Biologically Plausible, Human-scale Knowledge Representation

by

Eric Crawford

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2014

Author's Declaration

Abstract

Several approaches to implementing symbol-like representations in neurally plausible models have been proposed. These approaches include binding through synchrony, mesh binding, and tensor product binding. Recent theoretical work has suggested that these methods will not scale well; that is, they cannot encode human-sized structured representations without making implausible resource assumptions. Here I present an approach that will scale appropriately, which is based on the Semantic Pointer Architecture. Specifically, I construct a spiking neural network composed of about 2.5 million neurons that employs semantic pointers to encode and decode the main lexical relations in WordNet, a semantic network containing over 117,000 concepts. I experimentally demonstrate the capabilities of this model by measuring its performance on three tasks which test its ability to accurately traverse the WordNet hierarchy, as well as its ability to decode sentences involving WordNet concepts. I argue that these results show that this approach is uniquely well-suited to providing a biologically plausible account of the structured representations that underwrite human cognition. I conclude with an investigation of how the connection weights in this spiking neural network can be learned online through biologically plausible learning rules.

Acknowledgements

I first want to thank all the members of the Computational Neuroscience Research Group for their support and friendship over the years, and for their patience during the dark undergrad period. I'd especially like to thank Xuan Choo without whose patience and encyclopedic knowledge of technical matters this thesis would never have seen the light of day. Also, thanks to Terry Stewart for always having his door open, and for his boundless willingness to drop what he is working on to help a labmate out. I also greatly appreciate Bryan Tripp and Jeff Orchard for agreeing to read this thesis.

Huge thank you to my supervisor Chris Eliasmith for his gentle guidance and for the perfect research topic, and to Jen Eliasmith for supporting the lab in so many ways.

Finally, thank you to my girlfriend Erika for everything.

Dedication

To my mom and dad for their time, love and care.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

One of the central challenges for contemporary cognitive modeling is scaling. As Geoff Hinton recently remarked in his address to the Cognitive Science Society, "In the Hitchhiker's Guide to the Galaxy, a fearsome intergalactic battle fleet is accidentally eaten by a small dog due to a terrible miscalculation of scale. I think that a similar fate awaits most of the models proposed by Cognitive Scientists" [19, p. 7]. Whether or not we agree, this observation can at least be taken as a challenge for cognitive modelers: Will the principles demonstrated in small-scale cognitive models scale up to the complexity of a human-sized cognitive system?

This scaling problem has often been thought to be a special challenge for biologically inspired approaches to cognitive modeling [22, 18]. This is because the basic principles employed in such models do not provide a straightforward characterization of structured representations. Consequently, it is reasonable to wonder how such principles will ultimately be able to account for *human-scale* structured representations, which they clearly must do if they hope to provide convincing explanations of cognitive behavior. This same concern is not as immediate for symbolic approaches, which typically take structured representations to be primitive [42, 1].

In this paper we present a novel method for representing structured knowledge in biologically plausible neural networks, and show that it alone is capable of scaling up to a human-sized lexicon. This approach combines a method for encoding structured knowledge in vectors with a framework for building biologically realistic neural models capable of representing and transforming those vectors. In previous work, we have demonstrated that this approach meets many of the challenges that have been posed for connectionist accounts of structured representation, including the ability to account for the systematicity, compositionality and productivity of natural languages, as well as the massive binding problem and the rapid variable creation problem [7]. That work has also given theoretical reasons to think that this approach will scale better than others; here our focus is on empirically demonstrating that claim. We achieve this by using our method to encode the human-scale knowledge base known as WordNet in a spiking neural network, and show that, unlike past approaches, this network places plausible neural resource demands given what is known about the size of relevant brain areas.

The remainder of the paper is organized as follows. In Section 2 we review past connectionist approaches to the problem of representing structure, and discuss recent criticisms of those approaches which suggest that they will not scale. In Section 3 we introduce the concept of a semantic pointer, the main type of representation employed by our approach. In Section 4 we present Holographic Reduced Representations (HRRs), a vector algebra capable of encoding structured representations in vectors, which we use to create semantic pointers. In Section 5 we show how to use these semantic pointers to encode WordNet and outline an algorithm for extracting the relational information stored in the resulting encoding. In Section 6 we show how to build a biologically realistic neural network based on this extraction algorithm, and show that it uses far fewer neural resources than the previously discussed approaches.

In Section 7 we demonstrate the capabilities of both the abstract extraction algorithm and its neural implementation by subjecting them to a number of experiments designed to confirm that WordNet is accurately encoded. In particular, these experiments show 1) that structural information can be extracted from arbitrary WordNet concepts, 2) that hierarchies of arbitrary depth within WordNet are correctly represented, and 3) that the network can be used to extract the constituents of sentences involving arbitrary WordNet concepts. In Section 8 we consider exactly how our approach is able to achieve its superior scaling, and discuss how this work relates to theoretical debates in the cognitive science literature. In Section 9 experimentally investigate the viability of alternate techniques for encoding semantic networks in vectors. Finally, in Section 10 we make preliminary investigations into how one of the central components in our model, the neural associative memory, can be learned online from training data.

Chapter 2

Past approaches

There have been many approaches to representing structure in connectionist networks. We consider three of the most successful: binding through synchrony, "mesh" binding, and tensor product binding.

2.1 Binding through synchrony

The suggestion that structured cognitive representations could be constructed using binding through synchrony [43] was imported into cognitive modeling from the earlier hypothesis that feature binding in vision can be accounted for by the synchronization of spiking neurons in visual cortex [55].

In their SHRUTI architecture, Shastri & Ajjanagadde [43] demonstrated that exploiting synchrony can provide a solution to the variable binding problem in several simple examples. More recently, binding through synchrony has seen a revival in the LISA [20] and DORA [5] architectures, which focus on representing structures for analogical reasoning.

In all of these models, the temporal relationships between connectionist nodes are employed to represent structured relations. In DORA and LISA specifically, a structured representation such as bigger(Max, Eve) is constructed out of four levels of representation. The first level consists of nodes representing "sub-symbols" (e.g. furry, female, etc.). The second level consists of units connected to a set of sub-symbols relevant to defining the meaning of the second level term (e.g. Max is connected to furry, Eve to female, etc.). The third level consists of "sub-proposition" nodes that bind roles to objects (e.g. Max+larger, or Eve+smaller, etc.). The fourth level consists of proposition nodes that bind sub-propositions to form whole propositions.

As has been argued in more detail elsewhere, this kind of representational scheme will not scale well [51, 7] because the number of nodes needed to support arbitrary structured representations over even small vocabularies (e.g. 2000 lexical items) is larger than the number of neurons in the brain.[1]

[1] Briefly, the calculation is as follows. Assume 1500 nouns and 500 verbs. The number of nodes needed to

Notably, this criticism is not problematic because of the use of synchrony per se, but rather because of the way binding has been mapped to network nodes. However, it has also been suggested that synchrony itself will not scale well to binding complex structures [35, 51].

2.2 Mesh binding

A different approach to structure representation has been taken by van der Velde and de Kamps [53] in their work on the Neural Blackboard Architecture (NBA). To avoid the exponential growth in resources needed for structure representation in a DORA-like scheme, the NBA employs "neural assemblies." These assemblies are temporarily bound to particular symbols using a "mesh grid" of neural circuits (e.g. bind(noun1, Max)). Larger structures are then built by binding these assemblies to roles using a gating circuit (e.g. gate(agent1, bind(noun1, Max))). Neural assemblies that bind roles (and hence their gated word assemblies) are used to define higher level "structure assemblies." Such structure assemblies can be used to represent sentential structures.

The use of temporary binding in this manner significantly reduces the resource demands of this approach compared to the synchrony-based approaches. However, it does not reduce the demands sufficiently to make the NBA neurally plausible. As argued in [51], and demonstrated in more detail in [7], in order to represent simple sentences of the form *relation(agent, theme)* from a vocabulary of 60,000 terms, this approach requires roughly 480 cm^2 of cortex, approximately one-fifth of total cortical area.[2] This is much larger than the combined sizes of "naming" cortex (about 7 cm^2; 34), Wernicke's and Broca's areas (about 20 cm^2 each; 24), and the remaining parts of the language "implementation system" (supramarginal gyrus, angular gyrus, auditory cortex, motor cortex, and somatosensory cortex, about 200 cm^2; 6). Critically, these areas do far more than represent structure; they account for phonological processing, oral motor control, grammatical processing, sentence parsing and production, etc. Consequently, while the NBA has improved scalability compared to DORA, it remains implausible.

2.3 Tensor product binding

The final approach we consider, first proposed by Paul Smolensky [46], is the earliest and best known member of the class of proposals broadly called Vector Symbolic Architectures (VSAs; 15). In general, these approaches represent symbols with vectors, and propose some kind of nonlinear

represent arbitrary structures of the form relation(noun, noun) is $500\times1500\times1500=1.1\times10^9$ nodes. Assuming 100 neurons per node, which provides a signal-to-noise ratio of 10:1 [8], results in 1.1×10^{11} neurons. There are about 20×10^6 neurons per cm^2 [37], and 2500 cm^2 of cortex [38] giving about 50×10^9 neurons in cortex, less than that required for the assumed representation.

[2]Following the values used in ff1, the calculation is as follows. [53] notes that each connection between symbol and word assemblies requires 8 neural groups, and that 100 assemblies per role should be sufficient. Assuming only two grammatical roles (to be conservative) results in $60{,}000\times200\times8=96\times10^6$ groups needed. This suggests 96×10^8 neurons are needed, which works out to about 480 cm^2 of cortex.

vector operation to bind two vectors together. Later, the constituents of the binding can be extracted using an unbinding operation. Smolensky's architecture employs the tensor product as the binding operation, which has the advantage that it allows the constituents of a binding to be perfectly extracted.

In terms of scaling, this approach has the benefit that symbols and propositions can be represented by patterns of neural activity, alleviating the need for devoted neural resources for each proposition (as in the case of synchrony based approaches) and complex gating mechanisms (as in the case of the mesh binding approaches). However, the use of the tensor product as a binding operator creates a separate scaling issue. Because the tensor product of two n-dimensional vectors is an n^2-dimensional vector, this framework scales exponentially poorly as the depth of the encoded structure increases. For example, [7] shows that encoding a two-level sentence such as "Bill believes that Max is larger than Eve", where lexical items may have hierarchical relations of depth two or more, will require approximately 625 cm^2 of cortex.[3]

We now present the details of our approach for connectionist structured representation, with the aim of showing that, unlike the approaches explored here, it can encode a human-scale lexicon using a plausible number of neurons.

[3]Briefly, the calculation is as follows. Conservatively assume that only 8 dimensions are needed to distinguish the lowest-level concepts (e.g. isA, mammal). Then the representation of Eve requires $8\times8\times8$=512-dimensional vectors (i.e. Eve = isA$\otimes$person+... = isA$\otimes$isA$\otimes$mammal+...). Assuming at least one concept at each level of the sentence requires such a representation means that $512\times512\times512=12.5\times10^7$ dimensions or 12.5×10^9 neurons are required, which works out to 625 cm^2 of cortex. Again, this is only for structure representation – not processing– and is significantly larger than relevant language areas.

Chapter 3

Semantic pointers

Semantic pointers are neurally realized vector representations generated through a compression method, and are typically of a high dimensionality [7]. In general, semantic pointers are constructed by compressing information from one or more high-dimensional vector representations, which can be semantic pointers themselves. The newly generated semantic pointer has a dimensionality that is less than or equal to the dimensionality of its constituents. Semantic pointers can be subsequently decompressed (or dereferenced) to recover (much of) the original information.

Examples of compression that lowers dimensionality and loses information abound in the digital world. Jpegs, mp3s, and H.264 videos are all examples of lossy compression. These methods are lossy because, from an information theoretic perspective, the decompression of the compressed vector contains less information than the original pre-compressed vector. However, from a psychophysical perspective, the pre-compressed and reconstructed vectors can be nearly indistinguishable. The reason such compression methods are ubiquitous is because they can massively decrease the amount of data that must be manipulated or transmitted, while preserving the essential features of that data. Semantic pointers are proposed to play an analogous role in our mental lives.

Because semantic pointers are compact ways of referencing large amounts of data, they function similarly to pointers as understood in computer science. Typically, in computer science a pointer is the address of a large amount of data stored in memory. Unlike the data in memory, pointers are easy to transmit, manipulate and store, because they occupy a small number of bytes. Hence pointers can act as an efficient proxy for the data they point to. Semantic pointers are proposed to provide the same kind of efficiency benefits in a neural setting.

Unlike pointers in computer science, however, semantic pointers are *semantic*. That is, they are systematically related to the information that they reference, because they were generated from that information via compression. This means that semantic pointers carry similarity information that is derived from their source (unlike computer science pointers). If two uncompressed structures are similar, then their compressed semantic pointers will also be similar, given an appropriate compression method and similarity measure (e.g. dot product or cosine similarity).

The similarity relations between semantic pointers are best thought of as capturing shallow semantics. That is, semantics that can be read off of the surface features of the semantic pointers themselves. To get at deep semantics — e.g. semantics dependent on subtle structural relations of the uncompressed data — it can be crucial to more directly compare the uncompressed states. In many ways, this distinction between shallow and deep semantics is reminiscent of shallow and deep processing proposed in dual-coding theory [36]. Typically, this older distinction is taken to map onto the distinction between verbal and perceptual processing [17]. Recent fMRI and behavioral experiments are supportive of this view [47, 44]. Semantic pointers can be thought of as a computational specification of this distinction.

In sum, semantic pointers are neurally realized, compressed (and hence efficient) representations of higher dimensional data. They carry surface semantics, for which similarity can be cheaply computed, and they can be decompressed to access deeper semantics with additional computation.

In the current paper, we primarily use semantic pointers for their deep semantics, and make little use of the shallow semantics. However, the shallow semantics are nonetheless present, which has important theoretical consequences and leaves open a number of interesting extensions that we discuss in Section 8.

Use of semantic pointers requires the specification of both a compression algorithm and a corresponding decompression algorithm. For example, in the vision system of the Spaun model, both compression and decompression take the form of a generative, hierarchical vision model [9]. A semantic pointer for a visual scene is created by running the model “forward”, extracting a relatively low-dimensional representation that captures the scene’s important features. The full visual scene can later be approximately reconstructed by running the model “backward”, with the top of the hierarchy clamped to the desired semantic pointer. In the current study, we use compression algorithms that are better suited to structured representations. In particular, we use the operations provided by Holographic Reduced Representations, a Vector Symbolic Architecture.

In Section 6, we show how to use the Neural Engineering Framework to build spiking neural networks that represent and transform high-dimensional vectors, providing a neural implementation of this form of semantic pointer.

Chapter 4

Holographic Reduced Representations

Holographic Reduced Representations (HRRs) are a type of Vector Symbolic Architecture, and, as such, constitute a means of representing structured knowledge in a vector format. HRRs have some similarities to Smolensky's tensor product technique, but use a compressive binding operator which allows the dimensionality of the representations to remain constant as the depth of the encoded structure increases. This is an important difference which lends HRRs superior scaling properties. Here we briefly sketch the components of the HRR vector algebra. See [39] for a more complete introduction to HRRs, and [40] for an in-depth treatment of the mathematics involved as well as a number of applications.

To begin, the basic elements of the structure that we want to represent are each assigned a random D-dimensional vector, where D is fixed ahead of time. We can then use the operations specified by the HRR formalism to create vectors encoding structured combinations involving those basic elements. Other operations can later be applied to the structured vectors to extract the constituent vectors. In this section we discuss the necessary vector operations and their properties, before going on to explicitly show how they can be used together to encode structured information. The three vector operations specified by the HRR algebra are *circular convolution*, *vector addition*, and *involution*.

4.1 Circular Convolution

Circular convolution, represented by the $\circledast$ symbol, plays the role of a *binding* operator. For two vectors $\mathbf{x} = [x_{(0)}, \ldots, x_{(D-1)}]$ and $\mathbf{y} = [y_{(0)}, \ldots, y_{(D-1)}]$, and $j \in \{0, \ldots, D-1\}$, the jth element of $\mathbf{x} \circledast \mathbf{y}$ is:

$$(\mathbf{x} \circledast \mathbf{y})_{(j)} = \sum_{k=0}^{D-1} \mathbf{x}_{(k)} \mathbf{y}_{(j-k)}$$

where the indices are taken modulo D. The circular convolution of two vectors is dissimilar to both of them, using the dot product as a measure of similarity.

4.2 Vector addition

Vector addition plays the role of a *superposition* operator. In particular, it allows multiple bindings to be stored in a single vector. The jth element of $\mathbf{x}+\mathbf{y}$ is:

$$(\mathbf{x}+\mathbf{y})_{(j)} = \mathbf{x}_{(j)} + \mathbf{y}_{(j)}$$

Vector addition returns a vector that is similar to both of its inputs, again using the dot product as a measure of similarity. The exception is when $\mathbf{x}$ and $\mathbf{y}$ are close to being additive negatives of one another; though if they are random vectors in a high-dimensional space, this situation is unlikely to occur.

4.3 Involution

The third HRR operation, involution, is represented by an overbar (e.g. $\overline{\mathbf{x}}$). The jth element of $\overline{\mathbf{x}}$ is given by :

$$\overline{\mathbf{x}}_{(j)} = \mathbf{x}_{(-j)}$$

where the indices are again taken modulo D. Put simply, the first element of the vector stays in place, and the remaining elements are reversed. For example, if $\mathbf{x} = [1,2,3,4,5]$, then $\overline{\mathbf{x}} = [1,5,4,3,2]$. Involution is the approximate inverse of a vector with respect to circular convolution. Specifically, for two vectors $\mathbf{x}$ and $\mathbf{y}$, we have $\mathbf{x} \circledast \mathbf{y} \circledast \overline{\mathbf{y}} \approx \mathbf{x}$. It can thus be thought of as an *unbinding* operator, since it facilitates the extraction of the constituents of bindings. Circular convolution does have an exact inverse but, for reasons outlined in detail in [40], it performs poorly when $\mathbf{x}$ is noisy. Because we will later be concerned with neural representations of these vectors, which are inherently noisy, throughout this work we employ involution rather than the exact inverse.

The circular convolution, vector addition and involution operations can be thought of as vector analogs of the familiar algebraic operations of multiplication, addition and taking the reciprocal, respectively. Indeed, they have many of the same algebraic properties. For example, circular convolution is commutative, associative, and distributes over vector addition, similar to multiplication. The mathematical details of these operations, and, in particular, why involution is the approximate inverse of convolution, can be found in either of the above references on HRRs.

4.4 Semantic Pointers for Structured Representations

Together, these operations permit the construction of vectors that represent complex structure. Most usefully for the current work, we can construct a vector which stores multiple pairs of other vectors. Later, given some query vector, we can determine which vector it is paired with in the structured vector. For example, suppose we have 6 elements in our vocabulary, each of which has been assigned a vector, $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{f}$ and we want to store the pairs $\langle \mathbf{a}, \mathbf{b} \rangle$, $\langle \mathbf{c}, \mathbf{d} \rangle$ and $\langle \mathbf{e}, \mathbf{f} \rangle$. We can use circular convolution and vector addition to do this as follows:

$$\mathbf{t} = \mathbf{a} \circledast \mathbf{b} + \mathbf{c} \circledast \mathbf{d} + \mathbf{c} \circledast \mathbf{d} \tag{4.1}$$

$\mathbf{t}$ is typically then normalized to have a norm of 1. An important note is that $\mathbf{t}$ has dimensionality D, the same as that of each of the vectors on the right-hand side of this equation. This is because both circular convolution and vector addition return vectors with the same dimensionality as their inputs. This feature is what sets HRRs apart from Smolensky's tensor product technique. In particular, it prevents the size of the vectors from undergoing a combinatorial explosion as the depth of the encoded knowledge structure increases, permitting the efficient representation of hierarchical structures. For instance, $\mathbf{t}$ itself could be included in another structured vector, which would also have dimensionality D.

Given a query vector, we can then use circular convolution and involution to retrieve an approximation of the vector it is paired with in $\mathbf{t}$. For example, to extract the vector that $\mathbf{a}$ is paired with, we compute:

$$\begin{aligned}
&\mathbf{t} \circledast \overline{\mathbf{a}} \\
&= (\mathbf{a} \circledast \mathbf{b} + \mathbf{c} \circledast \mathbf{d} + \mathbf{e} \circledast \mathbf{f}) \circledast \overline{\mathbf{a}} \\
&= \mathbf{a} \circledast \mathbf{b} \circledast \overline{\mathbf{a}} + \mathbf{c} \circledast \mathbf{d} \circledast \overline{\mathbf{a}} + \mathbf{e} \circledast \mathbf{f} \circledast \overline{\mathbf{a}} \\
&= \mathbf{b} \circledast \mathbf{a} \circledast \overline{\mathbf{a}} + \mathbf{d} \circledast \mathbf{c} \circledast \overline{\mathbf{a}} + \mathbf{e} \circledast \mathbf{f} \circledast \overline{\mathbf{a}} \\
&\approx \mathbf{b} + \mathbf{d} \circledast \mathbf{c} \circledast \overline{\mathbf{a}} + \mathbf{e} \circledast \mathbf{f} \circledast \overline{\mathbf{a}} && (4.2) \\
&= \mathbf{b} + \mathit{noise} && (4.3)
\end{aligned}$$

Here we have used both the commutative and distributive properties of circular convolution.

We can now see that these structured vectors can be treated as a special kind of compressed representation. We can think of the original, uncompressed vector as the concatenation of $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}$ and $\mathbf{f}$, which has a dimensionality of $6D$. $\mathbf{t}$, which has only D dimensions, thus represents a significantly compressed version of the original vector. We can then look at the process of circularly convolving $\mathbf{t}$ with the involution of a query vector as lossy decompression, since we extract a vector that is similar but not identical to part of the original, uncompressed vector. But notice that this is a type of decompression that is especially well-suited to structured representation, because we do not have to reconstruct the entire original vector at once. Instead, we can choose which bit of it to decompress by changing the query vector. Throughout this paper we refer to the neural implementation of these structured vectors as semantic pointers.

