input dimensions of Spaun's visual hierarchy, the output of the visual attention circuit can be projected directly to the input of the visual hierarchy (see Figure 6.2). The most challenging part of the integration process would be the implementation of the control circuitry that is necessary to detect and determine various aspects of the visual scene. For example,

- Determining temporal events as part of the task progression (e.g., the start of the task – traditionally indicated with the "**A**" character; the prompt for a response – traditionally

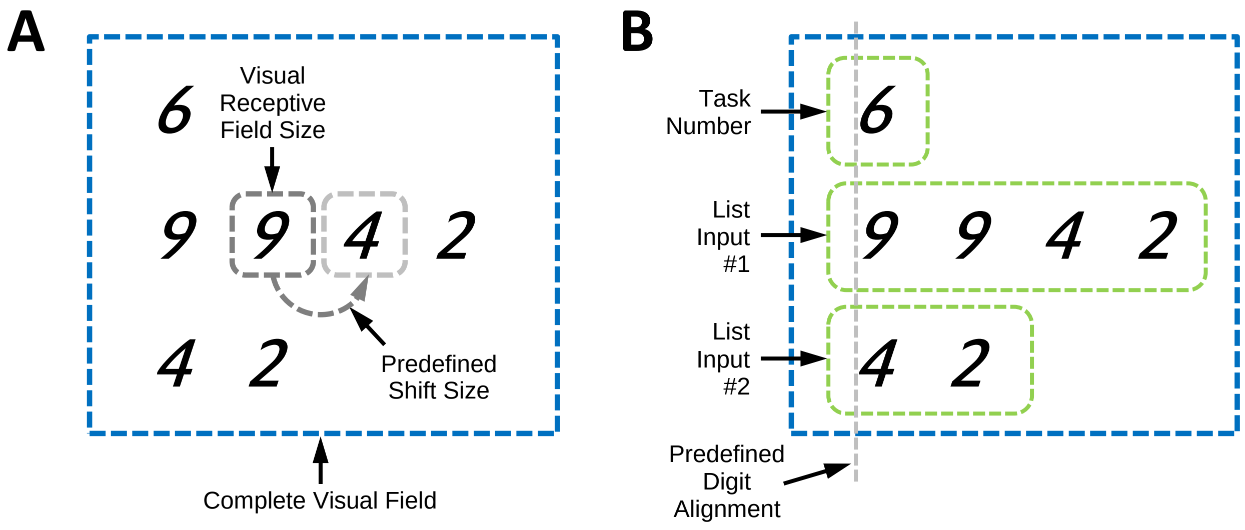


Figure 6.1: Layout of the visual field for Spaun's proposed visual attention routing circuit. (**A**) Proposed interaction between the visual attention routing circuit and visual stimulus. The attention routing circuit routes a predefined portion (receptive field) of the entire visual stimulus to the visual hierarchy. To simplify the initial implementation, shifts in the receptive field are proposed to be of a fixed size. (**B**) Proposed organization of Spaun's visual stimulus, using predefined areas to segregate task specific information. The organizational areas are aligned to a predetermined vertical coordinate to reduce the complexity of the proposed implementation.

indicated with the "**?**" character). A possible solution would be to have the routing circuit return back to a pre-determined location in the visual scene after processing the individual task elements. This location in the visual scene can then be used to initiate new tasks (by showing a task number), or to prompt for a response (by showing the question mark character).

- Determining the start and ends of input lists. A possible solution would be to fix the start of each list along the same vertical axis, and use a "blank" input to indicate the end of the list (see Figure 6.1B for an example).

6.2.2 Motor Module

Compared to Spaun's original implementation of the motor system, Spaun 2.0's implementation of the motor system was a marked improvement, with the improvements stemming from its realization as a fully spiking neural network. However, as described in Section 4.2.2, because

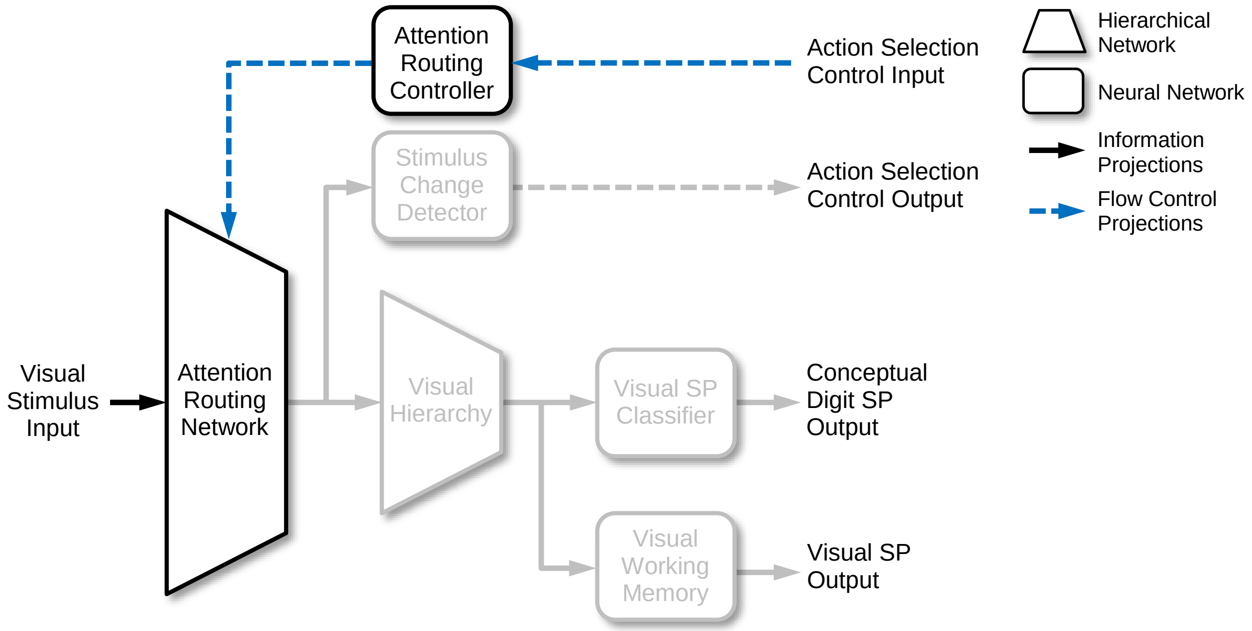


Figure 6.2: Proposed schematic of Spaun's visual system that includes an attention routing network. Existing Spaun networks have been grayed out to highlight the proposed additions.

Spaun 2.0's implementation of the motor hierarchy is missing the DMP system, it is not a complete REACH [DeWolf, 2014] model. Implementing the full REACH model would allow Spaun to reproduce effects like the ability to modify the scale and rotation of the output digits by changing a single parameter of the DMP system, as the Spaun simulation is running [DeWolf et al., 2016; DeWolf and Eliasmith, 2017]. Additionally, the use of the DMP system can produce "hybrid" digits that contain features of multiple "standard" digits (see Figure 6.3) [DeWolf and Eliasmith, 2017], or to combine multiple digit sequences with one fluid motion. This will open up possibilities for Spaun to produce more fluid handwriting, or allow for the implementation of tasks while require Spaun to be "creative" with its output (e.g., expressing a combination of concepts as a single output).

With the updates to the Nengo simulation software – which now has support for vector-based function representations [DeWolf and Eliasmith, 2017] – the most obvious avenue of future work is to adapt the DMP system to function within the Spaun ecosystem. Much of the infrastructure is already present, and in theory, all that is required is to use a basis-function evaluator network combined with the point attractor network to implement a Spaun-compatible DMP network, as illustrated in Figure 6.4. Several implementation details would have to be determined and tested, including:

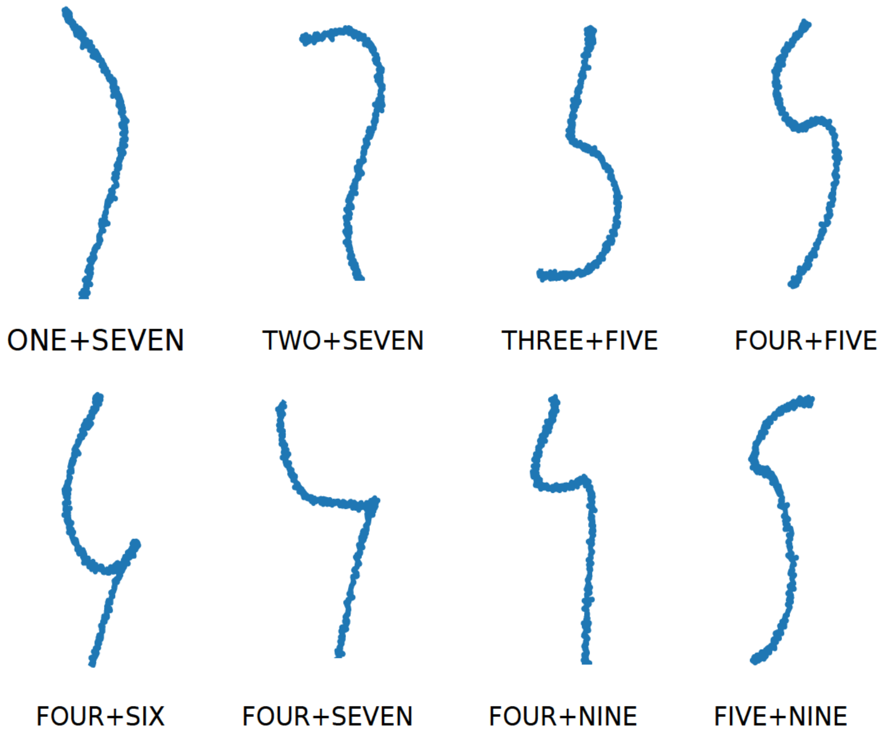


Figure 6.3: Illustration of "compound" digits produced by the REACH model. Each "compound" digit is produced by combining features of the numerical digits listed below each diagram. (Reproduced from [DeWolf and Eliasmith, 2017] with permission)