Chapter 5

Encoding Structured Knowledge in Semantic Pointers

Thus far we have seen how to vectorially encode structured representations at a small scale. In this section we significantly scale up this technique. We first introduce WordNet, a human-scale semantic network, and then show how to create a vector encoding of WordNet using semantic pointers.

5.1 WordNet

In order to empirically demonstrate that our technique can scale up to the size of a human vocabulary, we require a structured knowledge base of sufficient magnitude. One approach would be to construct an arbitrary structured representation, perhaps using random graph techniques; however, there is no reason that such a representation would statistically resemble the structure of human knowledge. A better approach is to choose a sufficiently large structured representation constructed with human knowledge specifically in mind. Fortunately, there are projects that have taken up the monumental task of encoding human knowledge in machine readable form. Two of the most well-known such projects are Cyc and WordNet.

Cyc's aim is ostensibly to codify the entirety of common-sense knowledge in a machine-usable format, with potential applications ranging from medicine to machine learning. Cyc is truly a marvel of perseverance; according to its creator Doug Lenat, more than a person-century of work has gone into the manual construction of at least a million "common sense axioms" [26].

WordNet is another manually constructed database [32, 11], but with slightly more modest goals. It aims to be a lexical database of the English language instead of a database of the entirety of human knowledge. Due to its reduced scope and its much smaller set of basic relationships, WordNet tends to have applied structural features more consistently than Cyc. As well, Cyc organizes concepts abstractly with logical assertions and microtheories, whereas WordNet's design is intended to reflect the organization of concepts in a psychologically plausible way using a handful of common relationships. In total, WordNet contains 117,659 concepts and a high degree of hierarchical structure (e.g. *entity* can be reached from *dog* in 13 steps), suggesting it will be an adequate test of the scalability of our technique. For these reasons, we have chosen WordNet as the structured knowledge base that we will encode.

The basic unit in WordNet is a *synset*, a set of words that have the same meaning. Words that have multiple meanings are listed in multiple synsets. Each synset possesses a number of *relations*, each of which represents a semantic link between that synset and another synset. Relations are unidirectional; each has a source and a target. Each relation also has a type, the most prominent being hypernymy (roughly "isA") and holonymy (roughly "partOf"). These relation types can be further subdivided: hypernymy into *instance* and *class*, and partOf into *part*, *member*, and *substance*. As a concrete example, we show a subset of the relational structure of the *dog* synset:

$$dog = class(canine)\ and\ member(pack) \tag{5.1}$$

Here *dog* is the source of a *class* relation and a *member* relation. *canine* is the target of the *class* relation, and *pack* is the target of the *member* relation. Only the 5 relation-types we have mentioned are encoded in our model. The inverses of these relations are also implicitly included, although we do not test their extraction as this requires more complex control of signal flow that is beyond our present scope. The depiction of lexical relations found in WordNet is somewhat simplified, though it is sufficient for our purposes; a complete description of the simplifications made can be found in [11].

5.2 Semantic Pointers and WordNet

It is relatively straightforward to use semantic pointers to encode the relational structure of a WordNet synset as represented in Equation (5.1). The first step is to fix a dimension D for our vectors. Previous investigations have shown that using 512 dimensions provides sufficient representational capacity to accommodate human-scale knowledge bases [7], so D = 512 in all the work presented here. We then assign each WordNet synset a D-dimensional vector called an *ID-vector*, chosen uniformly at random from the D-dimensional unit hypersphere, which acts as a unique identifier for that synset. Each relation-type (*class*, *member*, etc.) is also assigned a vector in the same way.

The next step is to encode the information about how synsets are related to one another. Each synset is thus assigned a second D-dimensional vector storing the relational information about the synset. In particular, this vector is a semantic pointer constructed using the technique from Section 4.4, where each pair stored in the vector corresponds to a relation belonging to the synset, and consists of the vector for the relation-type and the ID-vector for the target of the relation. The following equation demonstrates this process for the *dog* synset:

$$\mathbf{dog_{sp}} = \mathbf{class} \circledast \mathbf{canine_{id}} + \mathbf{member} \circledast \mathbf{pack_{id}} \tag{5.2}$$

where all variables are D-dimensional vectors. We have disambiguated the two vectors assigned to a synset by denoting ID-vectors with the **id** subscript, and semantic pointers with the **sp** subscript. The semantic pointer is typically normalized thereafter.

We can now use the combination of circular convolution and involution to access the relations that belong to *dog*. As an example, imagine we want to extract the synset that *dog* is related to via the *class* relation-type. We could achieve this by circularly convolving $\mathbf{dog_{sp}}$ with $\overline{\mathbf{class}}$:

$$\begin{aligned}
&\mathbf{dog_{sp}} \circledast \overline{\mathbf{class}} \\
&= (\mathbf{class} \circledast \mathbf{canine_{id}} + \mathbf{member} \circledast \mathbf{pack_{id}}) \circledast \overline{\mathbf{class}} \\
&= \mathbf{class} \circledast \mathbf{canine_{id}} \circledast \overline{\mathbf{class}} + \mathbf{member} \circledast \mathbf{pack_{id}} \circledast \overline{\mathbf{class}} \\
&= \mathbf{canine_{id}} \circledast \mathbf{class} \circledast \overline{\mathbf{class}} + \mathbf{member} \circledast \mathbf{pack_{id}} \circledast \overline{\mathbf{class}} \\
&\approx \mathbf{canine_{id}} + \mathbf{member} \circledast \mathbf{pack_{id}} \circledast \overline{\mathbf{class}} \\
&= \mathbf{canine_{id}} + \mathit{noise}
\end{aligned} \tag{5.3}$$

yielding a vector that is similar to $\mathbf{canine_{id}}$.

One might wonder why we need ID-vectors at all; it might seem more straightforward to define the semantic pointers for a synset directly in terms of semantic pointers for related synsets. For example:

$$\mathbf{dog_{sp}} = \mathbf{class} \circledast \mathbf{canine_{sp}} + \mathbf{member} \circledast \mathbf{pack_{sp}} \tag{5.4}$$

This is problematic for a number of reasons. The most prominent is that WordNet (and semantic networks in general) have directed cycles, and thus some of the semantic pointers would have to be defined in terms of one another, which has no obvious solution.

5.3 Sentences

We can also use this technique to create semantic pointers encoding sentences involving any of the terms in WordNet. In this case, we pair up sentence roles and synsets filling those roles, and store the corresponding vectors in a semantic pointer. This requires assigning random vectors to the roles, just as we have done for relation-types. If we have the roles *subject*, *verb* and *object* with assigned vectors **subject**, **verb** and **object** respectively, then the semantic pointer for the sentence "dogs chase cats" would be:

$$\mathbf{sentence_{sp}} = \mathbf{subject} \circledast \mathbf{dog_{id}} + \mathbf{verb} \circledast \mathbf{chase_{id}} + \mathbf{object} \circledast \mathbf{cat_{id}}$$

Circular convolution and involution can later be used to extract the synset filling a given role in a sentence, similar to Equation (5.3). For example, $\mathbf{sentence_{sp}} \circledast \overline{\mathbf{object}}$ will be a vector similar to $\mathbf{cat_{id}}$.

We can also encode sentences with multiple levels of recursive depth. To demonstrate, the recursively structured sentence “mice believe that dogs chase cats” will have the semantic pointer:

$$\begin{aligned}\mathbf{deep\_sentence_{sp}} = \mathbf{subject} \circledast \mathbf{mouse_{id}} + \mathbf{verb} \circledast \mathbf{believe_{id}} + \\ \mathbf{object} \circledast (\mathbf{subject} \circledast \mathbf{dog_{id}} + \mathbf{verb} \circledast \mathbf{chase_{id}} + \mathbf{object} \circledast \mathbf{cat_{id}})\end{aligned}$$

Top-level constituents (e.g. $\mathbf{mouse_{id}}$, $\mathbf{believe_{id}}$) can be extracted in the usual way, while constituents of the embedded clause can be extracted by using a compound query vector. For example, $\mathbf{deep\_sentence_{sp}} \circledast \overline{(\mathbf{object} \circledast \mathbf{verb})}$ will be a vector similar to $\mathbf{chase_{id}}$. Importantly, because we are using circular convolution for binding, $\mathbf{deep\_sentence_{sp}}$ still has the same dimensionality as all its constituents.

5.4 Associative memory

The result of computing $\mathbf{dog_{sp}} \circledast \overline{\mathbf{class}}$ as in Equation (5.3) is insufficient in two ways. First, because involution is only an approximate inverse, and because of the other terms present, $\mathbf{dog_{sp}} \circledast \overline{\mathbf{class}}$ is only *similar to* $\mathbf{canine_{id}}$; in other words, there is noise that must be removed. Second, $\mathbf{canine_{id}}$ is not particularly useful on its own; it would be more useful to have $\mathbf{canine_{sp}}$, from which we could recursively extract further structural information. These problems can be solved simultaneously by an associative memory.

Associative memories store ordered pairs of vectors $\langle \xi, \eta \rangle$. In an analogy with computer memory, the first vector ξ can be thought of as an address, and the second vector η can be thought of as the information stored at that address. When the memory receives an input, if that input is sufficiently similar to some ξ, then the memory outputs the corresponding η. It is easy to see how this solves our problems if we let the ξ's be ID-vectors and the η's be semantic pointers: the associativity provides us with the semantic pointers instead of the ID-vectors, and the fact that the input only has to be *sufficiently similar* to some ξ solves the denoising problem.

Given N pairs of vectors to associate, $\langle \xi_k, \eta_k \rangle$ for $k \in 1 \ldots N$, the following simple algorithm implements this associative memory recall functionality:

Algorithm 5.4.1: ASSOCIATE(*input*)

```
$sum \leftarrow 0$
**for** $k \leftarrow 1$ **to** $N$
    $sim \leftarrow$ DOT PRODUCT( $\xi_k$ , $input$ )
    $scale \leftarrow 1$ **if** $sim > threshold$ **else** $0$
    $sum \leftarrow sum + scale * \eta_k$
**return** ($sum$)
```

If *threshold* is set correctly (we use 0.3), *scale* will be non-zero for at most one value of k, and the output will be either the zero vector or η_k. For example, if we were using a vocabulary that contained only the synsets *dog*, *canine*, and *pack*, then the pairs stored in the associative memory would be $\langle \mathbf{dog_{id}}, \mathbf{dog_{sp}} \rangle$, $\langle \mathbf{canine_{id}}, \mathbf{canine_{sp}} \rangle$ and $\langle \mathbf{pack_{id}}, \mathbf{pack_{sp}} \rangle$. Now recall that:

$$\mathbf{dog_{sp}} \circledast \overline{\mathbf{class}} \approx \mathbf{canine_{id}} + \mathbf{member} \circledast \mathbf{pack_{id}} \circledast \overline{\mathbf{class}}$$

Since $\mathbf{canine_{id}}$ is contained in $\mathbf{dog_{sp}} \circledast \overline{\mathbf{class}}$ through vector addition, $\mathbf{canine_{id}}$ and $\mathbf{dog_{sp}} \circledast \overline{\mathbf{class}}$ be similar. On the other hand, $\mathbf{dog_{id}}$ does not appear in this equation at all, and because two random high-dimensional vectors have a low probability of being similar, $\mathbf{dog_{sp}} \circledast \overline{\mathbf{class}}$ is unlikely to be similar to $\mathbf{dog_{id}}$. Finally, $\mathbf{pack_{id}}$ appears in $\mathbf{dog_{sp}} \circledast \overline{\mathbf{class}}$ through circular convolution, which, as mentioned above, returns a vector that is dissimilar to its inputs. In short, $\mathbf{dog_{sp}} \circledast \overline{\mathbf{class}}$ will be similar to $\mathbf{canine_{id}}$ and dissimilar to the other ID-vectors. Thus, when $\mathbf{dog_{sp}} \circledast \overline{\mathbf{class}}$ is given as input to the associative memory, *scale* will only be non-zero for the *canine* pair, and the output will be $\mathbf{canine_{sp}}$.

5.5 The statistics of decompressed vectors

Until now two of the parameters in our model, the threshold in the associative memory and the dimensionality of the vectors used in the encoding, have been given specific values without much justification. Here we provide at least a glimpse of why these values work. Both are chosen to make the associative memory work properly given that it has to store on the order of 100,000 pairs of vectors.

In general the input to the associative memory will be a vector obtained through the result of decompression (the combination of circular convolution and involution). For example, first let us define:

$$\mathbf{S_N} = \mathbf{R_1} \circledast (\mathbf{x_1})_{\mathbf{id}} + \mathbf{R_2} \circledast (\mathbf{x_2})_{\mathbf{id}} + \cdots + \mathbf{R_N} \circledast (\mathbf{x_N})_{\mathbf{id}} \tag{5.5}$$

where $\mathbf{S_N}$ is a semantic pointer, $\mathbf{x_1} \ldots \mathbf{x_N}$ are N distinct synsets, and $\mathbf{R_1}, \ldots \mathbf{R_N}$ are N distinct D-dimensional random unit vectors which could be relation-type or role vectors. Then the input to the associative memory will generally have the form $\mathbf{S_N} \circledast \overline{\mathbf{R_i}}$ for arbitrary $i \in 1 \cdots N$. For the associative memory to correctly map this input to $(\mathbf{x_i})_{\mathbf{sp}}$, it must be the case that:

$$\max_{\mathbf{z} \neq \mathbf{x_i}} [(\mathbf{S_N} \circledast \overline{\mathbf{R_i}}) \cdot \mathbf{z_{id}}] < \textit{threshold} < (\mathbf{S_N} \circledast \overline{\mathbf{R_i}}) \cdot (\mathbf{x_i})_{\mathbf{id}} \tag{5.6}$$

If the first condition is satisfied but not the second, then the output of the associative memory will be the zero vector. If the second condition is satisfied but not the first, then other semantic pointers will be incorrectly added into the output.

It should be clear from this requirement that the threshold is important to the success of the associative memory, but it is perhaps not clear why the dimensionality matters. The short answer is that the dimensionality of the vectors determines the statistics of the two quantities in Condition (5.6). We will demonstrate this by sampling from the distributions of these two quantities.

We begin by sampling from the distribution of the right-most quantity in Condition (5.6) for different values of N and different dimensionalities D. For each value of D, we begin by randomly selecting 100,000 random D-dimensional unit vectors to play the role of ID-vectors, and then 10 more to play the role of relation-type vectors. Then for each value of N, we draw 1000 samples by randomly selecting N ID-vectors and N relation-type vectors, constructing a semantic pointer $\mathbf{S_N}$ from them in the manner of Equation (5.5), and, finally, computing $(\mathbf{S_N} \circledast \overline{\mathbf{R_1}}) \cdot (\mathbf{x_1})_{\mathbf{id}}$. Histograms generated from this sampling procedure are given in Fig. 5.1. The most striking features of these plots are that for fixed N, the mean stays constant as D increases, whereas the variance decreases. Moreover, for fixed D, as N increases the variance stays constant while the mean decreases. Judging from this plot alone, we would seem to have a range of values for both the threshold and the dimensionality: for any distribution, simply pick a threshold that is below all the samples. However, this will change once we take into account the other constraint on the threshold.

To do this we augment our plot with samples from the distribution of the left-most quantity in Condition (5.6). For each value of N, we randomly select 250 of the semantic pointers generated in the previous process and compute:

$$\max_{\mathbf{z} \neq \mathbf{x_1}} [(\mathbf{S_N} \circledast \overline{\mathbf{R_1}}) \cdot \mathbf{z_{id}}] \tag{5.7}$$

These values are pooled and plotted in purple for each dimension. The augmented plots are shown in Fig. 5.2. From that plot it is clear that only once $D = 512$ is there a value of a threshold that separates the purple sample from the other samples with high-probability. Choosing $D = 512$ and threshold = 0.3 will give us very good performance with semantic pointers containing up to 7 terms, corresponding to WordNet synsets that are the source of up to 7 relations. We should also get acceptable performance with semantic pointers containing even more terms. While WordNet does contain synsets with more than 7 relations, they are rare and unlikely to affect overall performance significantly.

5.6 Extraction Algorithm

To summarize, we encode WordNet by assigning every synset two vectors: a randomly chosen ID-vector, and a semantic pointer encoding the synset's structural relations. Later, given a semantic pointer corresponding to a synset and some query vector corresponding to a relation-type, if the synset has a relation of the given type, then we can extract the semantic pointer for the target of

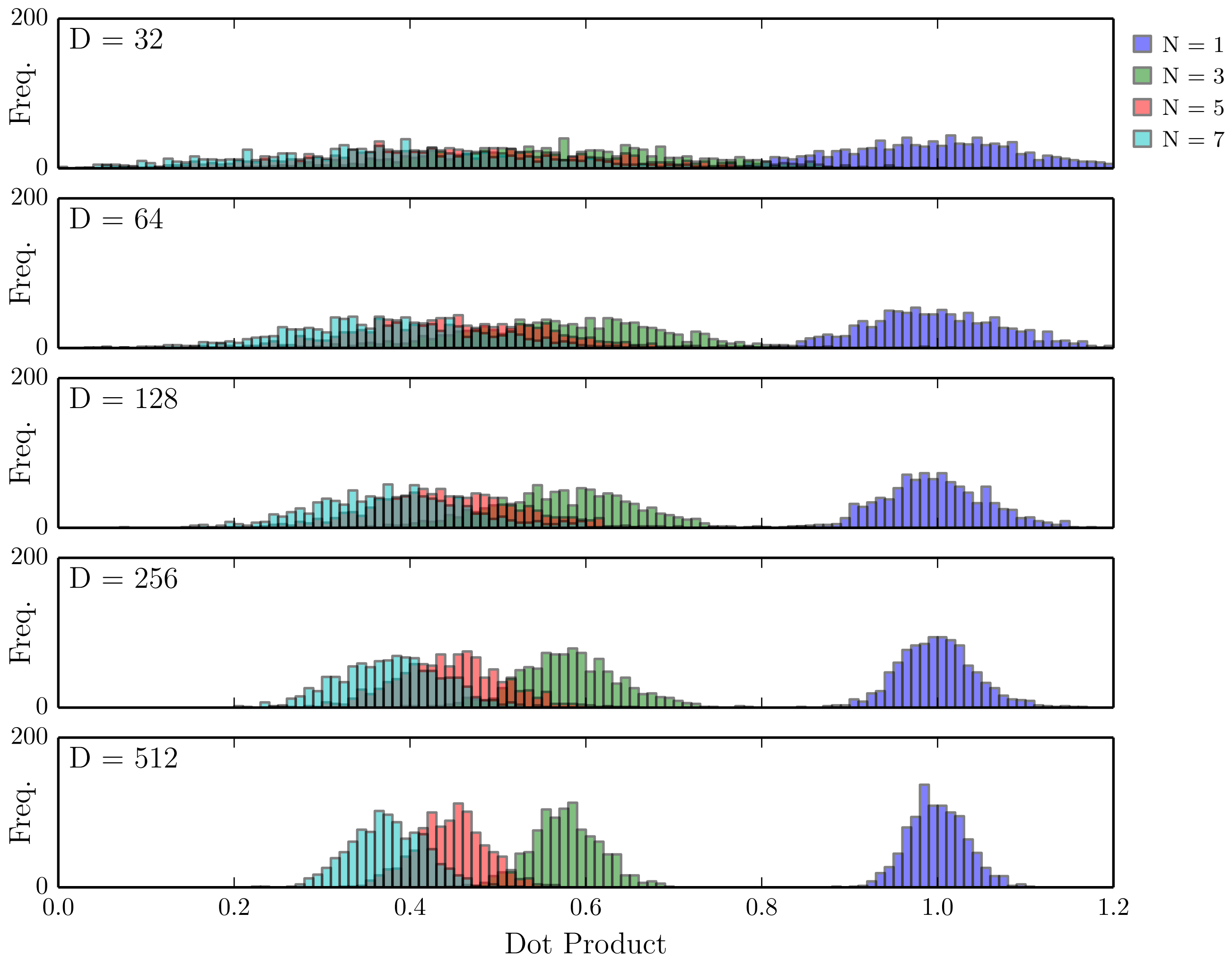


Figure 5.1: Distribution of dot products between vectors obtained through decompression and the original pre-compressed vector. D denotes the dimensionality of the involved vectors, and N denotes the number pairs in the semantic pointer being decompressed.

that relation with the following algorithm:

Algorithm 5.6.1: EXTRACTION(*sp*, *query*)

```
inv_query ← INVOLUTION( query )
noisy_id ← CIRCULAR-CONVOLUTION( sp, inv_query )
target_sp ← ASSOCIATE( noisy_id )
return ( target_sp )
```

This exact same algorithm can also be used to extract the constituents of semantic pointers encod-

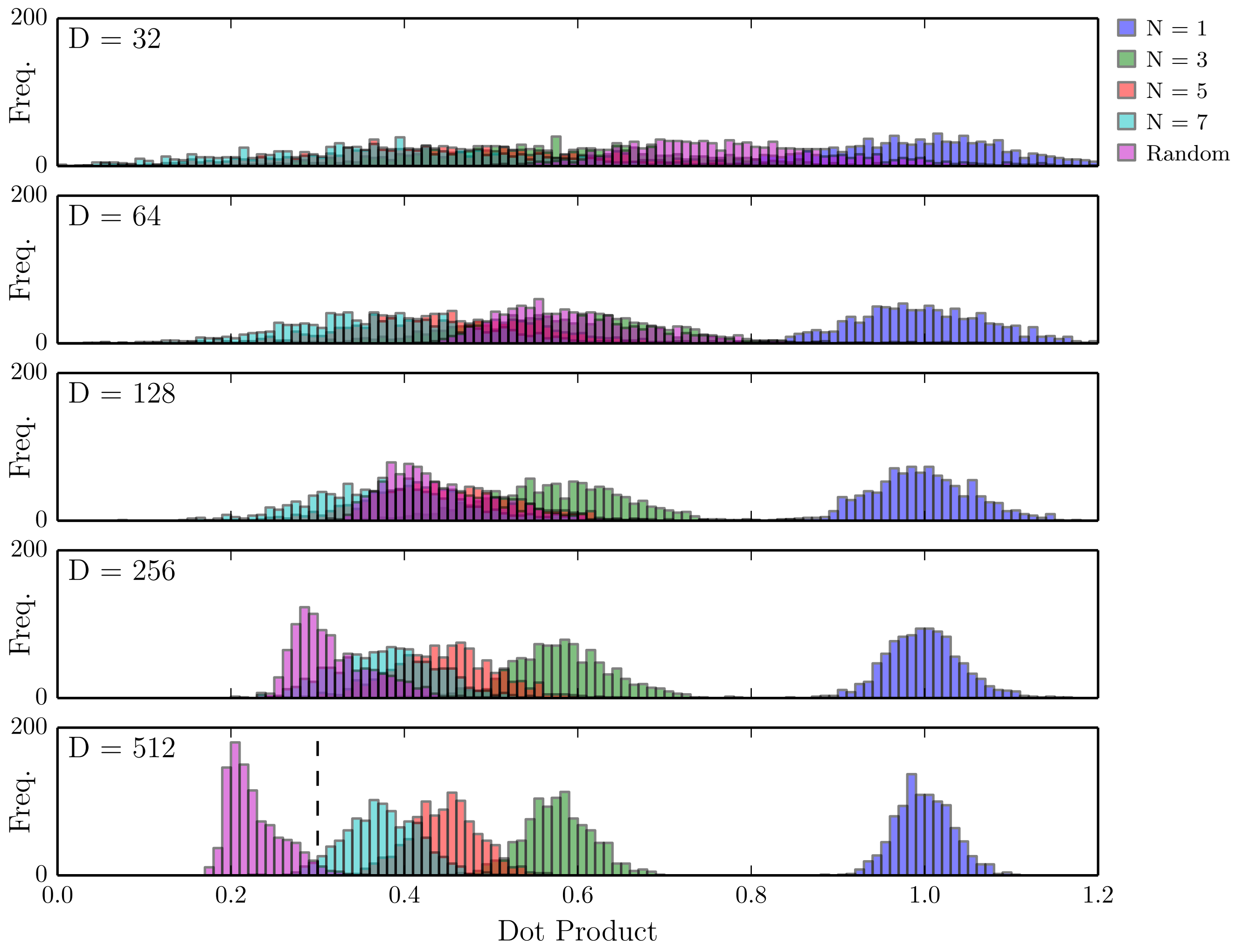


Figure 5.2: Same as Fig. 5.1, but including samples obtained by computing the dot-product of each decompressed vector with 100,000 random vectors and taking the maximum.

ing sentences composed of WordNet synsets.