- Determining an appropriate solution to generating the "target point", which is necessary for the correct operation of the point attractor network, and is specific to each type of digit (i.e., numerical value) being written.
- Testing the robustness of using the vector representation of each digit's specific forcing function when used with the algorithm for generating the copy-drawing transformation matrices. Thus far, the algorithm has proved successful with trajectory-based vector representations, but it is unknown if this will be the case with function-based representations.

6.2.3 Information Encoding, Decoding, Working Memory Modules

While the implementation of the working memory system (both the original OSE implementation [Choo, 2010], and the updated Spaun 2.0 implementation) has proved to be effective at reproducing the recall behaviours observed in human experiments, it is also to some extent,

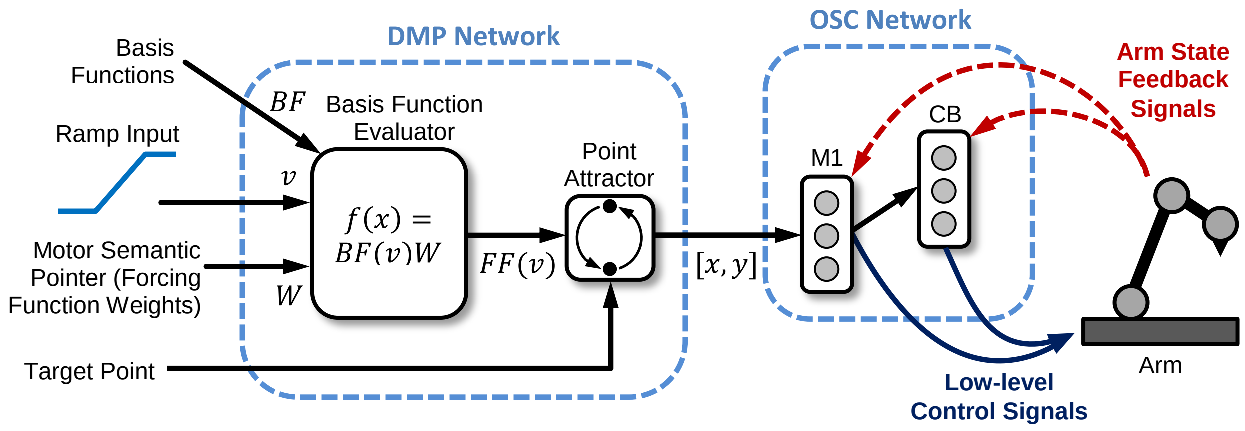


Figure 6.4: Schematic of Spaun's proposed motor system implemented using the DMP network. In this network, a sub-network is used to combine a set of basis functions (BF) with a vector-based representation of the desired forcing function (W). The output of the forcing function ($FF(v)$) is computed by taking the inner product of the motor semantic pointer (W) with the basis functions evaluated at the value determined by the ramp input (i.e., $BF(v)W$).

unsatisfactory. This is primary because Spaun's implementation of the working memory system approximates the rehearsal process simply as a scaling factor on the inputs to the working memory networks (see Section 3.1.3).

Since Spaun is a fully enclosed cognitive architecture, it should be possible to use the existing encoding and decoding networks to create a physical rehearsal process – in which list items are recalled, and subsequently re-encoded back into memory. Combined with an implementation of the hippocampal network (e.g. [Trujillo, 2014]), the physical rehearsal mechanism may provide Spaun the ability to reproduce memory effects that the current model cannot reproduce (e.g., the suppression of the primacy effect when the rehearsal process is interrupted).

6.2.4 Transformation Module

Spaun's transformation module forms the heart of its "cognitive" ability, as it contains all of the networks (e.g., binding network, associative memory networks, etc.) necessary to perform the SPA computations needed to generate the correct answers to Spaun's tasks. However, the design of the transformation module (in particular, the projections to the selector networks) is currently determined by the computations necessary to complete Spaun's tasks. As an example, in Figures 3.33 and 3.34, "Input Selector 2" only receives inputs from "∼*mem*1", "*mem*2", and "*ave_mem*". With the addition of the instruction processing module, the limited connectivity constrains the types of computation that can be performed (e.g., computing "$mem2 \circledast \sim mem1$" is not possible).

To address this issue, additional projections need to be added to the transformation module's network. However, projections must not be added frivolously, and the configuration of the projections should maximize the general use capability of the network while minimizing the additional resources (i.e., neuron count) required.[46]

Additionally, there are currently no projections from the output of the transformation module to the inputs of the memory network. This means that computational products produced by the transformation module (e.g., computing the result of applying an induction transform to a value stored in memory) cannot be transferred back into memory for future use. Implementing these projections (along with the necessary additions to the action selection hierarchy) will provide Spaun with the ability to perform more interesting tasks (e.g., performing the question answering task on the products of an induction task).

6.2.5 Instruction Processing Module

As stated in Section 5.3, the instruction processing system is still very much a work-in-progress, and several improvements can be made to the network. First and foremost, the method by which the instruction semantic pointer is constructed and provided to Spaun is somewhat artificial, as it requires the entity constructing the instruction representation to have in-depth knowledge of the implementation details of Spaun's action selection hierarchy. Additionally, it requires Spaun to receive an input that is already processed as a semantic pointer, deviating from Spaun's original goal of having the semantic pointers be internally generated. The proposed solution to this problem is two-fold: the inclusion of a sentence parsing system (e.g., [Stewart et al., 2014], [Blouw and Eliasmith, 2015]), and the use of a hierarchical associative mapping.

Assuming that the Spaun is given the ability to process words as individual concepts – either through upgrading the visual system (see Section 6.2.1), or by some other form of stimulus input processing (e.g., audition [Bekolay, 2016]) – a sentence parsing system would provide Spaun with the necessary mechanism to convert input stimuli into semantic pointer representations, thus achieving the goal of having only internally generated semantic pointer representations. However, having just the sentence parsing system is probably insufficient as it is highly unlikely that the syntax and semantics of the sentence conform to the representations required by the encoding schema of the instruction processing system (e.g., "Perform the working memory task"

[46]Assuming the selector networks use ensemble arrays to gate the semantic pointer representations, each additional projection to a selector network adds an additional $D \times n$ neurons. In Spaun, D is 512, while n is typically 50, totalling to a not insignificant $25,600$ neurons.

does not logically parse as the semantic pointer "$(\mathsf{\textbf{TASK}} \circledast \mathsf{\textbf{WM}})\mathbf{P}_{cons}$"). As such, some sort of associative memory network – perhaps even requiring a hierarchy of associative memory networks – is needed to convert the output of the sentence parsing network into a form the instruction processing system can decode. It should be noted that the proposed solution is purely speculative, and other solutions may be more feasible.

In addition to the issue regarding the construction of the instruction semantic pointer, the instruction processing system itself has limitations. As demonstrated in Figure 5.26, one of these limitations is its sensitivity to the amount of information encoded in the instruction semantic pointer. Another limitation is the inconsistent support of compound conditionals. In the action selection hierarchy, compound conditionals are implemented using a weighted sum of the conditions. However, in the instruction processing system, using a weighted sum would involve adjusting the cleanup threshold of the list position memory, which may adversely impact its ability to accurately extract information from the instruction semantic pointer. It may be possible to address these limitations using a different encoding schema, or a different design to the instruction processing network. However, these conclusions are difficult to reach without further investigation and testing of these implementations.

6.2.6 Reward Evaluation Module

The reward evaluation system in Spaun is currently limited to modifying only the condition-consequence pairs used by Spaun's learning task (n-arm bandit task). A possible area of future work would be to integrate the reward evaluation module with the instruction processing system to provide Spaun with the ability to learn new tasks (i.e., converting the actions generated by the output of the instruction processing system into physical changes in the condition-consequence pairs implemented in the action selection hierarchy). Using the counting task as an example, Aubin et al. [2016] demonstrate that task learning is possible in models built using the SPA and NEF.