One potential issue with this algorithm is the way in which it handles synsets that have multiple relations of the same type (which is not uncommon in WordNet). For example, the synset *lion* is related to both *pride* and *panthera* via the *member* relation-type. When the Extraction Algorithm is run with $\mathbf{lion_{sp}}$ and **member** as input, the output will be $\mathbf{pride_{sp}} + \mathbf{panthera_{sp}}$. This vector still contains all the relational structure of both *pride* and *panthera*, and can be used in further extractions without difficulty. Consequently, returning the sum of these two vectors in response to a *member* query is considered correct in the experiments we run in Section 7. We also note that the fact that this is an issue may be a quirk of WordNet; in human knowledge bases, concepts with multiple relations of the same type may be rare or non-existent. For example, these two member relations (*lion* $\rightarrow$ *panthera* and *lion* $\rightarrow$ *pride*) could be given two different relation types, reflecting the two different domains of knowledge they are concerned with. We discuss this in further detail in Section 9.3.

We now move on to neural implementation, and show that given a list of pairs of ID-vectors and semantic pointers encoding WordNet, we can construct a spiking neural network that performs the Extraction Algorithm.

Chapter 6

Neural implementation

Since our end goal is a scalable, biologically plausible system for representing and manipulating structured knowledge, our next step is to show how the Extraction Algorithm we have been discussing can be implemented in realistic spiking neurons. We begin by presenting a framework that provides a principled approach to constructing networks of spiking neurons that represent and transform high-dimensional vectors. We then show how this technique can be applied to create a network implementing both involution and circular convolution. Finally, we use a slightly more advanced application of this method to construct a neural associative memory which is able to map a noisy version of any ID-vector to the corresponding semantic pointer. These networks can be composed into a single network implementing the Extraction Algorithm from the previous section, permitting the representation and extraction of structured knowledge by biologically realistic neurons.

6.1 Neural representation and transformation

For the purpose of neural representation and computation, we employ the Neural Engineering Framework (NEF), a set of methods for building biologically plausible neural models [8]. These methods have been broadly employed to generate detailed spiking neural models of a wide variety of neural systems and behaviors, including the barn owl auditory system [12, 13], parts of the rodent navigation system [3], escape and swimming control in zebrafish [23], tactile working memory in monkeys [45], decision making in humans [28] and rats [25, 29], and the basal ganglia system [50, 48]. These methods also underlie the recent Spaun model, currently the world's largest functional brain model [9]. Here we present a brief discussion of the aspects of the NEF that are required for neural structured representation using HRRs. In particular, we discuss the NEF's principles of neural *representation* and *transformation*. The NEF also provides principles for *dynamics*, facilitating the implementation of arbitrary dynamical systems in recurrent neural networks, though these are not required for the feedforward networks used in the present model. All figures in this section were created using the Nengo neural simulation software package, which is available online at http://nengo.ca/.

The core idea behind the NEF's approach to neural representation is that the activity of a population of neurons at any given time can be interpreted as representing a vector. Importantly, the dimensionality of the represented vector is typically not equal to the size of the neural population. A typical case would have the activity of a population of 40 neurons representing a 2-dimensional vector. We now outline the details of the relationship between the activities of a neural population and the vector those activities are taken to represent.

Let E denote the dimensionality of the vectors that a given neural population is capable of representing. A basic assumption of the NEF is that each neuron in the population has a "preferred direction vector" of dimensionality E, essentially a direction in the population's "represented space" which the neuron responds to most strongly. For instance, this is a useful way to characterize the behavior of motor neurons. Georgopoulos found that neurons in motor cortex of rhesus monkeys have a preferred arm movement direction, the direction being different for each neuron [16]. These neurons become more active as the monkey's current arm movement direction approaches their preferred direction. The activities of these neurons, taken together, can be interpreted as representing the direction of arm movement in 3-dimensional physical space. This idea is quite intuitive in motor cortex, since the represented vector is directly observable; however, this notion is useful in general, and the NEF extends it to all neural representation.

To formalize the notion of preferred direction vector, the NEF assumes that the activity of the ith neuron in a neural population can be written:

$$a_i(\mathbf{x}) = G_i(\mathbf{e}_i\mathbf{x}) \tag{6.1}$$

where a_i is the activity of the neuron, G_i is the neuron's activation function, $\mathbf{e}_i$ is the neuron's preferred direction vector (a row vector) and $\mathbf{x}$ is the E-dimensional input to the neural population (a column vector). $\mathbf{e}_i\mathbf{x}$ is the dot product between the neuron's preferred direction vector and the input vector, and acts as a measure of similarity. The NEF works for arbitrary neural activation functions, so, in accordance with our goal of biological realism, we choose G_i to be a spiking leaky integrate-and-fire (LIF) activation function in all the work presented in this paper. The details of the LIF activation function can be found in the Appendix A.

Equation (6.1) is referred to as the *encoding* equation because it describes how an input vector, in this case $\mathbf{x}$, is encoded into the activities of a neural population. Preferred direction vectors will henceforth be called *encoding vectors* because of their central role in this process. When building networks using the NEF, the encoding vectors for a neural population are typically chosen uniformly at random from the unit hypersphere in the population's represented space. The encoding process is depicted in Fig. 6.1 for a neural population capable of representing 2-dimensional vectors. Note that while only 4 neurons are used for ease of presentation, it typically requires more than 4 neurons to accurately represent 2-dimensional vectors.

So far we have shown how a vector can be *encoded* into neural activities. However, to fully characterize neural representation, we also need to say something about *decoding* those neural activities. In other words, given the activities of a neural population, how do we reconstruct the vector that the population is representing? The NEF assumes that a linear decoding is sufficient for capturing information transfer between neural populations. Given $a_i(\mathbf{x})$ for $i \in 1 \ldots N$, the set

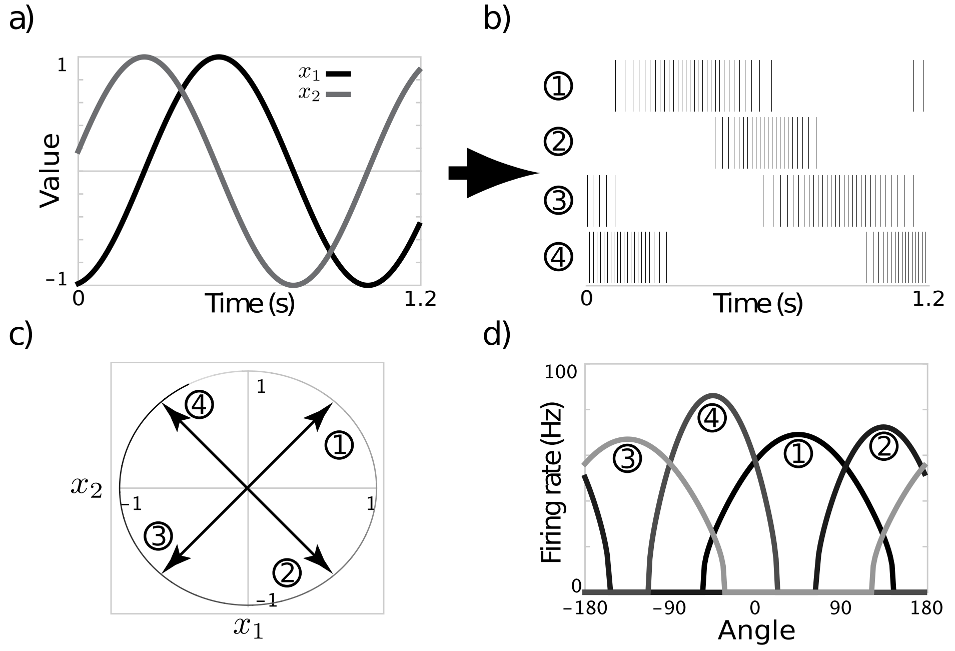


Figure 6.1: NEF encoding. A population of four neurons encoding a 2-dimensional vector. a) Both dimensions of the input to the neurons, plotted over a period of 1.2 seconds. The input vector is determined by $x_1 = \sin(6t)$ and $x_2 = \cos(6t)$. b) Spikes generated by four neurons driven by the input in a), according to the encoding equation (Equation (6.1)). c) A different visualization of the input in a). The input vector traces a clockwise path around a unit circle. Older inputs are in lighter gray. The encoding vectors of all four neurons are also shown. Comparing b) and c) shows that the neurons are most active when the input vector is closest to their encoding vectors. d) The firing rate tuning curves of all four neurons as a function of the angle between the input vector and the encoding vector. Parameters for G_i, the neural activation function, are randomly chosen for each neuron, which is why the tuning curves are different heights and widths. (Reproduced from [9], with permission).

activities of a neural population, the vector represented by that population can be approximately reconstructed as:

$$\hat{\mathbf{x}} = \sum_i a_i(\mathbf{x})\mathbf{d}_i \tag{6.2}$$

where $\hat{\mathbf{x}}$ is a reconstruction of $\mathbf{x}$, and the $\mathbf{d}_i$ are a set of appropriately chosen column vectors (one for each neuron) called *decoding vectors*. These decoding vectors all have dimensionality E.

Decoding vectors that provide the best reconstruction can be found through a least-squares optimization process, outlined in detail in Appendix A. Essentially, we find the $\mathbf{d}_i$ that minimize the equation:

$$\begin{aligned} Error &= \frac{1}{2}\int (\mathbf{x} - \hat{\mathbf{x}})^2 d\mathbf{x} \\ &= \frac{1}{2}\int (\mathbf{x} - \sum_i a_i(\mathbf{x})\mathbf{d}_i)^2 d\mathbf{x} \end{aligned} \tag{6.3}$$

This optimization is typically performed offline, before the network is instantiated.

The decoding vectors found by minimizing Equation (6.3) produce the optimal linear reconstruction of $\mathbf{x}$ from the activities of the neurons. In principle, however, we can also find decoding vectors that reconstruct $f(\mathbf{x})$, an arbitrary vector-valued function of $\mathbf{x}$. We denote these decoding vectors $\mathbf{d}_i^f$. The reconstruction of $f(\mathbf{x})$ is then computed from the activity of the neural population using the equation:

$$\widehat{f(\mathbf{x})} = \sum_i a_i(\mathbf{x})\mathbf{d}_i^f \tag{6.4}$$

These decoding vectors are found by minimizing:

$$\begin{aligned} Error &= \frac{1}{2}\int (f(\mathbf{x}) - \widehat{f(\mathbf{x})})^2 d\mathbf{x} \\ &= \frac{1}{2}\int (f(\mathbf{x}) - \sum_i a_i(\mathbf{x})\mathbf{d}_i^f)^2 d\mathbf{x} \end{aligned} \tag{6.5}$$

with respect to $\mathbf{d}_i^f$. In this more general case, the dimensionality of the decoding vectors is equal to the dimensionality of the range of the function f. The accuracy of the reconstruction depends on the type of function and the tuning curves of the neurons. See [8, section 7.3] for a discussion of this topic.

Thus far, we have primarily concerned ourselves with the question of neural representation, that is, how can an input vector be encoded in the activities of a neural population, and how can those activities be decoded to obtain a reconstruction of the encoded vector or a function thereof. However, representation alone is not terribly useful; to perform interesting information processing, we also need to be able to *transform* those representations. Fortunately, we've already defined the concepts required to understand the NEF's approach to neural transformation.

Suppose we have two neural populations A and B, and that there are all-to-all feedforward connections from the neurons in A to the neurons in B. Further suppose that we want to set the connection weights between A and B such that if, at a given time, A is representing some vector $\mathbf{x}$, then B will represent $f(\mathbf{x})$, where f is some arbitrary vector-valued function. In other words, we want to derive connection weights between the neurons in A and B such that $\widehat{f(\mathbf{x})}$ is first decoded from the activities in population A, and then encoded into the activities of population B. Conveniently, the NEF tells us that we can derive connection weights that achieve this in terms of the encoding vectors of B and decoding vectors of A for function f. Formally, we substitute $\widehat{f(\mathbf{x})}$ into Equation (6.1) (modified for population B):

$$\begin{aligned} b_j(\mathbf{x}) &= G_j(\mathbf{e}_j\widehat{f(\mathbf{x})}) && (6.6)\\ &= G_j(\mathbf{e}_j(\sum_i a_i(\mathbf{x})\mathbf{d}_i^f)) && (6.7)\\ &= G_j(\sum_i(\mathbf{e}_j\mathbf{d}_i^f)a_i(\mathbf{x})) && (6.8)\end{aligned}$$

where a_i is the activity of the ith neuron in A, and b_j is the activity of the jth neuron in B. Thus we have the activity of neuron j in B in terms of a weighted sum of the activities of the neurons in A. Following the standard interpretation of connection weights, this indicates that the weight from neuron i in population A to neuron j in population B should be:

$$\omega_{ij} = \mathbf{e}_j\mathbf{d}_i^f \tag{6.9}$$

which is simply the dot product, or similarity, between $\mathbf{e}_j$ and $\mathbf{d}_i^f$.

As a final note on transformation, if we additionally want to perform a linear transformation, represented by a matrix $\mathbf{L}$, on $f(\mathbf{x})$ (i.e. we want $\mathbf{L}f(\mathbf{x})$ to be represented in population B), then we can simply include $\mathbf{L}$ in the weight equation as follows:

$$\omega_{ij} = \mathbf{e}_j\mathbf{L}\mathbf{d}_i^f \tag{6.10}$$

This is the general weight equation for computing any combination of linear and non-linear functions between two neural populations.

To summarize, the process for creating two populations of neurons A and B, and deriving connection weights from A to B such that $\mathbf{L}f(\mathbf{x})$ is represented by population B whenever $\mathbf{x}$ is represented by population A, is as follows:

1. Create the neurons in populations A and B. For each neuron in each population, randomly choose parameters for the neural activation function and an encoding vector from the unit hypersphere.

2. Calculate decoding vectors for population A that minimize Equation (6.5).

3. Calculate the weight matrix between A and B using Equation (6.10).

Simulations of spiking neural networks with connection weights derived using this technique are

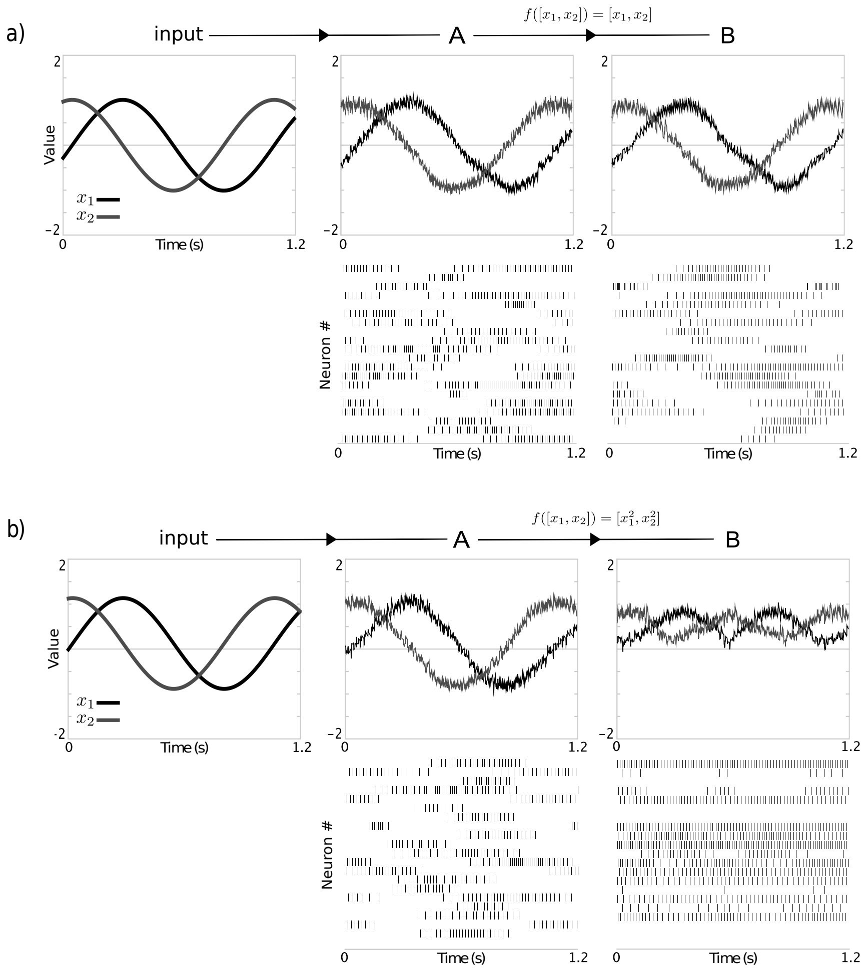


Figure 6.2: Using NEF-derived connection weights (Equation (6.9)) to compute functions between neural populations representing 2-dimensional vectors. a) Computing the identity function between A and B. b) Computing the element-wise square between A and B. These simulations are 1.2 seconds long, and the input vector is determined by $x_1 = \sin(6t)$ and $x_2 = \cos(6t)$. Both populations have 20 neurons, with randomly chosen encoding vectors and parameters for the neural activation function G_i.

depicted in Fig. 6.2 for the identity function and the element-wise square function, with $\mathbf{L}$ set to the identity matrix in both cases.

This brief discussion does not capture the generality of the NEF, although it is sufficient for characterizing neural structured representation. Since we are concerned with scaling, an important final note is that as more neurons are added to a population, the quality of its representation improves. Specifically, the mean-squared-error goes down as $1/N$ [8]. Consequently, representations and transformations can be implemented to any desired precision, as long as there is a sufficient number of neurons. One of the main concerns of this paper is to determine whether the transformations and representations necessary for representing human-scale lexical structure can be done with a reasonable number of neurons. We now show how the NEF can be applied to create spiking neural networks that compute the operations required by the Extraction Algorithm.

6.2 Circular convolution in spiking neurons

Like any kind of convolution, circular convolution can be formulated as an element-wise multiplication in the Fourier space. Since both the Fourier transform and its inverse are linear operators, circular convolution can be written in terms of linear operators and an element-wise multiplication:

$$\mathbf{x} \circledast \mathbf{y} = \mathbf{F}^{-1}(\mathbf{F}\mathbf{x} \diamond \mathbf{F}\mathbf{y})$$

where $\mathbf{x}$ and $\mathbf{y}$ are two arbitrary input vectors, $\diamond$ indicates an element-wise multiplication, and $\mathbf{F}$ and $\mathbf{F}^{-1}$ are matrices computing the Fourier transform and its inverse, respectively. A neural network that computes circular convolution using this formulation is shown in Fig. 6.3. Populations A and B represent the two input vectors, and have feedforward connections to population C. These connections are set up to compute the Fourier transform by multiplying by $\mathbf{F}$, causing C to represent the concatenation of the two Fourier transformed vectors. C is connected to population D, and these connections simultaneously compute the element-wise product of the two Fourier transformed vectors (using the appropriate decoding vectors), and take the inverse Fourier transform of the result by multiplying by $\mathbf{F^{-1}}$. The result is that $\mathbf{x} \circledast \mathbf{y}$ is represented in D.

The weights for this network are found using Equation (6.10). Here and throughout the paper, when showing connection weights, we will use i to index the neurons in the upstream (pre-synaptic) population, and j to index neurons in the downstream (post-synaptic) population. The weights for the convolution network are:

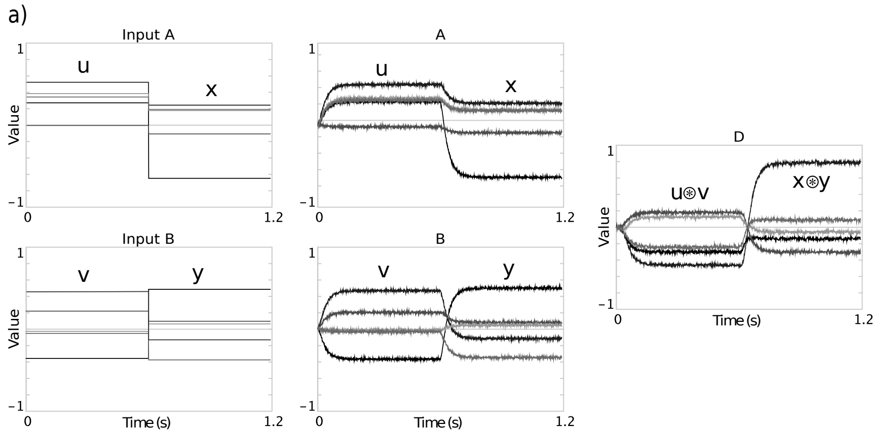
$$\begin{aligned}
\omega_{ij}^{A \to C} &= \mathbf{e}_j \mathbf{F}_1 \mathbf{d}_i \\
\omega_{ij}^{B \to C} &= \mathbf{e}_j \mathbf{F}_2 \mathbf{d}_i \\
\omega_{ij}^{C \to D} &= \mathbf{e}_j \mathbf{F}^{-1} \mathbf{d}_i^{\diamond}
\end{aligned}$$

where $d_i^{\diamond}$ are decoding vectors for the element-wise multiplication function for population C, and $\mathbf{F}_1$ and $\mathbf{F}_2$ are Fourier transform matrices padded with 0's so that $\mathbf{F}\mathbf{x}$ and $\mathbf{F}\mathbf{y}$ are concatenated in C, rather than added.

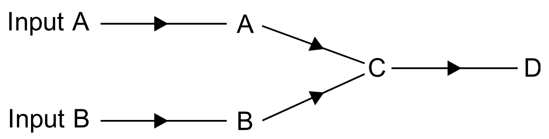
Although this operation may seem complicated, it is surprisingly natural for neural computation in several important respects. First, it has been shown to be learnable in a spiking network [49]. Second, for 512-dimensional vectors it has been shown to result in connection matrices that respect known neural connectivity constraints [7].

6.3 Involution in spiking neurons

As we saw in our initial discussion of the HRR algebra, the involution of a vector is an approximate inverse with respect to circular convolution. It can be computed by reversing all but the first element of the vector, which stays in place. Since this is a permutation, it is a linear operation, and therefore there exists a matrix $\mathbf{V}$ which computes it. Consequently, there is a straightforward a)



b)



c)

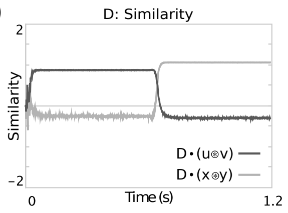


Figure 6.3: Simulation of a neural circuit for computing circular convolution of two 10-dimensional vectors. A, B, C and D are populations of spiking neurons. Connection weights between populations are derived using the techniques from Section 6.2 such that D represents the circular convolution of the two input vectors. Input vectors $\mathbf{u}, \mathbf{v}, \mathbf{x}$ and $\mathbf{y}$ were randomly chosen from the 10-dimensional unit hypersphere. a) Graphs showing input vectors and vectors represented by populations A, B and D over a 1.2 s simulation, where the inputs change after 600 ms. In each graph, only 5 of the 10 dimensions are shown to reduce clutter. First column: the input vectors. Second column: vectors represented by the populations A and B, neural representations of the input vectors. Third column: vector represented by population D, which should be the circular convolution of the two input vectors. b) Architecture of the neural circuit. The letters are populations of neurons and the arrows are all-to-all neural connections. c) Similarity between the vector represented by population D and the vectors $\mathbf{u} \circledast \mathbf{v}$ and $\mathbf{x} \circledast \mathbf{y}$ over the 1.2 s simulation. If the circuit is correctly computing circular convolution, we would expect the similarity to $\mathbf{u} \circledast \mathbf{v}$ to be near 1 before 600 ms, and the similarity to $\mathbf{x} \circledast \mathbf{y}$ to be near 1 after 600 ms, which is clearly the case.

modification we can apply to our convolution network such that the second input is involuted before the Fourier transform is applied, resulting in a network that computes $\mathbf{x} \circledast \bar{\mathbf{y}}$ rather than $\mathbf{x} \circledast \mathbf{y}$. Specifically, we change the connection weights between populations B and C to:

$$\omega_{ij}^{B \to C} = \mathbf{e}_j \mathbf{F}_2 \mathbf{V} \mathbf{d}_i \tag{6.11}$$

We now have a neural network computing 2 of the 3 operations required by the Extraction Algorithm. The last component is a neural associative memory, which requires a slightly more nuanced application of the NEF.

6.4 Neural associative memory

Recall that the aim of the associative memory is to associate pairs of vectors $\langle \xi, \eta \rangle$, and for our purposes the ξ's are the ID-vectors and the η's are the semantic pointers. We now show how the NEF can be applied to create a spiking neural network capable of efficiently implementing the associative memory functionality. The approach we employ here was first demonstrated by Stewart, Tang and Eliasmith in [52] to build an *auto-associative* memory (where for each stored pair $\langle \xi, \eta \rangle$, $\xi = \eta$), though it can be trivially extended to implement a general hetero-associative memory. That paper demonstrated that networks built using this approach significantly outperform linear associators, direct function approximators and standard multi-layer perceptrons in terms of both accuracy and scalability. Such networks also have an advantage in terms of biological plausibility, since they are implemented in spiking neurons.

This neural associative memory is essentially a neural implementation of the association algorithm presented in Section 5.4. Each pair to be associated is assigned a small ($\sim$20) neuron population. The encoding vectors of this population are set equal to ξ, and the neurons are given a high threshold so that they only spike when the input is sufficiently similar to ξ. The decoding vectors of the population are chosen to approximate a thresholding function, and η is used as a linear operator. The overall effect is that when a population is active, it outputs its assigned η, and a population is active if and only if the input is sufficiently similar to its assigned ξ. All these populations converge on a single output population, where the inputs are summed by the dendrites. In essence, each neural population computes, in parallel, one iteration of the loop in Algorithm 5.4.1. To be explicit, for sub-population k assigned the vector pair $\langle \xi_k, \eta_k \rangle$, the input and output weights are:

$$\omega_{ij}^{in} = \xi_k \mathbf{d}_i \qquad\qquad \omega_{ij}^{out} = \mathbf{e}_j \eta_k \mathbf{d}_i^{thresh} \tag{6.12}$$

where $\mathbf{d}^{thresh}$ are decoding vectors for the thresholding function $f(x) = 1.0$ **if** $(x > 0.3)$ **else** 0.

We have described all of the techniques required to create a spiking neural network for extracting the constituents of semantic pointers. We claim that the network obtained by composing these two networks, such that the output of the involution/convolution network is fed into a neural associative memory, constitutes a neural implementation of the abstract Extraction Algorithm. In what remains, we present the details of the neural model, and run experiments on it to determine how it performs at scale.

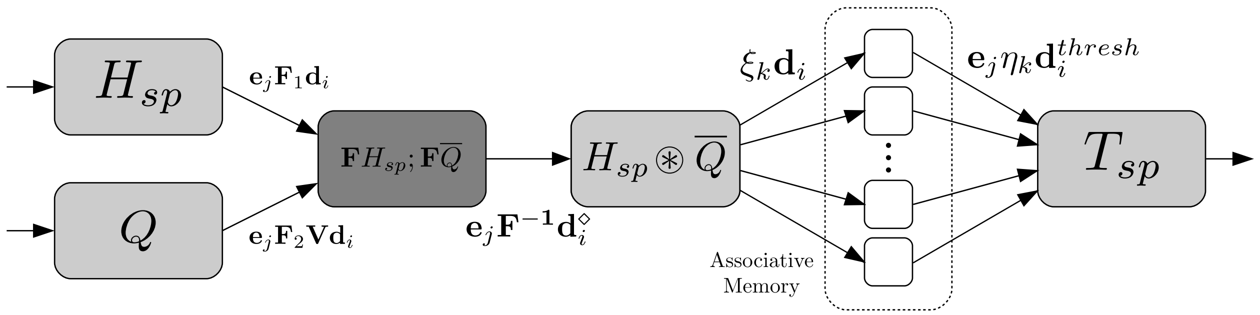
6.5 The complete neural model

The model consists of a network of 2,506,980 spiking neurons constructed using the techniques outlined above. Given a semantic pointer corresponding to a WordNet synset and a query vector corresponding to a relation-type, the network returns the semantic pointer corresponding to the target of the relation, implementing the Extraction Algorithm. The network can be used to traverse the WordNet hierarchy by running it recursively, with the output of one run used as input on the next run. Low-level details about the model and its parameters can be found in Appendix A.

A schematic diagram of the model is depicted in Fig. 6.4. The rectangles correspond to populations of spiking neurons which represent and manipulate high-dimensional vectors. The dark gray population, which represents the concatenation of the Fourier transforms of the two input vectors, contains 51,400 neurons, and the 4 light gray populations contain 25,600 neurons each. The associative memory contains a separate 20-neuron population for each of the 117,659 synsets in WordNet. The grand total is thus 2,506,980 neurons, equivalent to approximately 14.7 mm^2 or 0.147 cm^2 of cortex (as there are about 170,000 neurons per mm^2; 7). This is much smaller than any of the approaches discussed in Section 2, all of which require on the order of 500 cm^2 of cortex or more. More significantly, our approach is the only one whose neural resource requirements do not contradict our empirical knowledge about the size of relevant brain areas. Consequently, if our experiments confirm that our network can accurately extract the relational structure from WordNet synsets, it will constitute a significant advance in the study of biologically plausible representations of structured knowledge.

The tasks of moving the output into the input for hierarchical traversals, controlling which vector is used as input to the query population, etc., are not neurally implemented here as they are peripheral to our central concern of representing human-scale structured knowledge in a biologically plausible manner. However, Spaun, a large scale, functional brain model constructed using the NEF, is evidence that it is possible to achieve this kind of control in a scalable spiking neural network [9].

Our model is a significant departure from the majority of connectionist work in that no online learning occurs; the connection weights implementing the required transformations are derived offline before the network is instantiated, using the NEF techniques. While we will eventually have to account for how our network could be learned in a biologically plausible manner, we believe the problem of large-scale connectionist knowledge representation is difficult enough that it is sufficient to focus on representation alone for now, and leave the question of learning for future work. We do note that it has been shown that circular convolution can be learned in spiking neurons using a biologically plausible learning rule [49], and later in this thesis we make some preliminary investigations into the question of learning the type of associative memory used here.



H_{sp}
Q
$\mathbf{e}_j\mathbf{F}_1\mathbf{d}_i$
$\mathbf{e}_j\mathbf{F}_2\mathbf{V}\mathbf{d}_i$
$\mathbf{F}H_{sp};\mathbf{F}\overline{Q}$
$\mathbf{e}_j\mathbf{F^{-1}}\mathbf{d}_i^{\diamond}$
$H_{sp}\circledast\overline{Q}$
$\xi_k\mathbf{d}_i$
Associative Memory
$\mathbf{e}_j\eta_k\mathbf{d}_i^{thresh}$
T_{sp}

Figure 6.4: The network of spiking neurons designed to implement the Extraction Algorithm from Section 5.6. Assume $\mathbf{H_{sp}} = \mathbf{Q} \circledast \mathbf{T_{id}} + \mathbf{R} \circledast \mathbf{U_{id}}$. The rectangles correspond to populations of spiking neurons, and are labeled with the values we expect them to represent when the network is given $\mathbf{H_{sp}}$ and $\mathbf{Q}$ as input. Arrows represent all-to-all feedforward connections between populations, and are labeled with the elements of the NEF-derived weight matrices mediating them. In these weight matrices, *i* always indexes neurons of the upstream population, *j* always indexes neurons of the downstream population, and *k* indexes pairs of vectors in our vector encoding of WordNet. Light gray populations represent 512-dimensional vectors. The dark gray population represents the concatenation of two Fourier transformed 512-dimensional vectors, and thus represents a 1028-dimensional vector.

Chapter 7

Experiments

We performed three experiments on both the abstract Extraction Algorithm and its neural implementation, to test whether WordNet is accurately encoded. For convenience, we will refer to both implementations as "models". A *trial* consists of using a model to answer a single question about the WordNet graph (the question is different for each experiment). A *run* consists of a group of trials. No information was added to or removed from either model's associative memory between experiments, demonstrating that both models are capable of performing all three tasks unmodified.

For each experiment we execute 20 runs, calculate the performance on each run as the percentage of trials on which the model answered correctly, and report the mean performance over all the runs. To obtain distributional information about these results, we employ a bootstrapping method to obtain 95% confidence intervals.

Each trial consists of using a model for one or more extraction operations, where a semantic pointer and a query vector are presented as input and the algorithm outputs a vector (which may or may not be a semantic pointer). In the neural case, for each extraction operation the model was simulated for 100 ms with a simulation timestep of 1 ms, after which the vector represented by the rightmost population in Fig. 6.4 was taken to be the output of the model. Code for constructing the models and running the experiments is hosted online in a github repository at https://github.com/e2crawfo/hrr-scaling.

7.1 Experiment 1 - Simple Extraction

This experiment investigates the ability of a model to traverse a single edge in the WordNet graph. We present the model with a semantic pointer corresponding to a randomly chosen synset and the vector corresponding to a relation-type that the synset is known to possess, and see if the model outputs the semantic pointer corresponding to the target of that relation. For example, we might present the model with $\mathbf{dog}_{\mathbf{sp}}$ as the semantic pointer and **class** as the query vector, and expect the model to return $\mathbf{canine}_{\mathbf{sp}}$.

To be considered correct, the vector returned by the model must have a larger dot product with the correct semantic pointer than with any incorrect semantic pointer in the vocabulary, and this similarity must exceed a threshold of 0.7. The value of 0.7 is somewhat arbitrary, though it does ensure that the output vectors have sufficient fidelity to be put to further use. In this experiment, each run consists of 100 trials.

7.2 Experiment 2 - Hierarchical Extraction

The simple extraction experiment assesses the general accuracy of a model of knowledge representation, but it only tests individual relationship links. The hierarchy traversal experiment is designed to test a model's ability to traverse hierarchies of arbitrary depth in the WordNet graph.

To that end, we use the model to answer the following question: given a starting synset, a goal synset and a relation-type, can the goal synset be reached from the starting synset by following only links of the specified type? To have the model answer this question, we present it with the semantic pointer corresponding to the starting synset as well as the vector for the given relation-type. We then run the model and compare the output vector to the semantic pointer for the goal synset. If they are the same (their dot product is above a fixed threshold), then the model responds with a Yes. If not, we feed the output vector back into the model as the new semantic pointer and run the model again using the same query vector. This process is repeated until the model returns a vector with a norm below a fixed threshold, in which case the model responds with a No.

As an example, if the starting synset is *dog* and we follow only relations whose type is *class*, we first get *canine*, which in turn yields *carnivore*, followed by *placental mammal*, and so on, until the synset *entity* is finally reached in thirteen links. The correct answer is Yes if and only if the goal synset is one of these synsets. Further concrete examples of possible queries and correct responses are given in Table 7.1. Our tests were performed using only the *class* relation-type as it is the most prominent in WordNet and permits the deepest traversals. Each run consists of 40 trials with an even split between positive and negative instances. A positive instance is one in which the goal synset can be reached from the starting synset in the WordNet graph, and the correct response is Yes. Results of a simulation where we test the neural model on an instance of the Hierarchical Extraction test are shown in Figure 7.1.

Starting Synset	Target Synset	Relationship Type	Is Related?
dog	vertebrate	class	Yes
vertebrate	dog	class	No
dog	entity	class	Yes
dog	cat	class	No

Table 7.1: Examples of instances and correct responses in the Hierarchical Extraction test.

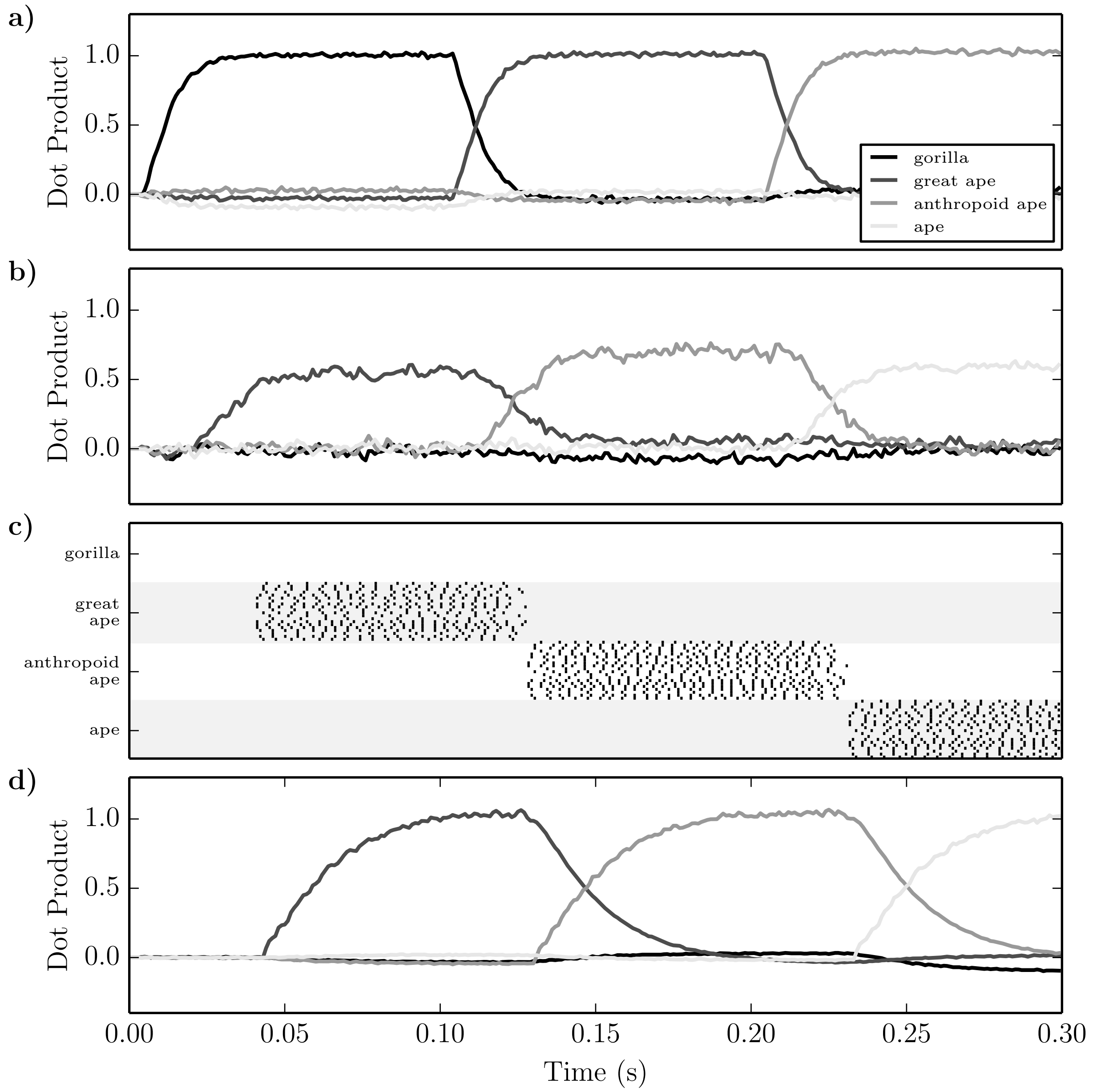


Figure 7.1: Results of a simulation during which the network was used to perform the Hierarchical Extraction test. The starting synset is *gorilla*, the relation type is *class*, and the target synset is *ape*. Every 100 ms we compute the norm of the output vector and the dot product between the output vector and $\mathbf{ape_{sp}}$. If the norm is below a threshold, the network's response is No. If the dot product is above a threshold, the network's response is Yes. Otherwise, the output vector is fed back in as input, and the simulation continues. **a)** Dot product between semantic pointers and the vector represented by the input population. **b)** Dot product between ID-vectors and the vector represented by the population that feeds into the associative memory. **c)** Spikes rasters for neurons in the association populations corresponding to the synsets listed in the legend. Association populations for all other WordNet synsets were included in the simulation, but had no spiking activity. **d)** Dot product between semantic pointers and the vector represented by the output population. The network can be seen to traverse the WordNet graph, starting from *gorilla* and eventually reaching *ape*. At 300 ms, the dot product of the output vector with $\mathbf{ape_{sp}}$ is above the threshold and the network responds (correctly) with Yes.

7.3 Experiment 3 - Extracting from Sentences

The previous experiments have focused on relations included in WordNet. The current experiment is designed to go further, and demonstrate that our representation is capable of encoding arbitrary sentence-like constructions possessing recursive structure, and that the models are capable of extracting the constituents of such sentences without modification.

If each role is assigned a random vector, then sentences can then be encoded as semantic pointers as we have done throughout the paper. One difference here is that role vectors are chosen randomly from the class of so-called "unitary" vectors instead of from the unit hypersphere as with relation-type vectors. Unitary vectors have the special property that involution is their *exact* inverse [40], which permits the sentence constituents to be extracted with higher accuracy.

For instance, recall that the sentence "mice believe that dogs chase cats" can encoded as:

$$\begin{aligned}\mathbf{deep\_sentence_{sp}} = {} & \mathbf{subject} \circledast \mathbf{mouse_{id}} + \mathbf{verb} \circledast \mathbf{believe_{id}} + \\ & \mathbf{object} \circledast (\mathbf{subject} \circledast \mathbf{dog_{id}} + \mathbf{verb} \circledast \mathbf{chase_{id}} + \mathbf{object} \circledast \mathbf{cat_{id}}) \end{aligned} \tag{7.1}$$

While this clearly leaves out many features of natural language sentences, the purpose of this experiment is not to validate this particular structure as a basis for linguistics. Rather, the goal is to confirm that the method of knowledge representation being tested is flexible enough to allow the elements of the vocabulary to bind to arbitrary roles in recursively structured sentences while still encoding the thousands of relationships between concepts.