6.2.7 Action Selection Module

As detailed in Section 3.3.8, the design and implementation of Spaun's action selection module was determined by the logical flow of Spaun's original tasks. However, with the addition of the instruction processing system, several improvements can be made to the action selection module.

With regards to the types of instructions that can be processed by the instruction processing system (see Section 5.3.7), it was noted that the tasks contained in an instruction must already exist as a condition-consequence pair in the action selection hierarchy. This limits the scope of the instruction processing system, as some tasks cannot be accomplished. As an example, in the current Spaun 2.0 model, it is not possible to instruct it to use the product of an induction task as the input to another task. This is because none of Spaun's original tasks require this function, and thus, the condition-consequence pair (and the necessary projections) have not been implemented. A potential solution to this issue is to construct a set of general use condition-consequence pairs, specifically to allow the routing of information between Spaun's different modules. Examples of these condition-consequence pairs include:

- $task\_stage \bullet \textsf{\textbf{MEM1\_MEM2}} \longmapsto mem1 \Rightarrow mem2$
- $task\_stage \bullet \textsf{\textbf{TRFM\_MEM3}} \longmapsto transform\_out \Rightarrow mem3$

It should be noted that while it is straightforward to enumerate the different combinations of routes that can be implemented, the difficulty lies in structuring the condition-consequence pairs in order to minimize the number and complexity of additional pairs required – so as to reduce the potential for conflicts with existing condition-consequence pairs.

Another proposed modification to the action selection hierarchy involves the division of the cortico-basal ganglia-thalamus loop into (semi-)independent loops. In the current Spaun architecture, the action selection hierarchy is constructed as a monolithic loop, where the basal ganglia network computes an approximate "winner-take-all" function for *all* of the condition-consequence pairs. However, with the proposed addition of functionality it may be beneficial to have multiple independent basal ganglia networks, each dedicated to processing functionally independent tasks. As an example, in addition to the "main" loop, an independent loop can be constructed to handle vision related processing (e.g., direction of the attention routing circuit), while another loop can be used to handle working memory related processing (e.g., the rehearsal mechanism). It should be noted that while multiple cortico-basal ganglia-thalamus loops seem like a logical improvement of the action selection hierarchy, there is some evidence to suggest that functional segregation of the cortico-basal ganglia-thalamus loop also exists in the primate brain (e.g., [Alexander et al., 1986; Haber, 2003]).

6.2.8 Code Base

Compared to its predecessor, Spaun 2.0's code base has been re-organized with the focus on creating clear delineations between the different functional components of Spaun. This has reduced the overhead required to allow implementation variants of functional Spaun modules (e.g., the ImageNet vision system described in Section 5.2, and the adaptive motor system described in Section 5.1). However, the necessary changes to the code base to allow these modules to be easily interchanged have only been applied to the vision and motor system. To fulfill Spaun's goal of being a test bed for various system implementations, these changes have to be propagated to the remainder of Spaun's systems.

6.3 Closing Remarks

In Choo [2010] it was noted that with the OSE model, the purely mathematical implementation of the model failed to reproduce the list recall curves observed in the human data. Rather, it was the dynamics of the neural implementation (hypothesized to be the saturation effect of the neurons) that were, for the most part, responsible for the model's ability to reproduce the characteristic U-shape curve observed in human recall data. This remains the case in Spaun, whereby a purely mathematical implementation would not be able to reproduce the variety of effects seen in the model's behaviour.

Figure 6.5 illustrates one such example, whereby the reduction of the presentation interval of each list digit from 0.571s (which matches the experimental setup of the results in Figure 4.24) to 0.150s (which is the "default" Spaun presentation interval – see Section 4.3) results in the suppression of the recency effect in the recall data. It is hypothesized that between the two experimental configurations, this is due to the differing amount of time the representations have had to decay. A purely mathematical implementation (i.e., using just the SPA operators) of Spaun would not have been able to produce such an effect, once again illustrating the importance of the neural implementation of the model. It remains to be seen if such an effect is reproduced in human experiments and serves as a prediction proposed by the Spaun 2.0 model.

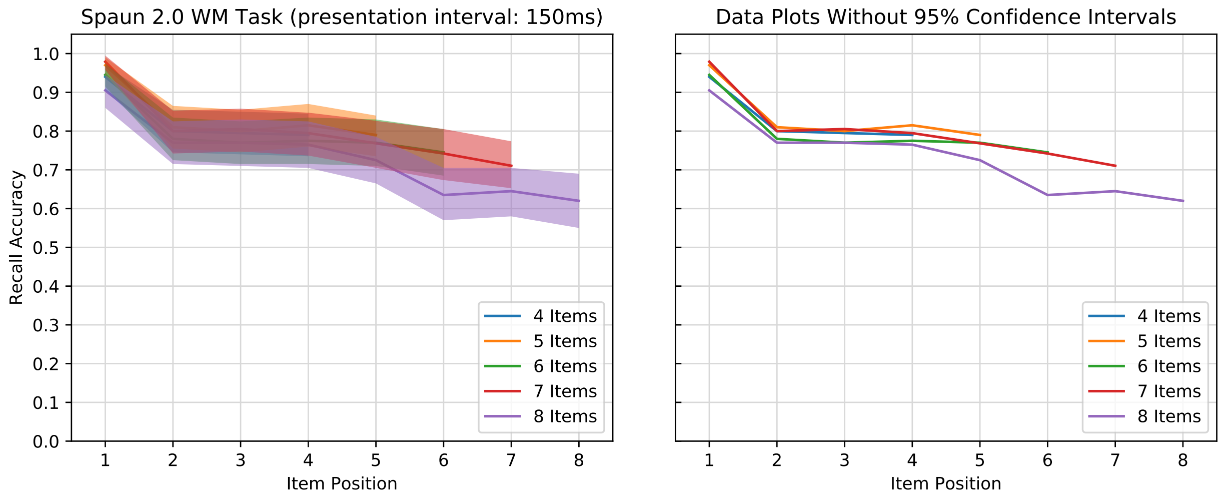


Figure 6.5: Forward recall accuracy for the working memory task with a digit presentation interval of 150ms. The data is plotted both with (left) and without (right) the 95% confidence intervals.

In addition to the importance of neural dynamics, Spaun also illustrates the importance that the network dynamics have to play in reproducing behavioural data observed in human experiments. This is evident in the results obtained for the stimulus-response tasks, where the overall response time of the Spaun 2.0 model only matched human performance when the processing time for both the visual and motor systems are taken into account (see Figure 5.17 and 5.20). Additionally, in Figure 5.20, the difference in reaction times between the non-instructed and instructed variants of the stimulus-response times can be attributed to the additional processing time introduced by the propagation of information through the instruction processing system.

Large-scale integrated neural models present an effective way of studying how neural dynamics and network dynamics affect behaviour on cognitive tasks. Importantly, these large-scale models also permit the exploration of the complex interplay between the low-level neural and high-level network dynamics. The Spaun architecture and model provides ample opportunity to characterize these dynamics in more detail, and the work presented in this thesis lays the foundation for continued extensions to the world's largest functional brain model.

References

Alexander, G. E., DeLong, M. R., and Strick, P. L. (1986). Parallel organization of functionally segregated circuits linking basal ganglia and cortex. *Annual review of neuroscience*, 9(1):357–381.

Anderson, J. R. (1996). Act: A simple theory of complex cognition. *American Psychologist*, 51(4):355.

Anderson, J. R., John, B. E., Just, M. A., Carpenter, P. A., Kieras, D. E., and Meyer, D. E. (1995). Production system models of complex cognition. In *Proceedings of the seventeenth annual conference of the cognitive science society*, volume 17, page 9. Psychology Press.

Anderson, J. R. and Lebiere, C. J. (2014). *The atomic components of thought.* Psychology Press.

Aubin, S., Voelker, A. R., and Eliasmith, C. (2016). Improving with practice: A neural model of mathematical development. *Topics in Cognitive Science.*

Bekolay, T. (2011). Learning in large-scale spiking neural networks. Master's thesis, University of Waterloo, Waterloo, ON.

Bekolay, T. (2016). *Biologically inspired methods in speech recognition and synthesis: closing the loop.* Phd thesis, University of Waterloo.

Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T. C., Rasmussen, D., Choo, X., Voelker, A., and Eliasmith, C. (2014). Nengo: a python tool for building large-scale functional brain models. *Frontiers in neuroinformatics*, 7:48.

Blouw, P. and Eliasmith, C. (2015). Constraint-based parsing with distributed representations. In *37th Annual Conference of the Cognitive Science Society*, pages 238–243.

Bobier, B. (2011). *The Attentional Routing Circuit: A Neural Model of Attentional Modulation and Control of Functional Connectivity.* Phd thesis, University of Waterloo, Waterloo, ON.

Carnevale, N. T. and Hines, M. L. (2006). *The NEURON book.* Cambridge University Press.