Each trial begins by randomly generating a sentence in symbolic form. To generate the surface-level construction, we randomly select roles for inclusion according to the probabilities in Table 7.2. One of the included roles is randomly chosen to be filled by an embedded clause, and the rest are filled with randomly chosen WordNet synsets. The embedded clause is then generated in the same manner, except now *all* roles are filled by WordNet synsets. Once the sentence has been generated symbolically, the next step is to create a semantic pointer representing the sentence in terms of ID-vectors for the chosen synsets and role vectors for the included roles, yielding a vector similar in form to Equation (7.1). Finally, the resulting semantic pointer is normalized.

We then test the model's ability to extract the constituents of the sentence. For the surface-level constituents, we present the model with the semantic pointer representing the sentence, and the appropriate role vector as the query vector. For example, if we present $\mathbf{deep\_sentence_{sp}}$ and **verb**, we should expect the model to return $\mathbf{believe_{sp}}$. The process is similar for constituents of the embedded clauses, except we use compound query vectors. If we present the model with $\mathbf{deep\_sentence_{sp}}$ and $\mathbf{object} \circledast \mathbf{subject}$, we expect it to output $\mathbf{dog_{sp}}$. Each run consists of 30 trials, and on each trial the performance is measured as the percentage of queries that the model responded to correctly, using the same correctness criteria as in Experiment 1. Results for surface and embedded constituents are reported separately.

Role name	Probability of occurrence	Part of speech
subject	1.0	noun
object	0.8	noun
verb	1.0	verb
adverb	0.6	adverb
subject adjective	0.3	adjective
object adjective	0.3	adjective

Table 7.2: Roles for sentence generation and their properties.

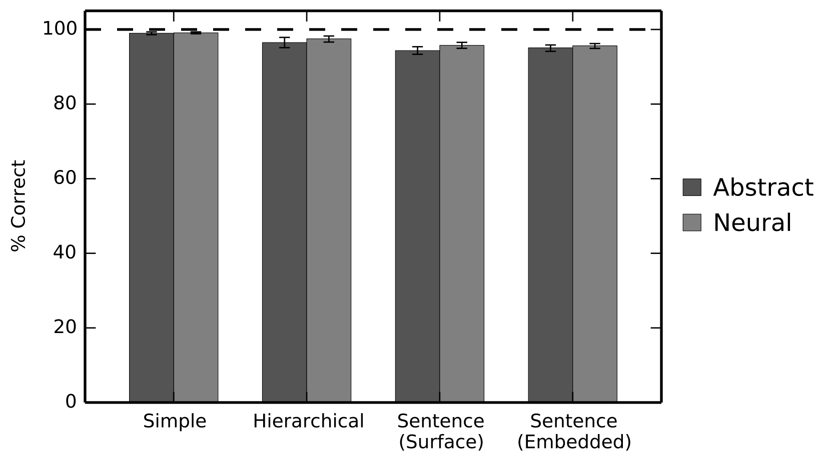


Figure 7.2: Extraction performance. Error bars are 95% confidence intervals.

7.4 Results

Results of the experiments are presented numerically in Table 7.3 and graphically in Fig. 7.2. Performance on all three tasks by both the abstract and neural implementations of the Extraction Algorithm is near 100%. The success of the abstract algorithm shows that WordNet is accurately stored in our vector encoding. The success of the neural implementation shows that no significant penalty is incurred by implementing the algorithm in spiking neurons. More generally, it shows that a spiking neural network is capable of extracting structure from any element of a human-scale vocabulary. Combined with the fact that, unlike previous approaches, this network places neural resource demands that are consistent with anatomical data, this constitutes the first biologically plausible neural implementation of a human-scale structured knowledge base.

Experiment	Type	% correct	95% CI lower	95% CI upper	Runs	Trials (per run)
Simple	Abstract	99.0	98.6	99.4	20	100
	Neural	99.2	98.9	99.3		
Hierarchical	Abstract	96.5	95.0	97.8	20	40
	Neural	98.5	97.8	99.3		
Sentence (surface)	Abstract	94.3	93.3	95.3	20	30
	Neural	97.2	96.3	97.9		
Sentence (embedded)	Abstract	95.1	94.2	95.9	20	30
	Neural	96.2	95.3	97.0		

Table 7.3: Numerical values for extraction performance

Chapter 8

Discussion

8.1 Scaling

We have presented the details of our approach to encoding structured representation and demonstrated empirically that it is capable of encoding a human-scale knowledge base with much more modest resource requirements than its competitors. Specifically, our model uses roughly 2.5 million neurons to encode a structured lexicon containing 117,659 words. Individual relations in the lexicon can be traversed with 99% accuracy, hierarchies with up to 13 levels can be traversed with 98% accuracy, and the network can be used to extract constituents of recursively structured sentences with 96% accuracy. Unlike past approaches, which use a minimum of 480 cm^2 of cortex to represent much simpler structures, the method as demonstrated here requires less than 1 cm^2 of cortex. Like past work, there are many aspects of linguistic processing that are not captured by our model. However, its modest use of cortical resources makes it plausible that it may do so with further development.

We believe that this improved scaling largely results from capturing structured representation using compressed, temporary signal processing states (i.e. semantic pointers encoding sentences or synsets), rather than using fixed neural resources. In contrast, synchrony-based approaches like DORA and LISA require a fixed neural node for every proposition that we require the network to be able to represent. The Neural Blackboard Architecture alleviates this need by allowing bindings to be represented by the activation states of a neural "mesh". However, the substantial complexity required to implement this mesh results in scaling that is still implausible. Moreover, both synchrony-based approaches and the NBA are capable of representing only a few propositions at a time. This means that while they are capable of representing short-term bindings (such as the sentences in our sentence experiment), the long-term storage and retrieval of relations in a large, structured knowledge base is well-beyond their abilities. In particular, it is unclear how they would accomplish the Hierarchical Extraction experiment.

Finally, our approach bears some resemblance to Smolensky's tensor product vector symbolic architecture. However, we make a different set of tradeoffs which result in significantly improved scaling. In particular, we use a compressive binding operator, supported by an associative memory. This avoids the explosion in neural resources required by Smolensky's approach as the depth of encoded structures increases. In particular, it permits us to encode and extract the constituents of recursively structured sentences without making implausible neural resource demands.

Our approach is largely a distributed one, The representations of the vectors in all populations other than the associative memory are distributed throughout the neurons in the population, and the contents of each term in the HRR vectors is distributed throughout the elements of the vector; the vector elements themselves are not semantically interpretable. Consequently, it inherits the scaling advantages of distributed representations; the populations outside the associative memory can represent any vector that is fed into them. The associative memory is significantly more localist, as we explicitly assign a group of neurons for every synset in WordNet. However, we are able to avoid the explosive scaling of previous localist schemes because each localist population is performing a simple job, that of comparing and thresholding, and thus each requires only a small number of neurons (20 in our model). Perhaps more significantly, we do not represent relations between synsets by explicit physical connections between the populations corresponding to those concepts; rather, the relations between synsets are encoded in the HRR vectors.

8.2 Theoretical Considerations

Our model speaks to the long-standing debate regarding the relationship between classical, symbolic theories of mind and connectionist research. On one side of the debate are researchers who take an implementational view of connectionism. These researchers hold that human behavior is best analyzed at a symbolic level, and that the role of connectionist research is to show how neurons might implement classical symbolic representations [14, 22]. The past approaches we discussed in Section 2 are all attempts to directly implement these classical symbols, where each element (e.g. a word) in a composite symbol (e.g. a sentence) is explicitly represented. On the other side of the debate are so-called eliminative connectionists who hold that human behavior can be accounted for without implementing a classical symbol system [2, 41, 10]. Upon first inspection, the approach we have presented here may appear implementational, since our neural network is essentially an implementation of the abstract Extraction Algorithm. However, semantic pointers, while possessing compositional properties, are not truly classical symbols, and consequently, we take our approach and its scalability as evidence in favor of non-classical architectures.

One reason that our approach is non-classical is that it is not necessary to perform the extraction operation on semantic pointers in order to do useful things with them, thanks to the shallow semantics that we mentioned in Section 3. For instance, semantic pointers that have similar relational structure will themselves be similar, which can be exploited to perform useful computation without first extracting the elements in the representation [7]. This is a hallmark of eliminative connectionism [2], and is, by definition, impossible in a classical symbolic architecture.

Another reason that we consider semantic pointers to be non-classical is that they are constructed through a lossy compression process. As a result, they do not contain complete information about their constituents. In particular, the result of the decompression operation (i.e. the involution/circular convolution combination) is only an approximation of a constituent vector. In contrast, tensor product vectors representing complex structures contain explicit representations of their constituents, and said constituents can be perfectly extracted. Consequently, McLaughlin's argument that tensor products are merely an implementation of a classical symbolic system [31] does not carry over to semantic pointers.

We suspect that, in fact, the poor scaling of the past approaches is a direct result of their use of classical representations. For instance, it is precisely because tensor products include complete representations of every item in the structure being represented that they scale poorly with the depth of the structure. On the other hand, because semantic pointers are created from their constituent vectors through lossy compression, the dimensionality of the representation remains constant as the depth of the encoded structure increases. This permits deep structures to be efficiently encoded, as we saw in the sentence extraction experiment. We are able to correct for the information lost through compression in a scalable manner using an associative memory.

One benefit of this kind of compressed representation is that models employing them have natural limits on the depth of structure they can encode. Thus, there is no need to appeal to a competence/performance distinction when theory and data differ. It is expected, rather, that the performance of theoretical models will reflect the actual observed performance of human subjects. For example, we have used these representations to capture human error rates as a function of list length in serial working memory tasks [9]. However, much work remains to be done to demonstrate that model and human performance will match across a wide variety of tasks.

8.3 Psychological plausibility

We do not believe that the model presented here is able to support significant claims of psychological plausibility, other than the very general observation that the our method can be used to model lexical processing at a psychological scale. In short, this work is best interpreted as a proof of principle, demonstrating that very large structured representations can be efficiently encoded in a realistic neural network using our method. It is tempting to make more specific psychological claims. However, our choice of WordNet as a lexical structure makes such claims implausible. WordNet was chosen because it is a readily available human-scale, structured representation that is intended as a lexical database of the English language, bearing a resemblance to the human conceptual system.

However, we remain uncommitted to WordNet from a psychological perspective because of its significant limitations. For instance, human conceptual systems likely employ relation-types that do not appear in WordNet, may contain concepts that WordNet omits, and potentially has high-degree concepts in WordNet broken down into intermediate concepts. In addition, there are many concepts in WordNet that are unlikely to be in an average person's conceptual system, either because they are domain-specific (e.g. *Gram's Method*, a staining technique used to classify bacteria) or culture-specific (e.g. *eisteddfod*, any of several annual Welsh festivals involving artistic competitions). These limitations force the model to perform some of the tasks in ways that may not be psychologically plausible. For instance, in performing a trial in the Hierarchical Extraction experiment where the model has to decide whether *dog* has the type *mammal* (i.e. whether *mammal* can be reached from *dog* via the *class* relation-type), the model must traverse *canine*, *carnivore* and *placental mammal* along the way. It is unlikely that this same traversal would occur for most human subjects.

In sum, we believe that the fact that our model is able to encode the WordNet semantic network in a reasonable number of neurons lends support to our technique, but we are not convinced that the specific lexical structure proposed by WordNet is psychologically plausible.

8.4 Extensions and future work

We have already mentioned a number of possible extensions to the present work. For instance, it is natural to embed our model within a control algorithm, such as that used in the recent Spaun model [9]. We could then make use of our network for cognitive tasks and bring it into better contact with behavioral data. Additionally, we have acknowledged that it will be crucial to investigate how this neural representation might be learned from training data while retaining its desirable scaling properties.

There are number of avenues for improvement beyond these two. For instance, while we have considered only lexical encoding here, semantic pointers are flexible enough to allow many types of information to be encoded simultaneously [7]. To construct a representation more reminiscent of a full-fledged concept, we could add perceptual, motor, or dynamics information to each semantic pointer. For the visual modality, this could be accomplished by adding to each semantic pointer a term of the form $\mathbf{vision} \circledast \mathbf{visualData}$ where **visualData** is visual information about the concept, and **vision** is a marker that is analogous to the relation-type vectors we have used throughout this study. **visualData** could then be extracted from the semantic pointer by a modified version of our model, with **vision** as the query vector. The model would have to be modified to use an associative memory storing visual information instead of the lexical information we have used in the present study. One could imagine a number of instances of our model in different cortical areas, each with an associative memory storing the type of information relevant for that brain area. Further modality-specific processing could then be performed on the extracted information. A similar approach for a small scale vocabulary has been pursued in the Spaun model [9].

Chapter 9

Alternate encodings

9.1 Doing away with ID-vectors

Earlier we mentioned the possibility of freeing our method of ID-vectors. This would reduce the complexity of the method, since we would only have to keep maintain one vector, the semantic pointer, per WordNet synset. This would entail defining the semantic pointer for a synset directly in terms of the semantic pointers for related synsets. For example, *dog* would be encoded as:

$$\mathbf{dog_{sp}} = \mathbf{class} \circledast \mathbf{canine_{sp}} + \mathbf{member} \circledast \mathbf{pack_{sp}}$$

The associative memory will also have to be modified, because now $\mathbf{dog_{sp}} \circledast \overline{\mathbf{class}}$ will be a vector similar to $\mathbf{canine_{sp}}$ rather than $\mathbf{canine_{id}}$. Consequently, both the address vectors and the stored vectors are equal to the semantic pointers, and the associative memory becomes an auto-associative memory.

The most obvious problem with this approach, as we previously mentioned, is that it does not allow the encoding of semantic networks with directed cycles, because some of the semantic pointers will be defined in terms of one another. However, we can free WordNet of directed cycles by deleting a small number of relations, allowing us to at least test this encoding and see whether it works in principle. Deleting the directed cycles allows us to define an ordering of the synsets such that for any relation in the semantic network, the target synset of the relation occurs earlier than the source synset. For example, *canine* would occur earlier than *dog* since there is a relation whose target is *canine* and whose source is *dog*. In graph theory this is known as a topological ordering. To assign semantic pointers to synsets, we begin by assigning random unit vectors to synsets that are not the source of any relations (for example, the first synset in the ordering), and then assign semantic pointers according to the topological ordering. The result is that whenever we assign a semantic pointer to a synset, all the other synsets that it is related to have already been assigned semantic pointers, and the process is consequently well-defined.

9.1.1 The naive approach

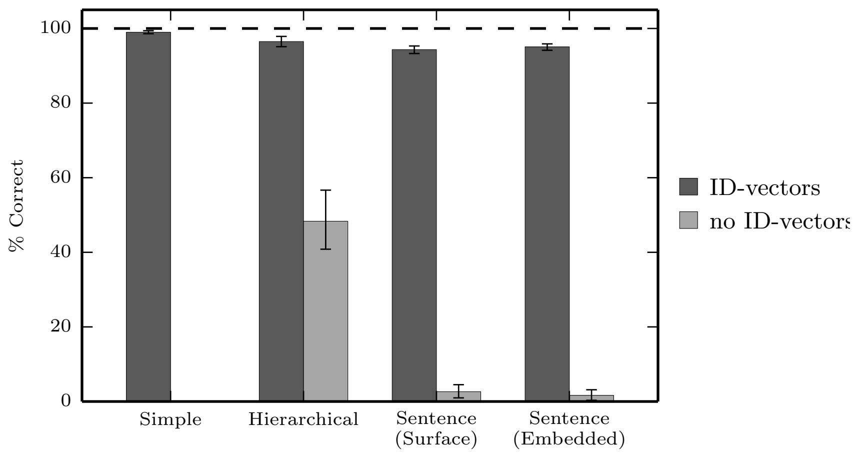


Figure 9.1: Performance on WordNet encoding without ID-vectors.

Fig. 9.1 shows the results of performing the experiments from Section 7 on this encoding. We note that these results, and in fact all results in this section are obtained from the abstract algorithm and not the neural implementation. As we can see, performance is much poorer than the performance obtained using the ID-vector encoding. The primary reason for this is that the semantic pointers are being used as the address vectors in the associative memory, but the semantic pointers also have a high-degree of similarity to one another. The result is that many of the pairs in the associative memory become active in response to any given input vector, and the output of the associative memory is a useless sum of many semantic pointers.

We can see the high mutual similarity of the semantic pointers in this new encoding by simply sampling a number of the semantic pointers that are created during this encoding, and taking their dot product. This is shown in Fig. 9.2. We can see that in the encoding we have used throughout the paper, the address vectors in the associative memory (the ID-vectors) have mutual similarity near 0. In contrast, in the new encoding where we use semantic pointers as the address vectors, the mutual similarity between the address vectors is often as high as 1.

The basic reason that the semantic pointers are similar to one another is that they are not chosen purely randomly as the ID-vectors are; rather, they encode relational structure, and semantic pointers that encode similar relational structure will themselves be similar (the “shallow semantics” of semantic pointers). However, particular aspects of this encoding make this effect worse than is strictly necessary. To see this, we expand out the semantic pointer for *dog*:

$$\begin{aligned}\mathbf{dog_{sp}} &= \mathbf{class} \circledast \mathbf{canine_{sp}} + \mathbf{member} \circledast \mathbf{pack_{sp}} \\ &= \mathbf{class} \circledast (\mathbf{class} \circledast \mathbf{carnivore_{sp}} + \mathbf{member} \circledast \mathbf{Canidae_{sp}}) + \mathbf{member} \circledast \mathbf{pack_{sp}} \\ &= \mathbf{class} \circledast \mathbf{class} \circledast \mathbf{carnivore_{sp}} + \mathbf{class} \circledast \mathbf{member} \circledast \mathbf{Canidae_{sp}} + \mathbf{member} \circledast \mathbf{pack_{sp}} \quad (9.1)\end{aligned}$$

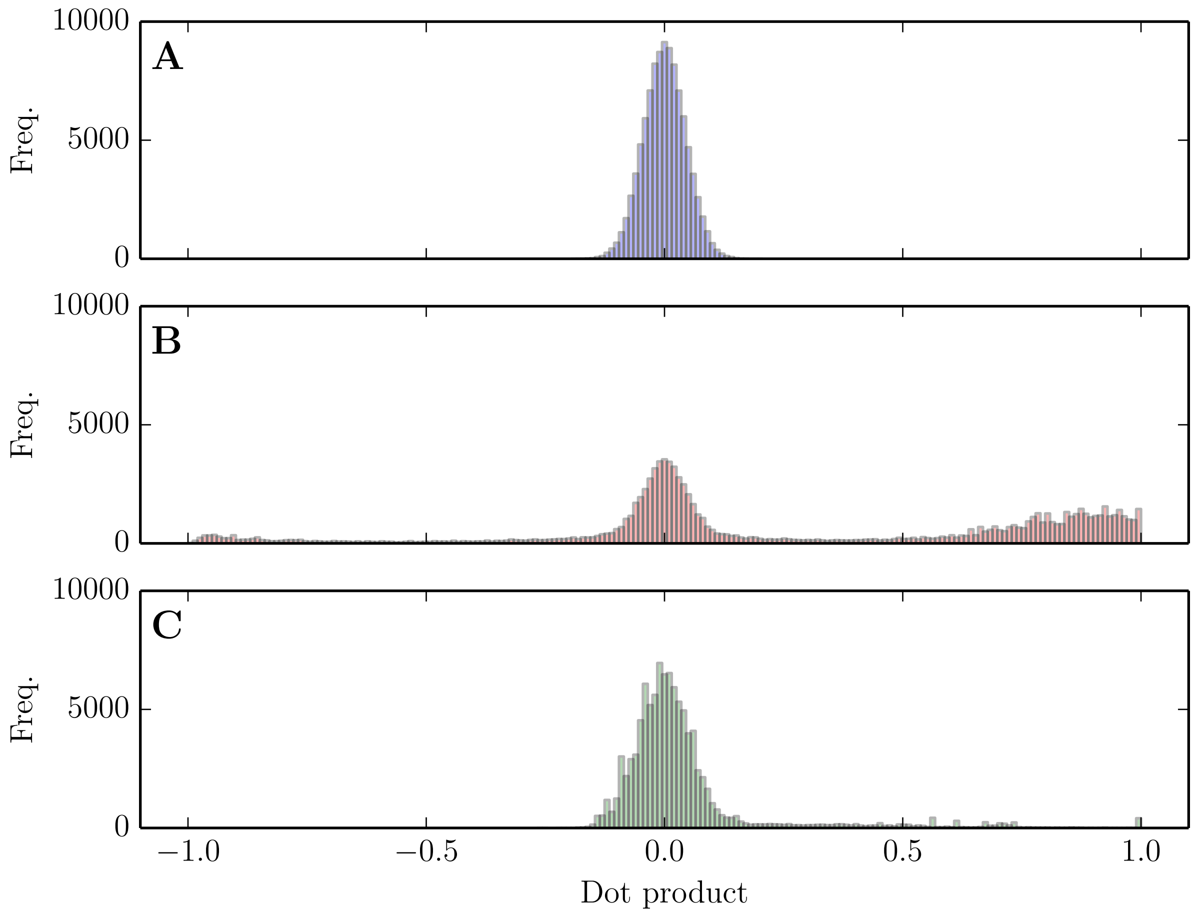


Figure 9.2: Histograms of dot products between address vectors of the associative memory under different encodings of WordNet. Data obtained by randomly sampling 100,000 pairs of distinct WordNet synsets (with replacement) and taking the dot product of their address vectors. A. Original encoding, where the address vectors are the ID-vectors. B. Encoding without ID-vectors, where the address vectors are the semantic pointers. C. Same as B, but with the relation-type vectors chosen to be unitary vectors.

Equation (9.1) can be expanded out much further, until we have defined the semantic pointer for *dog* in terms of the vectors assigned to leaf synsets. The full expansion is omitted for brevity. What this shows is that the distributive property of circular convolution implies that semantic pointers for non-leaf synsets will contain terms consisting of relation-type vectors convolved repeatedly with themselves. In fact the full expansion of $\mathbf{dog_{sp}}$ contains $\mathbf{class}^{13} \circledast \mathbf{entity}$, where $\mathbf{class}^{n}$ is understood to mean **class** convolved with itself *n* times. This is a problem because taking the circular convolution of a vector with itself repeatedly results in the magnitude of the vector growing rapidly. The effect is that other terms are largely lost when the semantic pointer is normalized. Thus, long chains consisting of a single relation-type are given significantly more weight in the semantic pointers. Consequently, even if two semantic pointers have quite distinct relational structures, if they share a long chain consisting of a single relation-type, they will end up being very similar.

9.1.2 Unitary relation-type vectors

Fortunately, this behavior has already been investigated in [40](p. 118-119) and there is a technique which addresses the problem. The proposed solution is to restrict the relation-types vectors to be vectors whose Fourier transformed components have a magnitude of one. Such vectors are called unitary, and we have seen them previously when we used them as the sentence-role vectors when we created semantic pointers encoding sentences in Section 7. One of the many unique properties of unitary vectors is that if $\mathbf{u}$ is unitary, then $\mathbf{u}^n$ has norm 1.0 for integers $n > 0$. When we create the encoding under this constraint and perform the same sampling of the address vectors (semantic pointers) as before, we see that the address vectors are, in general much less similar to one another. This is shown in the bottom plot of Fig. 9.2. We also rerun the experiments to determine whether this actually results in improved performance as we expect. The results are presented in Fig. 9.3, where we can see that performance has indeed increased significantly.

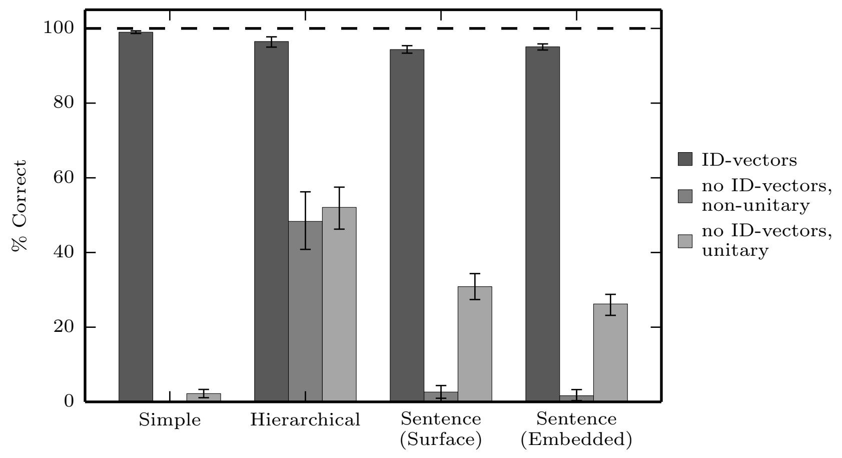


Figure 9.3: Performance on WordNet encoding without ID-vectors.

9.1.3 Adding noise

Restricting the relation-type vectors to be unitary has helped, though not as much as we might like; the results are still quite far from the results obtained through the ID-vector encoding. Looking again at the bottom plot of Fig. 9.2, its clear that even when we use unitary relation-type vectors, there are still some synsets that have very similar semantic pointers. We can reduce this similarity further by incorporating some of the strengths of the ID-vector approach. Specifically, we add random noise to the semantic pointer vectors. For instance,

$$\mathbf{dog_{sp}} = \mathbf{class} \circledast \mathbf{canine_{sp}} + \mathbf{member} \circledast \mathbf{pack_{sp}} + \mathbf{N} \tag{9.2}$$

where N is a random unit vector which is different for each synset. This should make the semantic pointers more dissimilar once they have been normalized, which should improve the performance of the associative memory. In fact we can add in arbitrary amounts of noise by adding in multiple noise vectors:

$$\mathbf{dog_{sp}} = \mathbf{class} \circledast \mathbf{canine_{sp}} + \mathbf{member} \circledast \mathbf{pack_{sp}} + \mathbf{N_1} + \cdots + \mathbf{N_n} \tag{9.3}$$

The downside is that adding this noise makes decompression itself noisier and therefore more prone to error. To find the optimal amount of noise to add, we can simply run our experiments with different amounts of noise added. Fig. 9.4 shows the results of this manipulation when using non-unitary relation-type vectors. We can see that it has helped somewhat, though is still nowhere near the performance when using ID-vectors. On the other hand, Fig. 9.5 shows that performing this same manipulation while using unitary relation-type vectors results in performance on par with that of the ID-vector encoding when $n = 5$ noise vectors are added to the semantic pointers.

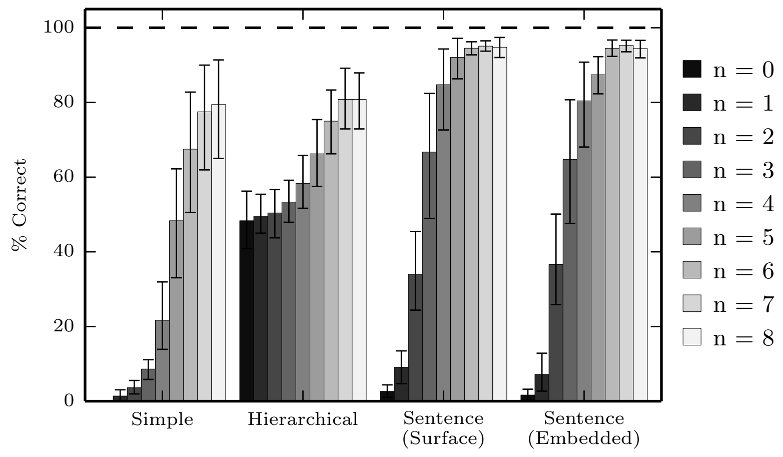


Figure 9.4: Encoding WordNet without ID-vectors, using non-unitary relations and noise added to the semantic pointers. n = 0 corresponds to the encoding with no noise added. Performance plateaus once n = 8 noise vectors have been added, and performance is never as good as the encoding that uses ID-vectors.

So it is possible at least in principle to encode a large structured knowledge base without resorting to ID-vectors. However, the constraint that graphs with directed cycles cannot, as of yet, be represented under this encoding is still a major mark against it. However, we cannot rule out that someone may come up with a systematic method for finding semantic pointers that can be written in terms of one another.

9.2 Including more relations

We have only included a subset of the relations that occur in WordNet. Specifically, we have included, *class*, *instance*, *member*, *part*, and *substance*, but have not included their inverses. Ex-

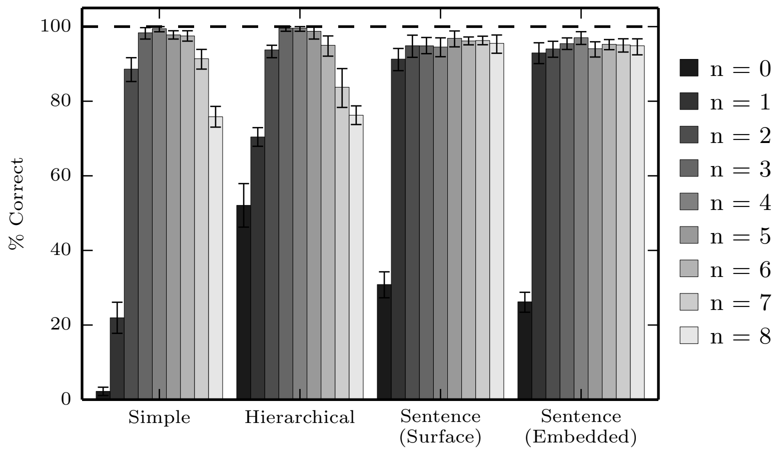


Figure 9.5: Encoding WordNet without ID-vectors, using unitary relations and noise added to the semantic pointers. n = 0 corresponds to the encoding with no noise added. Performance plateaus once n = 4 noise vectors have been added where it performs as well as the ID-vector encoding, and decreases thereafter, indicating that the semantic pointers contain too much noise for decompression to be reliable. Notably, the performance on the Sentence Extraction test never decreases, because noise is not added to the semantic pointers that encode sentences.

cluding these still allows complex inferences to take place, but including them would provide more flexibility. However, there are several aspects of the WordNet graph, modified to include those inverse relations, that are difficult for our technique.

The most obvious problem is that nodes with very high out-degree become much more common. For instance, *dog* will now be related to all possible dog breeds via the inverse of the *class* relation. When combined with the relations that we have already included in *dog*, this puts the out-degree of *dog* at 23. The consequence is that $\mathbf{dog_{sp}}$ will contain 23 terms. Extracting the constituents of such a vector in the straightforward way we have done so far would have extremely low fidelity. This is largely a result of the structure of WordNet, and may not be a problem in real human-knowledge bases. To solve this problem in the context of WordNet, we can cluster the synsets related to each high-degree synset and then create a new node for each cluster. For the case of *dog*, we could cluster the dog breeds by color, and end up with new nodes for *brown dog*, *white dog*, etc., resulting in a graph with more nodes, but fewer relations emanating from any given node.

9.3 Synsets with multiple relations of the same type

One mark against our approach, which we briefly noted above, is how it handles synsets that have multiple relations of the same type. For example, the synset *lion* is related to both *pride* and *panthera* via the *member* relation-type. The semantic pointer for *lion* :

$$\mathbf{lion_{sp}} = \mathbf{member} \circledast \mathbf{pride_{id}} + \mathbf{member} \circledast \mathbf{panthera_{id}} + \mathbf{class} \circledast \mathbf{bigcat_{id}}$$

Consequently, $\mathbf{lion_{sp}} \circledast \overline{\mathbf{member}}$ will be similar to both $\mathbf{pride_{id}}$ and $\mathbf{panthera_{id}}$. Then, when this vector is fed into the associative memory, the association populations for both *pride* and *panthera* will become active, and their outputs (the corresponding semantic pointers) will be summed in the dendrites of the output population. In short, the output of the network will be $\mathbf{pride_{sp}} + \mathbf{panthera_{sp}}$.

For the experiments we have performed in this study, this is acceptable behavior. For the Simple Extraction experiment, we take the dot product of the output vector with the semantic pointer we expect to get out, and if this is above a threshold of 0.7, we judge it as correct. If the output were $\mathbf{pride_{sp}} + \mathbf{panthera_{sp}}$, the dot product of this with $\mathbf{pride_{sp}}$ would likely be greater than 0.7. This highlights that fact that in the Simple Extraction test, we are testing for the *presence* of the vector we expect to get as output, not whether the output and the expected semantic pointer are identical. In the case of the Hierarchical Extraction experiment, the fact that the output is $\mathbf{pride_{sp}} + \mathbf{panthera_{sp}}$ is actually beneficial, as it effectively allows the network to simultaneously follow multiple paths through the WordNet graph. Finally, for the Sentence Experiment, the randomly constructed sentences never contain the same role twice, so this is not an issue.

However, in general one could see this behavior being a problem, and it would be desirable to always get a single semantic pointer. For this to happen, either one of the synsets needs to be chosen randomly, or the querier simply has to provide more information to disambiguate between the possible synsets that could be returned. Below we address outline several ways that the latter option could be implemented, leaving the former option for future work.

We first note that semantic pointers can be used to encode list-like structured representations. Instead of relation-type vectors, we use “position” vectors. For example, to encode the list [’lion’, ’tiger’, ’bear’], we could create the following semantic pointer:

$$\mathbf{list_{sp}} = \mathbf{zero} \circledast \mathbf{lion_{id}} + \mathbf{one} \circledast \mathbf{tiger_{id}} + \mathbf{two} \circledast \mathbf{bear_{id}}$$

We can use this structure to solve our problem as follows. In place of the semantic pointer creation process we have used up to this point, we instead convolve each relation-type vector with a semantic pointer encoding a list which stores the ID-vectors of all synsets that are related to the original synset by that relation-type. For example:

$$\mathbf{lion_{sp}} = \mathbf{member} \circledast (\mathbf{zero} \circledast \mathbf{pride_{id}} + \mathbf{one} \circledast \mathbf{panthera_{id}}) + \mathbf{class} \circledast (\mathbf{zero} \circledast \mathbf{bigcat_{id}})$$

The benefit of this approach is that if one knows the position of the item of interest, then one can use a compound query vector to extract exactly the desired semantic pointer. For example, to extract only $\mathbf{panthera_{sp}}$, one would use $\mathbf{member} \circledast \mathbf{one}$ as the query vector. If the querier does not have any specific index in mind, it can always use $\mathbf{member} \circledast \mathbf{zero}$, and the returned vector should be a semantic pointer corresponding to a single synset (if there are any synsets related by the *member* relation, and the 0 vector otherwise). The disadvantage is that if the querier *does* have a specific item in mind, but its position is not known, then a more complicated process will be required in order to extract the desired item, such as iterating over all the position vectors starting from **zero**.

There are yet more possibilities. Recall that *lion* is related to both *panthera* and *pride* via the *member* relation. However, the sense of the word "member" in each of these relations is slightly different. One refers to the genus to which the species lion belongs, and the other refers to the name for a collection of lions. The idea, then, is to simply sidestep the issue by using a richer and more finely grained set of relation-types across the whole knowledge base, making synsets with multiple relations of the same type non-existent.

However, it is unlikely that this will work in all cases. Consider the case of *dog* when reciprocal relations are included as we discussed in the previous section. Recall that *dog* is related to *poodle*, *corgi*, *dalmatian*, and 15 other dog breeds through the inverse *class* relation. Here it is not clear that the sense of the relation is different between each of these 18 uses, as was the case in the *lion* example above; therefore we require some other means of singling one of them out. One possibility is to have the relation vector include, or even be completely composed of, perceptual information relevant to the target of the relation. For instance, we could encode dog as:

$$\mathbf{dog_{sp}} = \mathbf{dalmatian_{percept}} \circledast \mathbf{dalmatian_{id}} + \mathbf{poodle_{percept}} \circledast \mathbf{poodle_{id}} + \mathbf{spitz_{percept}} \circledast \mathbf{spitz_{id}} + \dots$$

where the $\mathbf{x_{percept}}$ vectors consist of perceptual information combined with the relevant relation-type vector. Then $\mathbf{dog_{sp}} \circledast \overline{\mathbf{dalmatian_{percept}}}$ would be a vector similar to only $\mathbf{dalmatian_{id}}$, and the output of the associative memory would be $\mathbf{dalmatian_{sp}}$.

Things get especially interesting if we consider that perhaps the querier does not have a complete idea of the item it would like to extract. In other words, it uses a noisy version of the query vector. This may even be the norm, because it is likely that the perceptual information at query time, (perhaps a particular dalmatian), would differ somewhat from the stored perceptual information (perhaps an amalgamation or average of many different dalmatians).

A scheme similar to this has been explored by Hunsberger et al. in [21]. The authors of that paper created a semantic pointer from a number of labeled images, and later used it for classification. To do this, they first compressed the images using a hierarchical vision model (specifically, a deep autoencoder). Then each compressed image was circularly convolved with a vector marking the label of the image. Finally, all vectors of this form were combined into a single semantic pointer through addition. For instance:

$$\begin{aligned}
\mathbf{classifier_{sp}} &= \mathbf{label}_1 \circledast \mathbf{img}_{1,1} + \mathbf{label}_1 \circledast \mathbf{img}_{1,2} + \mathbf{label}_1 \circledast \mathbf{img}_{1,3} + \dots \\
&\quad + \mathbf{label}_2 \circledast \mathbf{img}_{2,1} + \mathbf{label}_2 \circledast \mathbf{img}_{2,2} + \mathbf{label}_2 \circledast \mathbf{img}_{2,3} + \dots \\
&\quad + \mathbf{label}_3 \circledast \mathbf{img}_{3,1} + \mathbf{label}_3 \circledast \mathbf{img}_{3,2} + \mathbf{label}_3 \circledast \mathbf{img}_{3,3} + \dots \\
&= \mathbf{label}_1 \circledast (\mathbf{img}_{1,1} + \mathbf{img}_{1,2} + \mathbf{img}_{1,3} + \dots) \\
&\quad + \mathbf{label}_2 \circledast (\mathbf{img}_{2,1} + \mathbf{img}_{2,2} + \mathbf{img}_{2,3} + \dots) \\
&\quad + \mathbf{label}_3 \circledast (\mathbf{img}_{3,1} + \mathbf{img}_{3,2} + \mathbf{img}_{3,3} + \dots) \\
&\quad + \dots
\end{aligned}$$

where $\mathbf{label}_1$, $\mathbf{label}_2, \dots$ are the randomly chosen label vectors, and $\mathbf{img}_{m,n}$ is the compressed version of the *n*th image belonging to the *m*th label. To classify a new image, they run the image through the compression algorithm yielding $\mathbf{img_{new}}$, and compute $\mathbf{classifier_{sp}} \circledast \overline{\mathbf{img_{new}}}$. The label of the new image is estimated to be the label whose corresponding vector has the highest dot product with this vector. In other words:

$$\text{estimated label} = \arg\max_c [(\mathbf{classification_{sp}} \circledast \overline{\mathbf{img_{new}}}) \cdot \mathbf{label}_c]$$

The model matched well with human data on several different tasks.

This setup is similar to our proposed solution, except for the notable difference that to get their label, they take an argmax over dot products whereas, we would run $\mathbf{classification_{sp}} \circledast \overline{\mathbf{img_{new}}}$ through an associative memory. Using an associative memory places more stringent demands on the statistics of the input vectors than taking the argmax, but it is not clear that taking the argmax in neurons will scale as well as an associative memory. Whether the decompressed vectors end up having statistics that allow an associative memory to function well is unclear, and should be the subject of further study.

We note that, since WordNet contains only lexical information, this solution is not available to us. However, there exist newer databases that have perceptual information added in; for instance, the creators of ImageNet have taken WordNet and annotated it with an average of 500 images per concrete noun [4]. Moreover, this technique is unlikely to work at all for abstract nouns, for which perceptual information is neither attainable nor useful. It seems more likely that a combination for these techniques would be optimal, with perceptually-constructed relation vectors used those synsets for which such information is available, and a richer set of relation-type vectors for the more abstract concepts.

Chapter 10

Learning associative memories

All the weights in WordNet model were "hand-coded". That is, weight matrices computing the desired functions were derived offline using the Neural Engineering Framework. This is a contrast to standard connectionist research, where a neural network is typically instantiated with random weights, and those weights are subsequently adjusted throughout the course of a training process according to some kind of learning rule, often Hebbian learning in the case of unsupervised learning or backpropogation in the case of supervised learning.

We have been satisfied with these "hand-coded" weights until now because merely *representing* large-scale structured knowledge bases, without worrying about how that representation is obtained, was still an open question. However, if brains do encode semantic knowledge in a manner that looks anything like the one we have presented, then there must be a way to derive these connection weights through a biologically plausible learning process in response to experience in the world. Thus, providing a biologically plausible account of how the weights were attained would lend additional support to our to our technique. Here we investigate how this could be achieved. Other work has demonstrated how the involution/circular convolution circuit can be learned in a biologically plausible manner [49]. Consequently, in the current chapter we restrict our scope to learning an associative memory.

10.1 Training regime

There is no clear-cut answer to the question of what form of data the network should be trained on. Here we make a plausible assumptions, but acknowledge that other regimes are possible. Essentially, we assume that there is some pre-defined list of pairs to be stored in the associative memory, $\langle \xi_k, \eta_k \rangle$ for $k \in 1 \ldots N$, and that each of these pairs is presented to the memory one at a time, for 1 second each. As before, we call the ξ_k vectors the "address" vectors and the η_k vectors the "stored" vectors. In our simulations, we will assume that the list of vector pairs encodes a semantic network. That is, we assume that the address vectors are ID-vectors, randomly chosen from the D-dimensional unit hypersphere, and the stored vectors are D-dimensional semantic pointers.

Once training is complete, we expect to be able to provide a noisy version of any address vector ξ_k as input, and have the network output the corresponding stored vector η_k.

10.2 Network architecture

We start by defining a basic network architecture, to which we will subsequently add parts as we show that they are necessary. This basic architecture consists of 3 neural populations connected in a feedforward manner. Both the input and output populations are standard NEF populations which represent D-dimensional vectors.

Essentially, we want to start with this basic network architecture, and, through the training process, morph it into the same network that would result if we derived the connection weights offline using the NEF associative memory construction technique from Section 6.4. Recall that in the hand-coded associative memory, every pair of vectors is assigned a small neural population. The collection of those small populations corresponds to the middle population in our current network. Consequently, we call this middle population the association population. The difference between the association population here and the collection of populations in the "hand-coded" network is that, before training, none of the neurons in our association population will have been assigned pairs of vectors to associate; the intent is that said assignment happens as part of the training process. For convenience, we call the connections weights between the input population and the association population the "input weights", and the weights between the association population and the output population the "output weights"

We make a number of assumptions on the form of input and output weight matrices to facilitate learning. First, we assume that the input weight matrix is derived, in the usual NEF fashion, such that it computes a communication channel; that is, it is the product of the encoding vectors of the association population (which are chosen uniformly as random from the unit hypersphere) and the decoding vectors of the input population for the identity function. The second assumption is that the output weights are all initialized to be 0. The utility of this will become clear in Section 10.4 when we talk about modifying the output weights as part of the training process. These initial assumptions are depicted in the untrained network in Fig. 10.1, which gives a schematic view of the training process. The final assumption is that the neurons in the association population have high firing thresholds. The purpose of this is to make it so that during both training and testing, only a small number of association neurons will become active in response to any given vector presented as input. This is also known as a sparse representation. Assuming the threshold of the association neurons and the dimensionality of the vectors is fixed, we can choose the size of the association population such that when this address vector is presented as input during training, on average a small number of the neurons in the association population are activated. Steps for deriving the appropriate size of the association population are given in Appendix B. The relationship between the firing threshold, the number of neurons in the association population, and the number of active neurons per input vector is visualized in Fig. 10.3.