Chaaban, I. and Scheessele, M. R. (2007). Human performance on the usps database. *Report, Indiana University South Bend.*

Choo, X. (2010). The ordinal serial encoding model: Serial memory in spiking neurons. Master's thesis, University of Waterloo, Waterloo, ON.

Choo, X. and Eliasmith, C. (2013). General instruction following in a large-scale biologically plausible brain model. In *35th Annual Conference of the Cognitive Science Society*, pages 322–327. Cognitive Science Society.

DeFelipe, J. and Jones, E. G. (1988). *Cajal on the Cerebral Cortex: An Annotated Translation of the Complete Wrings.* Oxford Unive. Press.

DeWolf, T. (2010). Noch: A framework for biologically plausible models of neural motor control. Masters thesis, University of Waterloo, Waterloo, ON.

DeWolf, T. (2014). *A neural model of the motor control system.* Phd thesis, University of Waterloo.

DeWolf, T. and Eliasmith, C. (2017). Trajectory generation using a spiking neuron implementation of dynamic movement primitives. *27th Annual Meeting for the Society for the Neural Control of Movement.*

DeWolf, T., Stewart, T. C., Slotine, J.-J., and Eliasmith, C. (2016). A spiking neural model of adaptive arm control. *Proceedings of the Royal Society B*, 283(48).

Dosher, B. A. (1999). Item interference and time delays in working memory: Immediate serial recall. *International Journal of Psychology*, 34(5-6):276–284.

Eliasmith, C. (2013). *How to build a brain: A neural architecture for biological cognition.* Oxford University Press, New York, NY.

Eliasmith, C. and Anderson, C. H. (2003). *Neural engineering: computation, representation, and dynamics in neurobiological systems.* The MIT Press, Cambridge, MA.

Eliasmith, C., Gosmann, J., and Choo, X.-F. (2016). Biospaun: A large-scale behaving brain model with complex neurons. *ArXiv.*

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

Eliasmith, C. and Trujillo, O. (2013). The use and abuse of large-scale brain models. *Current Opinion in Neurobiology*, 25:1–6.

Elston, G. N. (2003). Cortex, cognition and the cell: New insights into the pyramidal neuron and prefrontal function. *Cerebral Cortex*, 13(11):1124–1138.

Forbes, A. R. (1964). An item analysis of the advanced matrices. *British Journal of Educational Psychology*, 34(3):223–236.

Georgopoulos, A. P., Schwartz, A., and Kettner, R. E. (1986). Neuronal population coding of movement direction. *Science*, 233:1416–1419.

Goldman-rakic, P. S. (1995). Cellular basis of working memory. *Neuron*, 14:477–485.

Grice, G. R., Nullmeyer, R., and Spiker, V. A. (1982). Human reaction time: toward a general theory. *Journal of Experimental Psychology: General*, 111(1):135.

Gurney, K., Prescott, T. J., and Redgrave, P. (2001). A computational model of action selection in the basal ganglia. i. a new functional anatomy. *Biological Cybernetics*, 84:401–410.

Haber, S. N. (2003). The primate basal ganglia: parallel and integrative networks. *Journal of chemical neuroanatomy*, 26(4):317–330.

Hadley, R. (2009). The problem of rapid variable creation. *Neural Computation*, 21(2):510–532.

Henson, R. N. A., Burgess, N., and Frith, C. D. (2000). Recoding, storage, rehearsal and grouping in verbal short-term memory: an fmri study. *Neuropsychologia*, 38:426–440.

Hunsberger, E. and Eliasmith, C. (2015). Spiking deep networks with lif neurons. *arXiv:1510.08829*.

Hunsberger, E. and Eliasmith, C. (2016). Training spiking deep networks for neuromorphic hardware. *arXiv:1611.05141*.

Hwang, K., Bertolero, M. A., Liu, W. B., and D'Esposito, M. (2017). The human thalamus is an integrative hub for functional brain networks. *Journal of Neuroscience*, 37(23):5594–5607.

Indiveri, G., Linares-Barranco, B., Hamilton, T. J., Van Schaik, A., Etienne-Cummings, R., Delbruck, T., Liu, S.-C., Dudek, P., Häfliger, P., Renaud, S., et al. (2011). Neuromorphic silicon neuron circuits. *Frontiers in neuroscience*, 5:73.

Kanerva, P. (1996). Binary spatter-code of ordered k-tuples. In von der Malsburg, C., vonSeelen, W., C., V. J., and Sendhoff, B., editors, *Artificial Neural Networks – ICANN Proceedings*, volume 1112 of *Lecture Notes in Computer Science*, pages 869–873, Berlin. Springer.

Kim, H., Sul, J. H., Huh, N., Lee, D., and Jung, M. W. (2009). Role of striatum in updating values of chosen actions. *Journal of neuroscience*, 29(47):14701–14712.

Kretchmar, R. M. (2002). Parallel reinforcement learning. In *The 6th World Conference on Systemics, Cybernetics, and Informatics*. Citeseer.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Kropotov, J. D. and Etlinger, S. C. (1999). Selection of actions in the basal ganglia–thalamocortical circuits: Review and model. *International Journal of Psychophysiology*, 31(3):197–217.

Krueger, L. E. (1989). Psychophysical law: Keep it simple. *Behavioral and Brain Sciences*, 12(2):299–320.

Laird, J. E. (2012). *The Soar cognitive architecture*. MIT press.

Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial intelligence*, 33(1):1–64.

Landauer, T. K. (1962). Rate of implicit speech. *Perceptual and motor skills*.

Le Gros Clark, W. E. (1932). The structure and connections of the thalamus. *Brain*, 55(3):406–470.

Markram, H. (2006). The blue brain project. *Nature Reviews Neuroscience*, 7:153–160.

McCormick, D. A., Connors, B. W., Lighthall, J. W., and Prince, D. A. (1985). Comparative electrophysiology of pyramidal and sparsely spiny stellate neurons of the neocortex. *Journal of Neurophysiology*, 54(4):782–806.

Mead, C. (1990). Neuromorphic electronic systems. *Proceedings of the IEEE*, 78(10):1629–1636.

Meyer, D. E. and Kieras, D. E. (1997). A computational theory of executive cognitive processes and multiple-task performance: Part i. basic mechanisms. *Psychological review*, 104(1):3.

Modha, D. S., Ananthanarayanan, R., Esser, S. K., Ndirango, A., Sherbondy, A. J., and Singh, R. (2011). Cognitive computing. *Communications of the ACM*, 54(8):62–71.

Plate, T. A. (1994). Estimating analogical similarity by dot-products of holographic reduced representations. In *Advances in neural information processing systems*, pages 1109–1116.

Plate, T. A. (2003). *Holographic reduced representations: distributed representations for cognitive structures.* CSLI Publications, Stanford, CA.

Rasmussen, D. (2010). A neural modelling approach to investigating general intelligence. Masters thesis, University of Waterloo, Waterloo, ON.

Rauch, A., La Camera, G., Luscher, H.-R., Senn, W., and Fusi, S. (2003). Neocortical pyramidal cells respond as integrate-and-fire neurons to in vivo-like input currents. *Journal of Neurophysiology*, 90(3):1598–1612.

Redgrave, P., Prescott, T. J., and Gurney, K. (1999). The basal ganglia: a vertebrate solution to the selection problem? *Neuroscience*, 89(4):1009–1023.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252.

Singh, R. and Eliasmith, C. (2006). Higher-dimensional neurons explain the tuning and dynamics of working memory cells. *Journal of Neuroscience*, 26:3667–3678.

Smolensky, P. (1990). Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence*, 46:159–216.

Stewart, T. C., Bekolay, T., and Eliasmith, C. (2012). Learning to select actions with spiking neurons in the basal ganglia. *Frontiers in Decision Neuroscience*, 6.

Stewart, T. C., Choo, X., and Eliasmith, C. (2010). Symbolic reasoning in spiking neurons: A model of the cortex/basal ganglia/thalamus loop. In Ohlsson, S. and Catrambone, R., editors, *32nd Annual Meeting of the Cognitive Science Society*, pages 1100–1105, Portland, Oregon. Cognitive Science Society.

Stewart, T. C., Choo, X., and Eliasmith, C. (2014). Sentence processing in spiking neurons: A biologically plausible left-corner parser. In *36th Annual Conference of the Cognitive Science Society*, pages 1533–1538. Cognitive Science Society.

Stewart, T. C., Tang, Y., and Eliasmith, C. (2011). A biologically realistic cleanup memory: Autoassociation in spiking neurons. *Cognitive Systems Research*, 12(2):84–92.

Stewart, T. C., Tripp, B., and Eliasmith, C. (2009). Python scripting in the nengo simulator. *Frontiers in Neuroinformatics*, 3.