10.3 General concept

During training, upon the presentation of a pair of vectors $\langle \xi_k, \eta_k \rangle$ to be stored, we will supply the address vector of that pair ξ_k as input to the network. If the size of the association population has been chosen appropriately with respect to the neural threshold and the dimensionality of the vectors, a small number of neurons should be active. These active neurons will be thought of as "assigned" to the current training pair, in the same way that a small association population was assigned to each pair in the hand-coded associative memory. The goal of the training, then, is to change both the input and output weights on these active neurons so that they perform similarly to the populations in the hand-coded associative memory.

There are two components involved in making these assigned neurons perform the association functionality for that pair. The first is to make it so that whenever those neurons are active, the stored vector of that pair, η_k, gets sent to the output population. The second is to make it so that this population of neurons is only active in response to noisy versions of the address vector, ξ_k, and completely inactive in response to noisy versions of other address vectors in the memory. The first component will be achieved by applying the Prescribed Error Sensitivity learning rule to the output weight matrix during training, which has the effect of making the decoding vectors of the active neurons equal to η_k. The second component is achieved by applying the Oja learning rule to the input weight matrix during training, which has the effect of making the active neurons fire preferentially for ξ_k. The network augmented with the components required for learning to occur is shown in Fig. 10.2. We now outline the specifics of these two learning rules.

10.4 Prescribed Error Sensitivity: Storing vectors in connection weights

The Prescribed Error Sensitivity (PES) rule is an error-modulated learning rule that can be used to modify the decoding vectors of a population of neurons online [30]. Effectively, the rule works to modify a connection weight matrix such that some error signal is minimized. Fig. 10.4 show the general form of a neural circuit employing the PES rule, as well as how the populations in that general circuit map on to the populations in our basic learning architecture.

The PES learning rule can be most succinctly expressed in terms of a decoding vector update equation:

$$\Delta d_i = \kappa \mathbf{E} a_i \tag{10.1}$$

where Δd_i is the additive change to the decoding vector of neuron i in the pre-synaptic population, κ is a learning rate constant, $\mathbf{E}$ is the error signal decoded from the error population, and a_i is the activity of the ith neuron. This equation can be mapped to an equation that operates at the level of connection weights and uses only values that are locally available at each synapse, making it biologically plausible. For our present purposes, it is more convenient to work with the equation

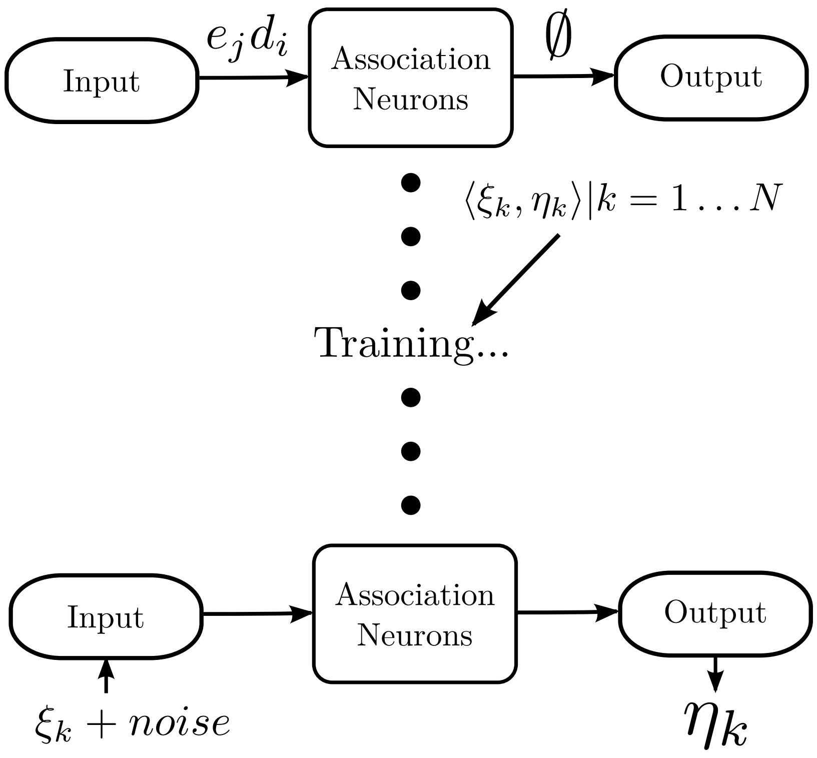


Figure 10.1: Schematic diagram of the training process. Start with a 3 layer network of spiking neurons, where the input weights are initialized in the usual NEF style, the output weights are set to 0, and the middle layer neurons have relatively high thresholds. Training proceeds by presenting each pair to be associated exactly once, for one second. Once the network has been trained, we should be able to give a noisy version of any address vector ξ_k as input, and get the corresponding stored vector η_k as output.

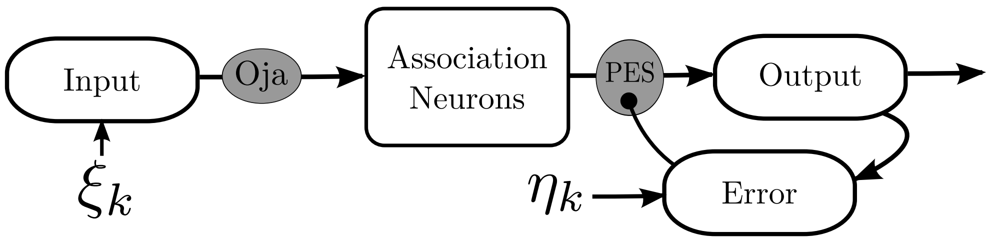


Figure 10.2: Schematic diagram of the network during training, showing the presentation of a pair of training vectors $\langle \xi_k, \eta_k \rangle$. The learning rules that modify the connection weights of active neurons in response to the training vectors are depicted in gray.

at the level of decoding vectors. Intuitively, this learning rule works by pushing the decoding vectors of the pre-synaptic neurons towards the current desired output vector (which is part of **E**) in proportion to how active the neuron is (hence the inclusion of a_i).

During training on a pair of vectors $\langle \xi_k, \eta_k \rangle$, ξ_k will be fed into the population, causing a small number of neurons in the association population to become active. We use the PES rule to minimize the value represented in the Error population, which is just the difference between the value represented in the Output population and η_k. This should have the effect of moving the

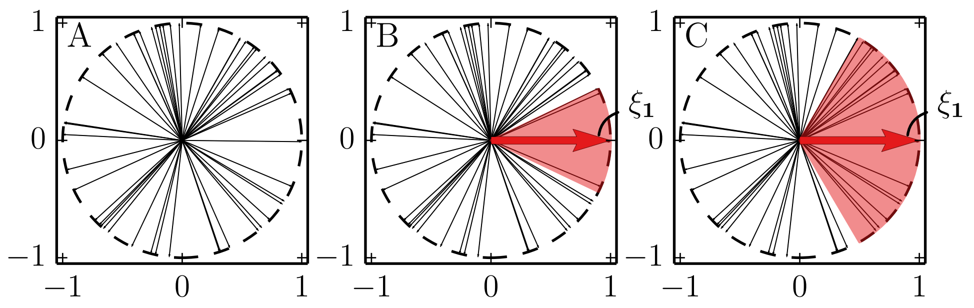


Figure 10.3: A. Visualization of the encoding vectors of an association population containing 50 neurons, before training. Here we use an association population that will store 2D vectors because 2D space can be easily visualized. However, we will typically want to store much higher-dimensional vectors B. Presentation of an address vector, in this case $\xi_1 = (1.0, 0.0)$, to the network. Neurons have firing threshold of 0.9. All encoding vectors inside the red wedge have a dot product with ξ_1 that exceeds the firing threshold, and consequently the neurons to which they belong will be active. C. Same as B, but the neurons have threshold 0.5. With lower neural thresholds, more neurons are active in response to any given address vector. In contrast, if we kept the threshold fixed but added more neurons the number of encoding vectors in each wedge, and thus the number of active neurons, would increase.

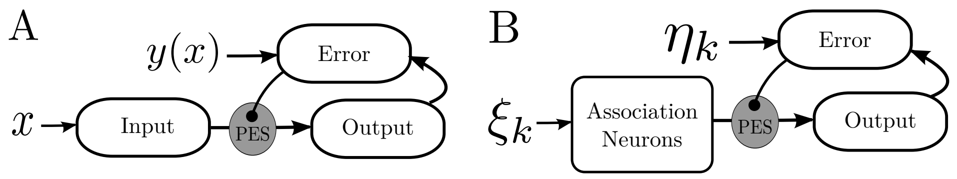


Figure 10.4: A. The general form of a circuit using the PES rule. The PES rule works to drive the output of the Output population closer to y, the desired output, by modifying the decoding vectors of the Input population (which are part of the weight matrix between the Input and Output populations). B. Mapping the populations in A on to our associative memory learning network. The desired output is η_k.

decoding vectors of the active neurons towards η_k. The idea, then, is that during testing, a noisy version of ξ_k will activate roughly the same association neurons as ξ_k itself did, which will cause η_k to be represented in the Output population. We can thus think of these neurons that have had their decoding vectors altered as having been assigned to the pair $\langle \xi_k, \eta_k \rangle$. We visualize this assignment in Fig. 10.5.

However, it is inevitable that during the presentation of a testing vector like $\xi_k + noise$, some neurons will become active that were *not* assigned to $\langle \xi_k, \eta_k \rangle$. If those neurons have not been assigned to any pair at all, it will not effect the output because the decoding vectors have been initialized to 0. This is demonstrated in Fig. 10.6. On the other hand, it will often happen that $\xi_k + noise$ activates some neurons that have been assigned to other vector pairs (i.e. to $\langle \xi_j, \eta_j \rangle$ for $j \neq k$). The decoding vectors of those neurons will have been set to η_j by the PES rule, and consequently the activation of those neurons will introduced error into the result. This phenomenon is visualized in Fig. 10.7.

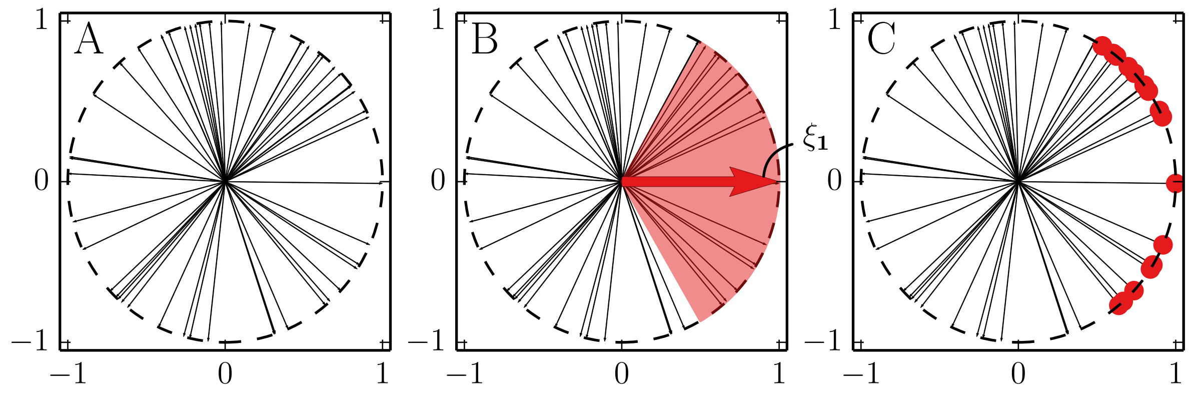


Figure 10.5: Visualization of the effects of applying the PES learning rule to the output weights while training on a vector pair $\langle \xi_1, \eta_1 \rangle$. A. The encoding vectors of the association population containing 50 neurons. B. Showing which neurons are active while ξ_1 is given as input and the threshold of the association neurons is 0.5. The PES rule takes effect during the presentation, and all active neurons have their decoding vectors modified to η_1. C. The neurons that were active have been marked by a red circle. We can think of these neurons as having been assigned to the pair $\langle \xi_1, \eta_1 \rangle$.

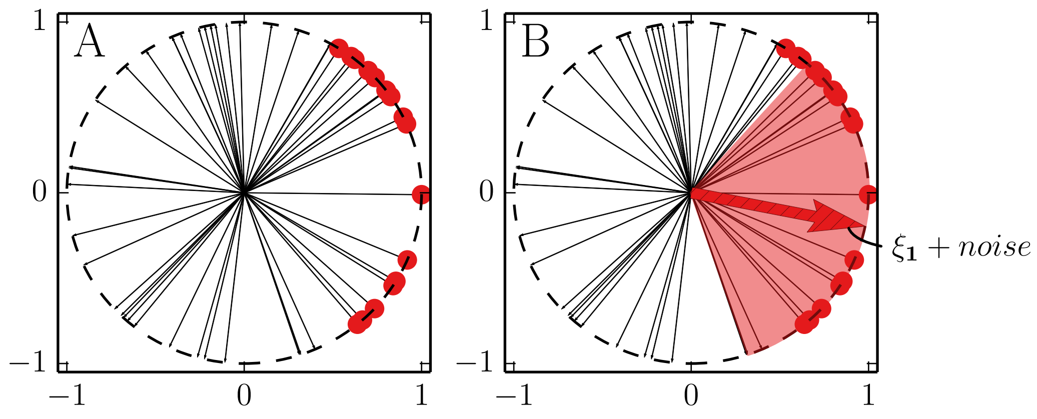


Figure 10.6: Demonstrating why the output weights are required to be 0 initially. A. State of association neurons after training on the pair $\langle \xi_1, \eta_1 \rangle$. B. Presentation of a testing vector (striped) derived from η_1. We can see that in the south south-east direction, there are several neurons that are active but have not been assigned to any vector pair. If output weights were not initially 0, then these unassigned neurons would affect the output and drive it away from η_1.

10.5 The Oja rule: Increasing neural selectivity

We solve this issue by augmenting our training process. We add an additional learning rule, this time on the input connection weights, whose purpose is to move the encoding vectors of active neurons towards the current input vector. For example, while the network is training on the pair $\langle \xi_k, \eta_k \rangle$, we move the encoding vectors of all active neurons towards ξ_k. The beneficial effects of this process are demonstrated in Fig. 10.8.

We add this to our model by using the well-known Oja learning rule [33]. This learning rule

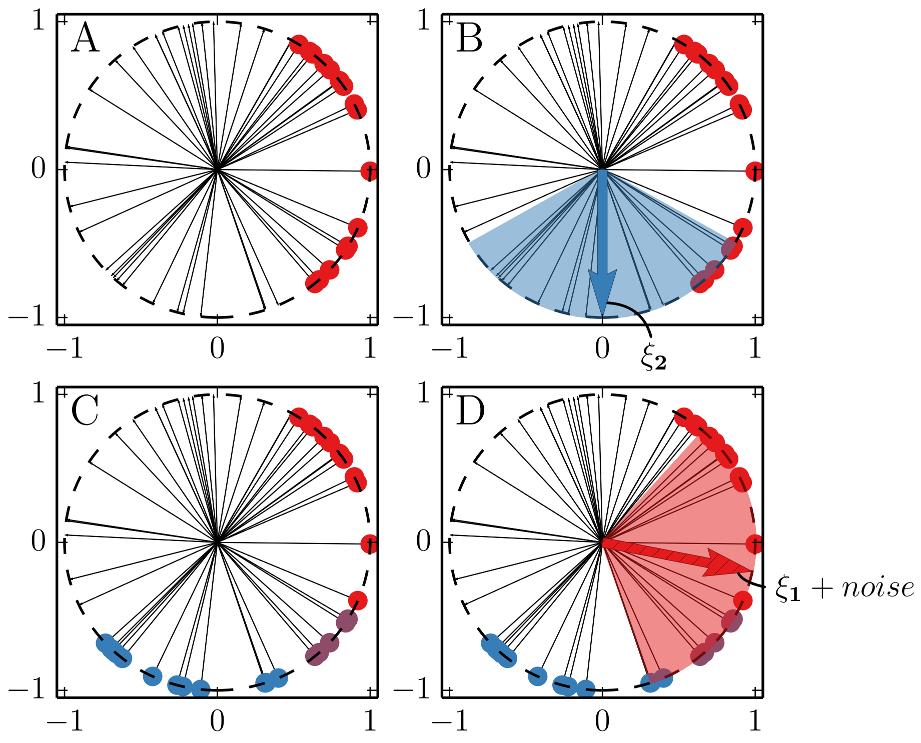


Figure 10.7: Demonstrating the need to move encoding vectors during training. A. State of association neurons after training on the pair $\langle \xi_1, \eta_1 \rangle$. B. Training on a second pair $\langle \xi_2, \eta_2 \rangle$. C. State of association neurons after training on both pairs. Purple neurons in the south-east direction have had their decoding vectors modified during the training of both pairs, and can therefore be thought of as having been assigned to both pairs; their decoding vectors will be a mixture of η_1 and η_2. D. Presentation of a testing vector derived from ξ_1. The testing vector can be seen to activate several purple and several blue neurons, which will incorrectly pull the output away from η_1.

operates directly at the level of connections weights, rather than at the level of encoding or decoding vectors as was the case with the PES rule. It is defined by the following weight update equation:

$$\Delta\omega_{ij} = \gamma b_j (a_i - b_j \omega_{ij}) \tag{10.2}$$

where ω_{ij} is the weight between the ith pre-synaptic neuron and the jth post-synaptic neuron, b_j is the activity of the jth post-synaptic neuron, and a_i is the activity of the ith pre-synaptic neuron. The Oja learning rule does not exactly move the encoding vectors towards the input vector; in fact, it operates completely independently of encoding vectors. Moreover, once the Oja learning rule has taken effect, it is generally not possible to decompose the connection weight matrix in terms of encoding vectors and decoding vectors. However, it does have the effect of making active neurons in the association population more selective for the address vector currently being presented. That is, it makes these neurons more likely to respond to input vectors that are similar to the address vector, and less likely to respond to input vectors that are dissimilar to the address vector. This is similar to what would be achieved by directly moving the encoding vectors. We

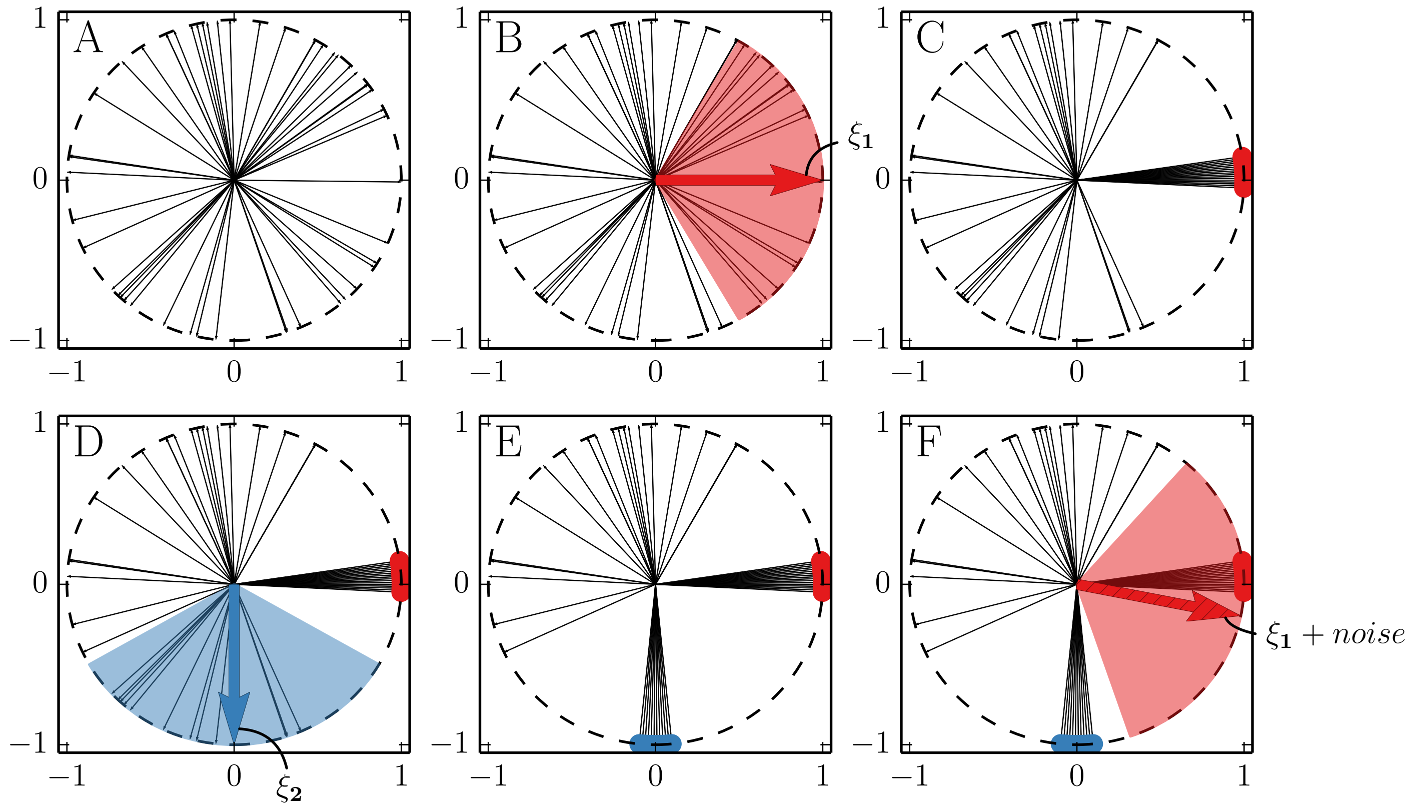


Figure 10.8: Demonstrating the benefits of moving encoding vectors during training. A-E. Similar training regime as before, but encoding vectors of active neurons are moved towards address vectors during training. F. When the testing vector is presented, it activates only neurons that have been assigned to the pair $\langle \xi_1, \eta_1 \rangle$, which should result in η_1 being the output.

use the Oja rule because it is a local learning rule, meaning all of the quantities on the right-hand side of Equation (10.2) could plausibly be available at the synapse, which is where the weight modification occurs. In Section 10.7 we discuss the Voja learning rule [54] which *does* work in terms of encoding vectors, directly moving them towards the vector currently represented in the pre-synaptic population. This simplifies analysis significantly, and generally results in better performance. However, more work needs to be done to fully establish the biological plausibility of the rule.