Stöckel, A., Jenzen, C., Thies, M., and Rückert, U. (2017). Binary associative memories as a benchmark for spiking neuromorphic hardware. *Frontiers in Computational Neuroscience*, 11:71.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

Tang, Y. and Eliasmith, C. (2010). Deep networks for robust visual recognition. In *Proceedings of the 27th International Conference on Machine Learning, June 21-24, 2010, Haifa, Israel.*

Tripp, B. and Eliasmith, C. (2010). Population models of temporal differentiation. *Neural Computation*, 22:621–659.

Trujillo, O. (2014). A spiking neural model of episodic memory encoding and replay in hippocampus. Masters thesis, University of Waterloo, Waterloo, Ontario.

Uno, M., Yoshida, M., and Hirota, I. (1970). The mode of cerebello-thalamic relay transmission investigated with intracellular recording from cells of the ventrolateral nucleus of cat's thalamus. *Experimental brain research*, 10(2):121–139.

Vives, F. and Mogenson, G. J. (1985). Electrophysiological evidence that the mediodorsal nucleus of the thalamus is a relay between the ventral pallidum and the medial prefrontal cortex in the rat. *Brain Research*, 344(2):329–337.

Voelker, A. R. and Eliasmith, C. (2015). Computing with temporal representations using recurrently connected populations of spiking neurons. In *Connecting Network Architecture and Network Computation.* Banff International Research Station for Mathematical Innovation and Discovery.

Appendix A

Mathematical Illustrations and Derivations

A.1 Convolution vs. Circular Convolution

For two N-dimensional vectors $\mathbf{X}$ and $\mathbf{Y}$, each vector element z_n of the result of the convolution of the two vectors ($\mathbf{Z} = \mathbf{X} * \mathbf{Y}$) is computed by:

$$z_n = \sum_{k=-N}^{N} x_k y_{n-k}$$

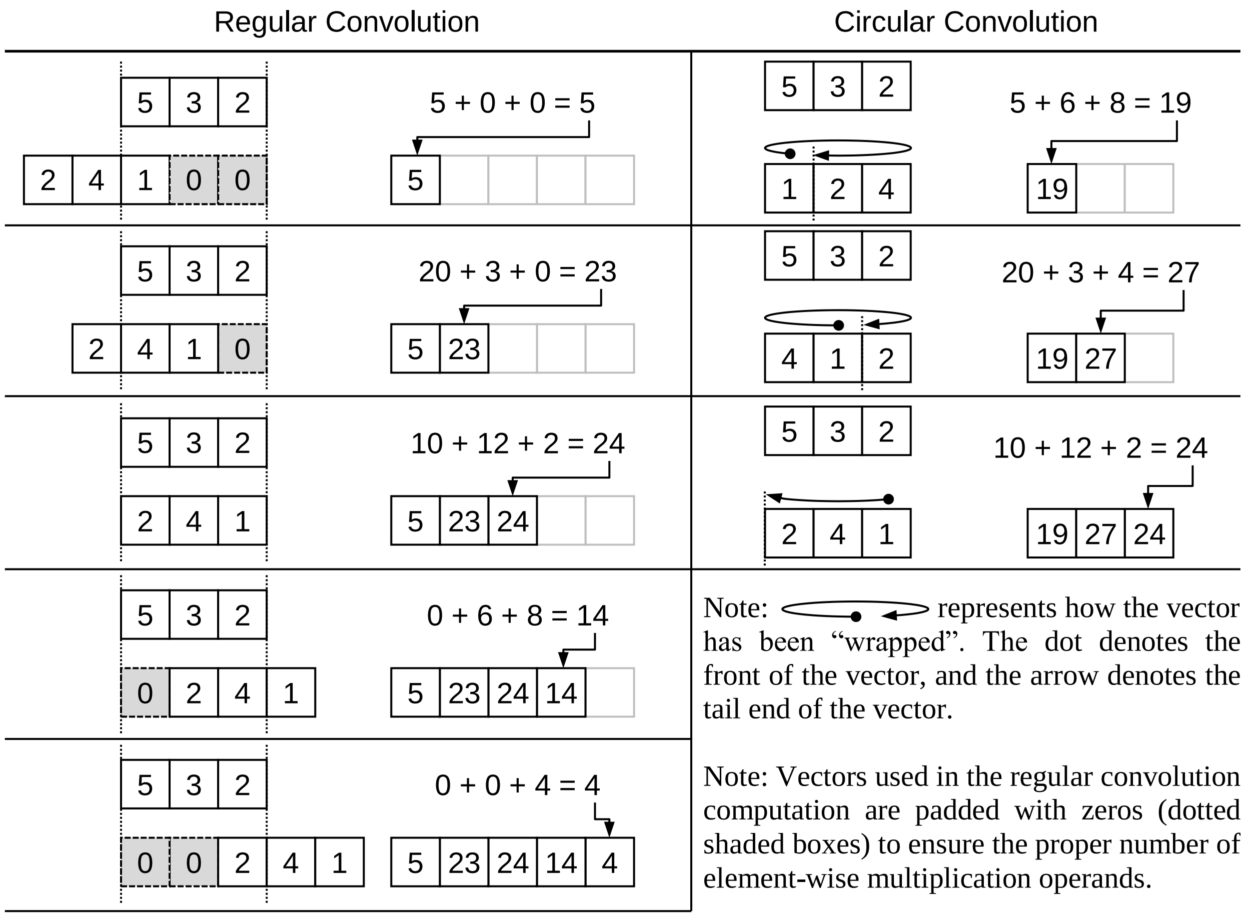
Likewise, each vector element c_n of the result of the circular convolution of the two vectors ($\mathbf{C} = \mathbf{X} \circledast \mathbf{Y}$) is computed by:

$$c_n = \sum_{k=-N}^{N} x_k y_{n-k}, \text{ where the subscripts are modulo-}N$$

Given the similarity of convolution operations, both follow similar steps to perform the computation. First, the second vector operand ($\mathbf{Y}$) is flipped. Next, the flipped vector is "slid" past the first vector operand, and each vector element of the convolution product is calculated as the summation of the element-wise products of the vectors. The only difference between the circular and non-circular variant of the convolution is that in the circular convolution, the elements in the second vector operand (i.e., the flipped vector) wraps around.[47]

[47]In the circular convolution operation, the elements of the second vector operand can also be thought of as

Figure A.1 shows a visual representation of the difference between the non-circular and circular variants of the convolution operation.



Note: represents how the vector has been "wrapped". The dot denotes the front of the vector, and the arrow denotes the tail end of the vector.

Note: Vectors used in the regular convolution computation are padded with zeros (dotted shaded boxes) to ensure the proper number of element-wise multiplication operands.

Figure A.1: A visual illustration of the difference between the convolution and circular convolution operators. For both convolution computations, two 3-dimensional vectors ($[5, 3, 2]$ and $[1, 4, 2]$) are convolved. Refer to text for additional details. (Figure adapted from [Choo, 2010] with permission)

repeating with a period of N.

A.2 HRR-SPA Approximate Inverse Operator

The HRR-SPA approximate inverse operator ($\sim$) for a semantic pointer A is:

$$\sim\mathsf{A} = \mathcal{F}^{-1}\left(\overline{\mathcal{F}(\mathsf{A})}\right), \tag{A.1}$$

where $\mathcal{F}$ is the Fourier transform operator, $\mathcal{F}^{-1}$ is the inverse Fourier transform operator, and $\overline{z}$ is the complex conjugate of the complex number z.

For vector representations, the Fourier transform is computed using the discrete Fourier transform (DFT). Performing the DFT on an N-dimensional vector $\mathbf{x} = [x_0, x_1, \ldots, x_{N-1}]$, results in the vector of complex numbers $\mathbf{X} = [X_0, X_1, \ldots, X_{N-1}]$. Each vector element X_k in $\mathbf{X}$ is computed as:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i k n}{N}}$$

Performing the complex conjugate on X_k results in:

$$\overline{X_k} = \sum_{n=0}^{N-1} x_n e^{\frac{2\pi i k n}{N}}$$

Thus, computing the complex conjugate of the result of the DFT of $\mathbf{x}$ is the complex vector $\mathbf{Y}$, where each vector element Y_k is equal to the vector element X_{-k}. I.e.,

$$\begin{aligned}
\mathbf{Y} &= \overline{DFT(\mathbf{x})}, \text{ where,} \\
Y_k &= \sum_{n=0}^{N-1} x_n e^{\frac{2\pi i k n}{N}} \\
&= \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i (-k) n}{N}} \\
&= X_{-k}
\end{aligned}$$

This means that the complex vector $\mathbf{Y}$ can be computed by permuting the complex vector $\mathbf{X}$ using the permutation matrix $\mathbf{L}$:

$$\mathbf{Y} = \mathbf{X}\mathbf{L},$$

where $\mathbf{L}$ is the matrix:

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 \\ 0 & 0 & 0 & \dots & 1 & 0 \\ \vdots & \vdots & \vdots & \iddots & \vdots & \vdots \\ 0 & 0 & 1 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \end{bmatrix} \tag{A.2}$$

With Eq. (A.2), Eq. (A.1) can be re-written as:

$$\begin{aligned} \sim\mathbf{A} &= IDFT\left(\overline{DFT(\mathbf{A})}\right) \\ &= IDFT\left(DFT(\mathbf{A})\mathbf{L}\right) \end{aligned} \tag{A.3}$$

Because the discrete Fourier transform operations are linear matrix transformations, Eq. (A.3) can be re-arranged into this form:

$$\begin{aligned} \sim\mathbf{A} &= IDFT\left(DFT(\mathbf{A})\right)\mathbf{L} \\ &= \mathbf{AL}, \end{aligned}$$

which is the result of applying the permutation matrix $\mathbf{L}$ to the HRR-SPA semantic pointer $\mathbf{A}$.

A.3 Compound *NOT* Conditional Statements

For the truth table discussed in this section, several assumptions are made:

- Only "full" (e.g., full: "$\mathbf{A}$" versus partial: "$0.5 \times \mathbf{A}$") semantic pointers are used for the "*state*" representations.
- The semantic pointers used are roughly orthogonal to each other. I.e., $\mathbf{A} \bullet \mathbf{B} \approx 0$.
- The semantic pointer $\mathbf{X}$ is used to represent a *state* representation that is not part of the conditional statement.
- The result of the conditional computation is only considered "active" if it evaluates to a value greater than 0.5. Active conditional evaluations will be indicated with a (*).

Conditional statements that compound multiple conditions are usually constructed by equally weighting the contributions of each condition. As an example, the conditional statement for "*state*1 is $\mathbf{A}$ *AND* *state*2 is $\mathbf{B}$" is:

$$(0.5 \times state1 \bullet \mathbf{A} + 0.5 \times state2 \bullet \mathbf{B})$$

For this conditional statement, the only situation in which it is desired to be active is when *both* $state1$ is **A** and $state2$ is **B**. Completing the truth table for the four combinations of values that $state1$ and $state2$ reveals that this is the case:

$state1$	$state2$	$(0.5 \times state1 \bullet \mathbf{A} + 0.5 \times state2 \bullet \mathbf{B})$
X	**X**	$0 + 0 = 0$
A	**X**	$0.5 + 0 = 0.5$
X	**B**	$0 + 0.5 = 0.5$
A	**B**	$0.5 + 0.5 = 1^*$

Conditional statements for the singular *NOT* condition are constructed by using a subtraction from 1. As an example, the conditional statement for "$state1$ is *NOT* **A**" is:

$$(1 - state1 \bullet \mathbf{A})$$

Once again, completing the truth table for this conditional statement reveals that it is only "active" when the $state1$ representation is any other value but **A**:

$state1$	$(1 - state1 \bullet \mathbf{A})$
X	$1 - 0 = 1^*$
A	$1 - 1 = 0$

Constructing compound conditional statements using the *NOT* condition may be unintuitive. Intuitively, the conditional statement for "$state1$ is **A** *AND* $state2$ is *NOT* **B**" is assembled by combining the weighting used for the *AND* condition, and the subtraction from 1 for the *NOT* condition, resulting in:

$$(0.5 \times state1 \bullet \mathbf{A} + 0.5 \times (1 - state2 \bullet \mathbf{B}))$$

Completing the truth table reveals that this conditional statement is only active for the desired state representations (only when $state1$ is **A** and $state2$ is **X**):

$state1$	$state2$	$(0.5 \times state1 \bullet \mathbf{A} + 0.5 \times (1 - state2 \bullet \mathbf{B}))$
X	**X**	$0 + 0.5 = 0.5$
A	**X**	$0.5 + 0.5 = 1^*$
X	**B**	$0 + 0 = 0$
A	**B**	$0.5 + 0 = 0.5$

While the conditional statement above does perform the intended computation, it is possible to construct a conditional statement that *only* produces a positive output when $state1$ is $\mathsf{\textbf{A}}$ and $state2$ is $\mathsf{\textbf{X}}$. This may be a desired property to reduce the amount of potential competition within the basal ganglia network. The conditional statement for this scenario is:

$$(state1 \bullet \mathsf{\textbf{A}} - (state2 \bullet \mathsf{\textbf{B}}))$$

Completing the truth table for this conditional statement demonstrates that the statement is only active when desired and, importantly, produces a positive value only in this active state.

$state1$	$state2$	$(state1 \bullet \mathsf{\textbf{A}} - (state2 \bullet \mathsf{\textbf{B}}))$
X	**X**	$0 - 0 = 0$
A	**X**	$1 - 0 = 1^*$
X	**B**	$0 - 1 = -1$
A	**B**	$1 - 1 = 0$

Appendix B

Spaun Implementation Details

B.1 Source Code

The source code for the original Spaun model is available online at
`http://models.nengo.ca/spaun`.

The source code for the Spaun 2.0 model is available online at
`https://github.com/xchoo/spaun2.0`.

B.2 List of the Condition-consequence Pairs in Spaun 2.0

This appendix lists all of the condition-consequence pairs used in the Spaun2.0 model to handle the task processing logic for each one of Spaun's tasks (including the Spaun2.0 extensions). For brevity, the following nomenclature substitutions have been made:

Task State Name (see Section 3.3.8.1)	Substituted Name
task	*task*
task_stage	*stage*
task_decode	*dec*
vision	*vis*
instr_data (Section 5.3.3)	*ins_data*
instr_task (Section 5.3.5)	*ins_task*
instr_task_stage (Section 5.3.5)	*ins_stage*
instr_task_decode (Section 5.3.5)	*ins_dec*

Task Semantic Pointer (see Table 3.5)	Substituted Semantic Pointer
INIT	**X**
COPY_DRAW	**W**
RECOG	**R**
LEARN	**L**
WM	**M**
COUNT	**C**
QA	**Q**
RVC	**V**
FLUID_IND	**F**
DEC	**DEC**
COMPARE (Section 5.2.2)	**CMP**
RESPONSE (Section 5.3.2)	**RESP**
INSTR_RESP (Section 5.3.3)	**INSTR**
INSTRUCT (Section 5.3.4)	**INSTR**[48]

Task Stage Semantic Pointer (see Table 3.6)	Substituted Semantic Pointer
STORE	**STORE**
TRANSFORM1	**TRANS1**
TRANSFORM2	**TRANS2**
QAP	**QAP**
QAK	**QAK**
COUNT1	**CNT1**
COUNT2	**CNT2**
LEARN	**LEARN**
TRANSFORMC (Section 5.2.2)	**TRANSC**
DIRECT (Section 5.3.2)	**DIRECT**
INSTRUCT_POS (Section 5.3.6)	**INSTRP**

Task Decode Semantic Pointer (see Table 3.7)	Substituted Semantic Pointer
FORWARD	**FWD**
COUNT	**CNT**
DRAW	**DECW**
INDUCTION	**DECI**

[48]Both the **INSTR_RESP** and **INSTR** task representations configure the action selection hierarchy to use information extracted using the instruction processing system. Thus, they have been merged.

The condition-consequence pairs for each of Spaun's tasks are as follows:

Copy Drawing Task:

$$\begin{aligned} 0.5(task \bullet \mathsf{X}) + 0.5(vis \bullet \mathsf{1}) &\longmapsto \mathsf{W} \Rightarrow task, \mathsf{STORE} \Rightarrow stage, \mathsf{FWD} \Rightarrow dec \\ (task \bullet (\mathsf{W} - \mathsf{DEC})) - (vis \bullet \mathsf{?}) &\longmapsto stage \Rightarrow stage \\ 0.5(vis \bullet \mathsf{?}) + 0.5(task \bullet (\mathsf{W} - \mathsf{DEC})) &\longmapsto (\mathsf{W} + \mathsf{DEC}) \Rightarrow task, stage \Rightarrow stage, \mathsf{DECW} \Rightarrow dec \end{aligned}$$

Digit Recognition Task:

$$\begin{aligned} 0.5(task \bullet \mathsf{X}) + 0.5(vis \bullet \mathsf{0}) &\longmapsto \mathsf{R} \Rightarrow task, \mathsf{STORE} \Rightarrow stage, \mathsf{FWD} \Rightarrow dec \\ (task \bullet (\mathsf{R} - \mathsf{DEC})) - (vis \bullet \mathsf{?}) &\longmapsto stage \Rightarrow stage \end{aligned}$$

Note that the digit recognition task shares a common "decode step" condition-consequence pair with the working memory, question answering, stimulus matching, stimulus-response, and instruction processing tasks. This condition-consequence pair is:

$$\begin{aligned} (vis \bullet \mathsf{?}) - 0.6(task \bullet (\mathsf{W} + \mathsf{L} + \mathsf{C} + \mathsf{V} + \mathsf{F} + \mathsf{RESP})) \longmapsto\ &(task + \mathsf{DEC}) \Rightarrow task, \\ &(stage + 0.5 \times \mathsf{STORE}) \Rightarrow stage, \\ &(dec + 0.5 \times \mathsf{FWD}) \Rightarrow dec \end{aligned}$$

N-arm Bandit Task:

$$\begin{aligned} 0.5(task \bullet \mathsf{X}) + 0.5(vis \bullet \mathsf{2}) - (vis \bullet \mathsf{?}) &\longmapsto \mathsf{L} \Rightarrow task, \mathsf{LEARN} \Rightarrow stage, \mathsf{FWD} \Rightarrow dec \\ 0.7(vis \bullet \mathsf{?}) + 0.3(task \bullet \mathsf{L}) &\longmapsto (\mathsf{L} + \mathsf{DEC}) \Rightarrow task, \mathsf{LEARN} \Rightarrow stage, \mathsf{FWD} \Rightarrow dec \end{aligned}$$

For each of the n arms of the bandit task, Spaun's action selection hierarchy also contain the following condition-consequence pair:

$$0.5(task \bullet \mathsf{L}) - (vis \bullet \mathsf{?}) \longmapsto \mathsf{ACT}n \Rightarrow action, \mathsf{LEARN} \Rightarrow stage, \emptyset \Rightarrow dec$$

Working Memory Task:

$$\begin{aligned} 0.5(task \bullet \mathsf{X}) + 0.5(vis \bullet \mathsf{3}) &\longmapsto \mathsf{M} \Rightarrow task, \mathsf{STORE} \Rightarrow stage, \mathsf{FWD} \Rightarrow dec \\ (task \bullet (\mathsf{M} - \mathsf{DEC})) - (vis \bullet (\mathsf{F} + \mathsf{R} + \mathsf{?})) &\longmapsto stage \Rightarrow stage \end{aligned}$$

The working memory task also supports switching between forward and backwards recall using the following condition-consequence pairs:

$$\begin{aligned}
0.5(task \bullet \textsf{\textbf{M}}) - (task \bullet \textsf{\textbf{DEC}}) + 0.5(vis \bullet \textsf{\textbf{F}}) - (vis \bullet \textsf{\textbf{?}}) &\longmapsto \textsf{\textbf{FWD}} \Rightarrow dec \\
0.5(task \bullet \textsf{\textbf{M}}) - (task \bullet \textsf{\textbf{DEC}}) + 0.5(vis \bullet \textsf{\textbf{R}}) - (vis \bullet \textsf{\textbf{?}}) &\longmapsto \textsf{\textbf{REV}} \Rightarrow dec
\end{aligned}$$

Counting Task:

$$\begin{aligned}
0.5(task \bullet \textsf{\textbf{X}}) + 0.5(vis \bullet \textsf{\textbf{4}}) &\longmapsto \textsf{\textbf{C}} \Rightarrow task, \textsf{\textbf{STORE}} \Rightarrow stage, \\
&\ \textsf{\textbf{FWD}} \Rightarrow dec \\
0.5(task \bullet \textsf{\textbf{C}}) - (task \bullet \textsf{\textbf{DEC}}) + 0.5(stage \bullet \textsf{\textbf{STORE}}) - (vis \bullet \textsf{\textbf{?}}) &\longmapsto \textsf{\textbf{CNT1}} \Rightarrow stage \\
0.5(task \bullet \textsf{\textbf{C}}) - (task \bullet \textsf{\textbf{DEC}}) + 0.5(stage \bullet \textsf{\textbf{CNT1}}) - (vis \bullet \textsf{\textbf{?}}) &\longmapsto \textsf{\textbf{CNT2}} \Rightarrow stage \\
0.5(task \bullet (\textsf{\textbf{C}} - \textsf{\textbf{DEC}})) + 0.5(vis \bullet \textsf{\textbf{?}}) &\longmapsto (\textsf{\textbf{C}} + \textsf{\textbf{DEC}}) \Rightarrow task, \\
&\ stage \Rightarrow stage, \textsf{\textbf{CNT}} \Rightarrow dec \\
\begin{bmatrix} 0.25(task \bullet \textsf{\textbf{DEC}}) + 0.25(stage \bullet \textsf{\textbf{CNT2}}) + ((dec \bullet \textsf{\textbf{CNT}}) - 1) + \\ 0.5(trfm\_compare \bullet \textsf{\textbf{NO\_MATCH}}) \end{bmatrix} &\longmapsto \textsf{\textbf{CNT2}} \Rightarrow stage, \textsf{\textbf{CNT}} \Rightarrow dec \\
\begin{bmatrix} 0.25(task \bullet \textsf{\textbf{DEC}}) + 0.25(stage \bullet \textsf{\textbf{CNT2}}) + ((dec \bullet \textsf{\textbf{CNT}}) - 1) + \\ 0.5(trfm\_compare \bullet \textsf{\textbf{MATCH}}) \end{bmatrix} &\longmapsto \textsf{\textbf{STORE}} \Rightarrow stage, \textsf{\textbf{FWD}} \Rightarrow dec
\end{aligned}$$

Question Answering Task:

$$\begin{aligned}
0.5(task \bullet \textsf{\textbf{X}}) + 0.5(vis \bullet \textsf{\textbf{5}}) &\longmapsto \textsf{\textbf{Q}} \Rightarrow task, \textsf{\textbf{STORE}} \Rightarrow stage, \textsf{\textbf{FWD}} \Rightarrow dec \\
(task \bullet (\textsf{\textbf{Q}} - \textsf{\textbf{DEC}})) - (vis \bullet (\textsf{\textbf{K}} + \textsf{\textbf{P}} + \textsf{\textbf{?}})) &\longmapsto stage \Rightarrow stage \\
0.5(task \bullet \textsf{\textbf{Q}}) - (task \bullet \textsf{\textbf{DEC}}) + 0.5(vis \bullet \textsf{\textbf{K}}) - (vis \bullet \textsf{\textbf{?}}) &\longmapsto \textsf{\textbf{QAK}} \Rightarrow stage \\
0.5(task \bullet \textsf{\textbf{Q}}) - (task \bullet \textsf{\textbf{DEC}}) + 0.5(vis \bullet \textsf{\textbf{P}}) - (vis \bullet \textsf{\textbf{?}}) &\longmapsto \textsf{\textbf{QAP}} \Rightarrow stage
\end{aligned}$$

Rapid Variable Creation (RVC) Task:

$$\begin{aligned}
0.5(task \bullet \textsf{\textbf{X}}) + 0.5(vis \bullet \textsf{\textbf{6}}) &\longmapsto \textsf{\textbf{V}} \Rightarrow task, \textsf{\textbf{STORE}} \Rightarrow stage, \\
&\ \textsf{\textbf{FWD}} \Rightarrow dec \\
0.5(task \bullet \textsf{\textbf{V}}) - (task \bullet \textsf{\textbf{DEC}}) + 0.5(stage \bullet \textsf{\textbf{STORE}}) - (vis \bullet \textsf{\textbf{?}}) &\longmapsto \textsf{\textbf{TRANS1}} \Rightarrow stage \\
0.5(task \bullet \textsf{\textbf{V}}) - (task \bullet \textsf{\textbf{DEC}}) + 0.5(stage \bullet \textsf{\textbf{TRANS1}}) - (vis \bullet \textsf{\textbf{?}}) &\longmapsto \textsf{\textbf{STORE}} \Rightarrow stage
\end{aligned}$$

Both the RVC task and the fluid induction task share the following common "decode step" condition-consequence pair:

$$0.5(vis \bullet \textsf{\textbf{?}}) + 0.5(task \bullet (\textsf{\textbf{V}} + \textsf{\textbf{F}} - \textsf{\textbf{DEC}})) \longmapsto (task + \textsf{\textbf{DEC}}) \Rightarrow task, stage \Rightarrow stage, \\ \textsf{\textbf{DECI}} \Rightarrow dec$$

Fluid Induction Task:

$$\begin{aligned} 0.5(task \bullet \textsf{\textbf{X}}) + 0.5(vis \bullet \textsf{\textbf{7}}) &\longmapsto \textsf{\textbf{F}} \Rightarrow task, \textsf{\textbf{STORE}} \Rightarrow stage, \\ &\qquad \textsf{\textbf{FWD}} \Rightarrow dec \\ 0.5(task \bullet \textsf{\textbf{F}}) - (task \bullet \textsf{\textbf{DEC}}) + 0.5(stage \bullet \textsf{\textbf{STORE}}) - (vis \bullet \textsf{\textbf{?}}) &\longmapsto \textsf{\textbf{TRANS1}} \Rightarrow stage \\ 0.5(task \bullet \textsf{\textbf{F}}) - (task \bullet \textsf{\textbf{DEC}}) + 0.5(stage \bullet \textsf{\textbf{TRANS1}}) - (vis \bullet \textsf{\textbf{?}}) &\longmapsto \textsf{\textbf{TRANS2}} \Rightarrow stage \\ 0.5(task \bullet \textsf{\textbf{F}}) - (task \bullet \textsf{\textbf{DEC}}) + 0.5(stage \bullet \textsf{\textbf{TRANS2}}) - (vis \bullet \textsf{\textbf{?}}) &\longmapsto \textsf{\textbf{STORE}} \Rightarrow stage \end{aligned}$$

Stimulus Matching Task:

$$\begin{aligned} 0.5(task \bullet \textsf{\textbf{X}}) + 0.5(vis \bullet \textsf{\textbf{C}}) &\longmapsto \textsf{\textbf{CMP}} \Rightarrow task, \\ &\qquad \textsf{\textbf{TRANS1}} \Rightarrow stage, \\ &\qquad \textsf{\textbf{FWD}} \Rightarrow dec \\ 0.5(task \bullet \textsf{\textbf{CMP}}) - (task \bullet \textsf{\textbf{DEC}}) + 0.5(stage \bullet \textsf{\textbf{STORE}}) - (vis \bullet \textsf{\textbf{?}}) &\longmapsto \textsf{\textbf{TRANS2}} \Rightarrow stage \\ 0.5(task \bullet \textsf{\textbf{CMP}}) - (task \bullet \textsf{\textbf{DEC}}) + 0.5(stage \bullet \textsf{\textbf{TRANS1}}) - (vis \bullet \textsf{\textbf{?}}) &\longmapsto \textsf{\textbf{TRANSC}} \Rightarrow stage \end{aligned}$$

Stimulus-response Task:

$$0.5(task \bullet \textsf{\textbf{X}}) + 0.5(vis \bullet \textsf{\textbf{8}}) \longmapsto \textsf{\textbf{RESP}} \Rightarrow task, \textsf{\textbf{DIRECT}} \Rightarrow stage, \textsf{\textbf{FWD}} \Rightarrow dec$$

For each stimulus-response pairing ($STIM\_N \rightarrow RESP\_N$), the following condition-consequence pair is used:

$$0.5(task \bullet \textsf{\textbf{RESP}}) + 0.5(vis\_mem \bullet STIM\_N) \longmapsto RESP\_N \Rightarrow transform$$

Instructed Stimulus-response Task:

$$0.5(task \bullet \textsf{\textbf{X}}) + 0.5(vis \bullet \textsf{\textbf{9}}) \longmapsto \textsf{\textbf{INSTR}} \Rightarrow task, \textsf{\textbf{DIRECT}} \Rightarrow stage, \textsf{\textbf{FWD}} \Rightarrow dec$$

Instructed Tasks (Stages 3 – 5):

$$\begin{aligned}
1.5(vis \bullet (\mathsf{M}+\mathsf{V})) &\longmapsto \mathsf{INSTR} \Rightarrow task, \mathsf{STORE} \Rightarrow stage, \mathsf{FWD} \Rightarrow dec \\
\begin{bmatrix} (task \bullet \mathsf{INSTR}) - (stage \bullet \mathsf{INSTRP}) - \\ (vis \bullet (\mathsf{?}+\mathsf{A}+\mathsf{M}+\mathsf{P}+\blacktriangleleft)) \end{bmatrix} &\longmapsto \begin{array}{l} ins\_task \Rightarrow task, ins\_stage \Rightarrow stage, \\ ins\_dec \Rightarrow dec, ins\_data \Rightarrow transform \end{array} \\
0.5(task \bullet \mathsf{INSTR}) + 0.5(vis \bullet \mathsf{P}) &\longmapsto \mathsf{INSTR} \Rightarrow task, \mathsf{INSTRP} \Rightarrow stage \\
0.5(task \bullet \mathsf{INSTR}) + 0.5(stage \bullet \mathsf{INSTRP}) &\longmapsto \mathsf{INSTR} \Rightarrow task, \mathsf{STORE} \Rightarrow stage
\end{aligned}$$

B.3 Bounded Error Value Computation

The reward evaluation system in the Spaun2.0 model introduced the additional requirement of factoring the basal ganglia utility values into the computation of the error values. This is done to limit the learned basal ganglia utility within the range of $[0, 1]$. Table 4.1 (reproduced below) shows the updated Spaun2.0 mapping between the action chosen, the reward stimulus received, and the basal ganglia utility value for Spaun's 3-arm bandit task.

		Error Values		
Action Chosen	Reward Digit	$\mathbf{E}_1$	$\mathbf{E}_2$	$\mathbf{E}_3$
$\mathbf{A}_1$	0	$\mathbf{U}_1$	$\mathbf{U}_2 - 1$	$\mathbf{U}_3 - 1$
	1	$\mathbf{U}_1 - 1$	$\mathbf{U}_2$	$\mathbf{U}_3$
$\mathbf{A}_2$	0	$\mathbf{U}_1 - 1$	$\mathbf{U}_2$	$\mathbf{U}_3 - 1$
	1	$\mathbf{U}_1$	$\mathbf{U}_2 - 1$	$\mathbf{U}_3$
$\mathbf{A}_3$	0	$\mathbf{U}_1 - 1$	$\mathbf{U}_2 - 1$	$\mathbf{U}_3$
	1	$\mathbf{U}_1$	$\mathbf{U}_2$	$\mathbf{U}_3 - 1$

From the table above, the computed error values take the form $\mathbf{U}_n - x$, where x is either 0 or 1. Error values for matching actions (i.e., $\mathbf{E}_1$ for $\mathbf{A}_1$), $x = 1$ when a reward is provided, and $x = 0$ otherwise. This is reversed for error values for non-matching actions. For Spaun's 3-arm bandit task, this can be summarized as:

		x Values		
Action Chosen	Reward Digit	$\mathbf{E}_1$	$\mathbf{E}_2$	$\mathbf{E}_3$
$\mathbf{A}_1$	0	0	1	1
	1	1	0	0
$\mathbf{A}_2$	0	1	0	1
	1	0	1	0
$\mathbf{A}_3$	0	1	1	0
	1	0	0	1

To generalize the behaviour of x, for a given action A_n, and a given error value E_m, four outcomes are possible:

- If $n \neq m$, and the reward digit is 0, then $x = 1$.
- If $n \neq m$, and the reward digit is 1, then $x = 0$.
- If $n \equiv m$, and the reward digit is 0, then $x = 0$.
- If $n \equiv m$, and the reward digit is 1, then $x = 1$.

To formalize a binary computation, the value a is assigned a value of 1 when $n \equiv m$, and a value of 0 when $n \neq m$. Likewise, the value r is assigned a value 1 when the reward digit is 1, and 0 otherwise. With these assignments, and given the four possible combinations of a and r, the truth table for the binary computation is:

a	r	x
0	0	1
0	1	0
1	0	0
1	1	1

From the truth table, it can be deduced that the value of x can be computed using the $XNOR$ binary operator ($\leftrightarrow$):

$$x = a \leftrightarrow r$$

The $XNOR$ operator can be further decomposed into the following binary computation using the NOT ($\neg$), AND ($\wedge$), and OR ($\vee$) operators:

$$x = (a \wedge r) \vee (\neg a \wedge \neg r)$$

Each of the binary operators has an algebraic form. As an example the algebraic form of the *NOT* operator is:

$$\neg x = 1 - x$$

Likewise, the *AND* operator can be written as:

$$x \wedge y = x \times y,$$

and finally, the *OR* operator has the form:

$$x \vee y = x + y,$$

Using the algebraic forms of the binary operators, the value of x can then be computed as:

$$\begin{aligned} x &= (a \wedge r) \vee (\neg a \wedge \neg r) \\ &= (ar) + ((1-a)(1-r)) \end{aligned} \tag{B.1}$$

Substituting Eq. (B.1) back into $(\mathbf{U}_n - r)$ and applying it to Spaun's 3-arm bandit task yields the results in Table 4.2 (reproduced below for convenience).

Action	a Values			Reward		Error Values $(\mathbf{U}_n - [(a_n r) + (1-a_n)(1-r)])$		
Chosen	a_1	a_2	a_3	Digit	r	$\mathbf{E}_1$	$\mathbf{E}_2$	$\mathbf{E}_3$
$\mathbf{A}_1$	1	0	0	0	0	$\mathbf{U}_1$	$\mathbf{U}_2 - 1$	$\mathbf{U}_3 - 1$
	1	0	0	1	1	$\mathbf{U}_1 - 1$	$\mathbf{U}_2$	$\mathbf{U}_3$
$\mathbf{A}_2$	0	1	0	0	0	$\mathbf{U}_1 - 1$	$\mathbf{U}_2$	$\mathbf{U}_3 - 1$
	0	1	0	1	1	$\mathbf{U}_1$	$\mathbf{U}_2 - 1$	$\mathbf{U}_3$
$\mathbf{A}_3$	0	0	1	0	0	$\mathbf{U}_1 - 1$	$\mathbf{U}_2 - 1$	$\mathbf{U}_3$
	0	0	1	1	1	$\mathbf{U}_1$	$\mathbf{U}_2$	$\mathbf{U}_3 - 1$