10.6 Simulations

To test the performance of this network, we run some simulations on it, first training it on predefined list of vectors, and then testing it with noisy versions of the address vectors.

We are primarily interested in using this associative memory for traversing semantic networks, so we use lists of vectors that encode semantic networks. We begin by randomly constructing a number of directed graphs with labeled edges and a pre-specified number of nodes, and then encode that graph in vectors using the same technique that we used to encode WordNet. We then initialize a spiking neural network with the initial conditions we specified in Section 10.2, and present the pairs of vectors in the encoding of the graph to the network one at a time for 1 second each.

During testing, we turn both the PES and Oja learning rules off. We test the performance of the network by randomly selecting a node in the graph, randomly selecting one of its edges, and computing $\mathbf{S_{sp}} \circledast \overline{\mathbf{R}}$, and feeding it to into the associative memory. Here $\mathbf{S_{sp}}$ is the semantic pointer for the chosen node in the graph, and $\mathbf{R}$ is the vector corresponding to the label of the chosen edge. As a measure of performance, we compare the similarity of $\mathbf{S_{sp}} \circledast \overline{\mathbf{R}}$ to the ID-vector it is supposed to be similar to, to the similarity between the output of the associative memory and the correct semantic pointer. The results of this experiment are shown in Fig. 10.9. Here we have used only 64 dimensional vectors; the significant computational complexity of the Oja rule has prevented us from simulating networks storing vectors with higher-dimensionalities. We can see that the performance of the learned memories is far from perfect. This is largely due to that fact that applying the Oja rule makes it impossible to analyze the input weight matrix in terms of encoders and decoders, which complicates analysis. In the next section we discuss the primary future direction of this project, the Voja rule, which, when used in place of the Oja rule, results in memories that are significantly more accurate and can be simulated much more rapidly.

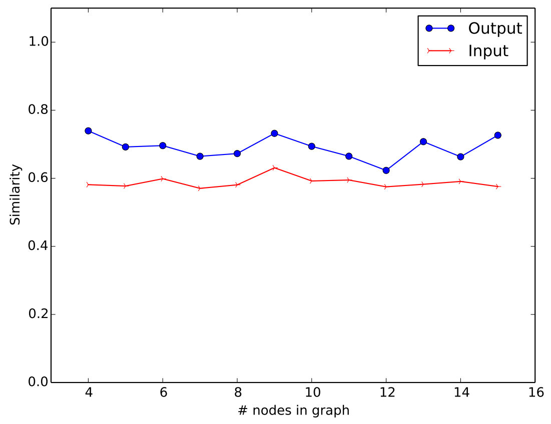


Figure 10.9: Simulation performance with 64 dimensional vectors, varying the number of pairs stored in the memory. Red line is the mean similarity of the input vectors to their corresponding ID-vectors, and blue line is the mean similarity of the output of the trained associative memory to correct semantic pointer. Clearly the associative memory is having some positive effect, but not as much as hoped. Perfect performance would have the blue line at 1.0. The red and blue used here have no relation to the red and blue used in the other plots in this section.

10.7 Future direction: The Voja learning rule

The primary difficulty with using the Oja learning rule is that once the rule has taken effect, it is, in general, no longer possible to decompose the connection weight matrix in terms of decoding vectors and encoding vectors. This complicates the analysis significantly, and results in associative memories with sub-optimal performance. In response to this, a fellow member of the Computational Neuroscience Research Group, Aaron Voelker, has created the Voja (Vector Oja) learning rule. This rule, inspired by the Oja rule, directly moves encoding vectors towards the vector currently being represented by the upstream, or pre-synaptic, population. Formally, the learning rule has the form:

$$\Delta \mathbf{e}_j = \gamma a_j (\mathbf{x} - \mathbf{e}_j) \tag{10.3}$$

where γ is a learning rate constant, a_j is the activity of the jth post-synaptic neuron (which will be an association neuron in our use-case), $\mathbf{e}_j$ is the encoding vector of the jth post-synaptic neuron, and $\mathbf{x}$ is the vector currently represented in the pre-synaptic population (the input population in our setup). Essentially, this is doing exactly what we were trying, somewhat unsuccessfully, to do with the Oja rule: move the encoding vectors of active association neurons towards the current input vector. Good large-scale associative memory performance has been obtained using this learning rule [54]. Because Voja operates on encoding vectors rather than on weights, it is also significantly more efficient to simulate than the Oja rule, permitting the learning of much larger associative memories.

One slight tradeoff is in the biological plausibility of the rule. It is generally assumed that in the brain, changes to connection weight matrices happen locally at the individual synapses, and, consequently, that changes to connection weights must be in terms of quantities that could plausibly be available at a synapse. For instance, recall that in the Oja rule, the weight update equation was specified in terms of the current weight, the activities of the pre- and post-synaptic neurons, and a learning rate constant, all of which are quantities that are generally assumed to be available at the synapse.

The Voja rule is specified in terms of the encoding vectors, but since encoding vectors do not have a simple physical correlate in the brain (they are part of the connection weight matrix), changes to encoding vectors have to be mapped to changes at the level of individual synapses. In other words, any learning rule that changes encoding vectors has to do so by modifying connection weights. This is similar to the PES learning rule, which modifies decoding vectors, but can be mapped to an equation that operates directly on connection weights. Calculating this mapping for the Voja rule, we have a connection weight update equation that takes the form:

$$\begin{aligned} \Delta \omega_{ij} &= \gamma a_j (\mathbf{x} - \mathbf{e}_j) \mathbf{d}_i \\ &= \gamma a_j (\mathbf{x}\mathbf{d}_i - \mathbf{e}_j \mathbf{d}_i) \end{aligned} \tag{10.4}$$

The right-most term in Equation (10.4) is proportional to the weight itself, and is thus likely to be at a synapse. On the other hand, the left most term, $\mathbf{x}\mathbf{d}_i$, or the dot product between the decoding vector of the *i*th pre-synaptic neuron and the vector represented in the pre-synaptic population, is not obviously available at the synapse. Thus further investigations will be required to establish biological mechanisms by which the Voja rule could operate. However, given the success of the Voja rule and its ability to create powerful, fast, efficient associative memories, it will undoubtedly be worth the effort.

Chapter 11

Conclusion

We have provided empirical results demonstrating what we believe to be the first implementation of a human-scale structured lexicon in a biologically plausible spiking neural network. We have argued that this significant improvement in scaling over previously available approaches is a result of employing the representational resources provided by the HRR vector algebra. We hope that by providing a specific, large-scale, functioning model we will encourage theoretical disagreements about structured representation to be replaced by implementations that can be quantitatively compared. In short, we believe that it will advance the field to expect that proposals regarding neural implementation of symbolic processing be implemented at scale. We have also provided a basic framework for how one of the central components of this model, namely its associative memory, can be learned online from training data in a biologically plausible manner.

Appendix A

WordNet model details

This appendix provides additional details related to the construction of the neural model which traverses the WordNet graph.

Finding decoding vectors. To find decoding vectors that decode a function f from the activity of neural population (denoted $\mathbf{d}_i^f$ where i indexes the neurons in the population), we said we had to minimize the expression:

$$\frac{1}{2}\int (f(\mathbf{x}) - \sum_i a_i(\mathbf{x})\mathbf{d}_i^f)^2 d\mathbf{x} \tag{A.1}$$

We minimize this numerically, using a finite number of evaluation points (values of $\mathbf{x}$) in some region of the represented space that we want our decoding vectors to perform well on. Let L denote the number of evaluation points, let M denote the dimensionality of the range of the function f, and let N denote the number of neurons in our population. We now define matrices that will aid us in the optimization. Let $\mathbf{D}$ denote the N x M matrix whose rows are the decoding vectors. Let $\mathbf{A}$ denote the L x N matrix whose rows are the activities of the neurons at a given evaluation point. Let $f(\mathbf{X})$ denote the L x M matrix whose rows are the values of the function f at different evaluation points. The jth row of $\mathbf{AD}$ is equal to the transpose of $\widehat{f(\mathbf{x}_j)} = \sum_i a_i(\mathbf{x}_j)\mathbf{d}_i^f$ where $\mathbf{x}_j$ is the jth evaluation point. Minimizing Equation (A.1) is then equivalent to solving for $\mathbf{D}$ in the following equation:

$$\begin{aligned} f(\mathbf{X}) &\approx \mathbf{AD} \\ \mathbf{A}^T f(\mathbf{X}) &\approx \mathbf{A}^T\mathbf{AD} \\ (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T f(\mathbf{X}) &\approx \mathbf{D} \end{aligned} \tag{A.2}$$

Since some neurons in the population are likely to have similar tuning curves, the matrix $\mathbf{A}^T\mathbf{A}$ is unlikely to be invertible. Thus, we typically take the *Moore-Penrose pseudoinverse* of $\mathbf{A}^T\mathbf{A}$ using Singular Value Decomposition (SVD), which is guaranteed to provide the least-squares optimal solution to Equation (A.2).

Sub-populations. This procedure for solving for the decoding vectors can be computationally intractable. Consider that in our neural model, one of the populations contains 51,400 neurons. The matrix $\mathbf{A}^T\mathbf{A}$ for that population would have dimensions 51,400 x 51,400. Taking the SVD of a matrix this large is not feasible.

Instead, we can consider these populations to be made up of many sub-populations, each of which represents a small subset of the dimensions of the overall population's represented space. The representational properties of the collection of sub-populations is very similar to that of a single large population. However, the computational properties are different [8]. The light gray populations from the Fig. 6.4 are split into 1-dimensional populations, whereas the dark gray population is split into 2-dimensional populations, each representing 1 dimension from each of the two Fourier transformed input vectors and computing their product (i.e. the ability to performed the required element-wise multiplication is retained).

This implementation allows for much more efficient computation of decoding vectors. For example, each of the light gray populations, which represent 512-dimensional vectors, is taken to be composed of 512 1-dimensional sub-populations of 50 neurons each instead of a single population of 25,600 neurons. As a result, SVD on a 25,600 x 25,600 matrix is replaced by 512 SVD's on 50 x 50 matrices which is computationally tractable. The only consequence of this is sparsification of the connection weight matrices; the same number of neurons are used in both cases.

Single Neuron Model and Parameters. All neurons are modeled as point-processes, and employ the leaky integrate-and-fire (LIF) neuron model. The sub-threshold behavior of the *i*th LIF neuron in a neural population is governed by the differential equation:

$$\frac{dV_i}{dt} = \frac{-1}{\tau_{RC}}(V_i - J_i(\mathbf{e_i x})) \tag{A.3}$$

The parameter τ_{RC} is a time constant governing the sub-threshold dynamics of the neuron. When the voltage V_i exceeds a voltage threshold $V_{th} = 1$, a spike is emitted from the neuron, the voltage is reset to zero, and a refractory period begins during which the voltage is fixed. The length of the refractory period is given by a constant τ_{ref}.

In Equation (A.3), $J_i(\mathbf{e_i x})$ is the input current of the *i*th neuron, and is given by $J_i(\mathbf{e_i x}) = \alpha_i \mathbf{e_i x} + J_i^{bias}$. The quantity $\mathbf{e_i x}$ is the dot product between the input vector $\mathbf{x}$ and the neuron's encoding vector $\mathbf{e_i}$. The parameters α_i and J_i^{bias} are uniquely determined by the neuron's range, maximum firing rate and firing threshold. The range specifies the interval of values of $\mathbf{e_i x}$ that a neuron is sensitive to. In particular, the high end of the range picks out the value of $\mathbf{e_i x}$ for which the neuron fires most frequently. If $\mathbf{e_i x}$ is larger than this value, then the neuron is largely saturated, and changes to this value will not be significantly reflected in changes to the neuron's firing rate until the value comes back within the valid range. Maximum firing rate specifies how frequently a neuron is firing when the neuron is saturated. Finally, the firing threshold specifies a lower bound on the values of $\mathbf{e_i x}$ for which the neuron fires; the neuron is inactive in response to values of $\mathbf{e_i x}$ that are below its firing threshold. Maximum firing rates and firing thresholds are chosen randomly for each neuron from a distribution. Numerical values for the parameters used in the neural model, as well as distributions for the values chosen randomly, are presented in Table 3.

For the synapse model, we take pre-synaptic spikes to evoke a post-synaptic current in the dendrites of the stimulated neuron. The equation governing this current is:

$$h_{PSC}(t) = e^{-t/\tau_{PSC}} \tag{A.4}$$

The post-synaptic time constant, τ_{PSC}, controls the shape of this waveform; smaller values cause it to decay faster. All connections between neurons are assumed to be mediated by either AMPA or GABA neurotransmitters, so all post-synaptic time constants are set at 5 ms. The LIF activation function, G, can thus be given by:

$$G_i(t) = \sum_i h_{PSC}(t - t_i) \tag{A.5}$$

where t_i are times of the spikes generated by the sub-threshold dynamics in Equation (A.3).

Parameter	Association Neurons	Standard Neurons
τ_{RC}	34 ms	20 ms
τ_{ref}	2.6 ms	2 ms
τ_{PSC}	5 ms	5 ms
Range	(-1.0, 1.0)	$(\frac{-5}{\sqrt{512}}, \frac{5}{\sqrt{512}})$
Max Firing Rate Dist.	U(200, 350) spikes/s	U(200, 400) spikes/s
Firing Threshold Dist.	0.3	$U(\frac{-5}{\sqrt{512}}, \frac{5}{\sqrt{512}})$

Table A.1: Parameters used in the neural model.

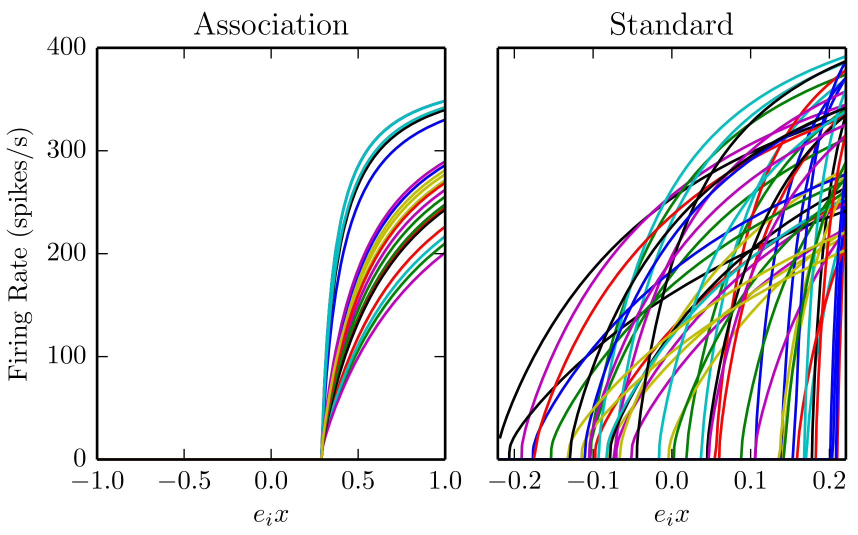


Figure A.1: Firing-rate tuning curves from different sub-population types.

Examples of population tuning curves for both associative and standard (i.e. not in the associative memory) neural sub-populations are shown in Fig. A.1. The effects of several of the parameters can be observed. In particular, the association neurons have firing thresholds above 0, in contrast to the standard neurons. This contributes to the thresholding function of the association populations. Also visible is the wider range for the associative neurons. This is a consequence of the fact that the association populations have to represent dot products which fall roughly within (-0.9, 0.9), whereas the range for the standard neurons is determined by the fact that represented vectors are taken to be 512-dimensional unit vectors, so the expected maximum length along any dimension is $\frac{1}{\sqrt{512}} \approx 0.04$. Finally, the wider spread of maximum firing rates for the standard neurons, which are chosen from the distribution U(100, 200) spikes/s, is evident.

Appendix B

Selecting size of association population in learned associative memory

In order to have a desired number of neurons assigned to any pair of vectors, we need to control how many neurons any given address vector will activate during training. We do this by setting the size of the association population (that is, the number of neurons it contains) as a function of the threshold of the association neurons and the dimensionality of the vectors we are working with. Let t_f denote the firing threshold of the association neurons, let D denote the dimensionality of the vectors we want to store (as well as the encoding vectors of the neurons), let n_{goal} be the approximate number of neurons we want to assign to any given input vector, and let n_{total} be the total number of neurons in the association population.

We assume that both the address vectors and the encoding vectors of the association population are chosen independently and uniformly at random. For an arbitrary address vector ξ, let X denote the random variable that gives the number of neurons active in response to it. This is equal to the number of neurons whose encoding vectors have a dot product with the address vector that exceeds the firing threshold. To determine the probability distribution of X, we require the probability that the dot product between two vectors chosen uniformly at random exceeds the threshold t_f. According to [27], this is given by:

$$p = \frac{\int_0^{arccos(t_f)} \sin^{D-2}\theta d\theta}{B(\frac{D-1}{2}, \frac{1}{2})}$$

where B is the beta function defined as $B(\alpha, \beta) = \int_0^1 t^{\alpha-1}(1-t)^{\beta-1}dt$.

Now since the encoding vectors of the neurons are chosen independently, the number of neurons with encoding vectors whose similarity with ξ is greater than t_f follows a binomial distribution $\mathcal{B}(n_{total}, p)$, with probability distribution:

$$P(x) = \binom{n_{total}}{x} p^x (1-p)^{n_{total}-x}$$

Recall that controlling this probability distribution is our goal. In other words, we want to set n_{total} such that $E[X] = n_{goal}$. We know that $E[X] = n_{total}p$, so we need $n_{total} = \frac{n_{goal}}{p}$.

References

[1] John Robert Anderson. *How can the human mind occur in the physical universe?* Oxford University Press, 2007.

[2] David J. Chalmers. Why fodor and pylyshyn were wrong: The simplest refutation. In *In Proceedings of the 12th Annual Conference of the Cognitive Science Society*, pages 340–347, 1990.

[3] John Conklin and Chris Eliasmith. An attractor network model of path integration in the rat. *Journal of Computational Neuroscience*, 18:183–203, 2005.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[5] L. A. A. Doumas, J. E. Hummel, and C. M. Sandhofer. A theory of the discovery and predication of relational concepts. *Psychological Review*, 115:1–43, 2008.

[6] N.F. Dronkers, S. Pinker, and A. Damasio. Language and the aphasias. In E.R. Kandel, J. Schwartz, and T. Jessell, editors, *Principles in Neural Science*, pages 1169–1187. McGraw-Hill, New York, New York, 4th edition, 2000.

[7] Chris Eliasmith. *How to build a brain: A neural architecture for biological cognition*. Oxford University Press, New York, NY, 2013.

[8] Chris Eliasmith and Charles H Anderson. *Neural engineering: Computation, representation and dynamics in neurobiological systems*. MIT Press, Cambridge, MA, 2003.

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

[10] J L Elman. *Distributed Representations, Simple Recurrent Networks, and Grammatical Structure*, pages 91–122. Connectionist approaches to language learning. Kluwer, Dordrecht, 1991.

[11] C. Fellbaum. *Wordnet: an electronic lexical database*. MIT Press, Cambridge, Massachusetts, 1998.

[12] Brian J Fischer. *A model of the computations leading to a representation of auditory space in the midbrain of the barn owl*. Phd, Washington University in St. Louis, 2005.

[13] Brian J Fischer, José Luis Peña, and Masakazu Konishi. Emergence of multiplicative auditory responses in the midbrain of the barn owl. *Journal of neurophysiology*, 98(3):1181–93, September 2007.

[14] JA Fodor and ZW Pylyshyn. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28:3–71, 1988.

[15] Ross W Gayler. Vector Symbolic Architectures answer Jackendoff's challenges for cognitive neuroscience. In P Slezak, editor, *ICCS/ASCS International Conference on Cognitive Science*, pages 133–138, 2003.

[16] A P Georgopoulos, J T Lurito, M Petrides, A Schwartz, and J Massey. Mental rotation of the neuronal population vector. *Science*, 243:234–236, 1989.

[17] W R Glaser. Picture naming. *Cognition*, 42:61–105, 1992.

[18] Robert F Hadley. The problem of rapid variable creation. *Neural computation*, 21(2):510–32, March 2009.

[19] G Hinton. Where do features come from? In *Outstanding questions in cognitive science: A symposium honoring ten years of the David E. Rumelhart prize in cognitive science*. Cognitive Science Society, 2010.

[20] John E Hummel and Keith J Holyoak. A symbolic-connectionist theory of relational inference and generalization. *Psychological review*, 110(2):220–264, 2003.

[21] Eric Hunsberger, Peter Blouw, James Bergstra, and Chris Eliasmith. A neural model of human image categorization. In *35th Annual Conference of the Cognitive Science Society*, pages 633–638. Cognitive Science Society, 2013.

[22] R Jackendoff. *Foundations of language: Brain, meaning, grammar, evolution*. Oxford University Press, 2002.

[23] D Kuo and Chris Eliasmith. Integrating behavioral and neural data in a model of zebrafish network interaction. *Biological Cybernetics*, 93(3):178–187, 2005.

[24] Sydney M Lamb. *Pathways of the brain: the neurocognitive basis of language*. 4. Current issues in linguistic theory. John Benjamins Publishing Company, 1999.

[25] M. Laubach, M. S. Caetano, B. Liu, N. J. Smith, N. S. Narayanan, and Chris Eliasmith. Neural circuits for persistent activity in medial prefrontal cortex. In *Society for Neuroscience Abstracts*, page 200.18, 2010.

[26] D. B. Lenat. CYC: A Large-Scale Investment in Knowledge Infrastructure. *Communications of the ACM*, 38(11), 1995.

[27] Shengqiao Li. Concise formulas for the area and volume of a hyperspherical cap. *Asian Journal of Mathematics & Statistics*, 4:66–70, 2011.

[28] Abninder Litt, Chris Eliasmith, and Paul Thagard. Neural affective decision theory: Choices, brains, and emotions. *Cognitive Systems Research*, 9:252–273, 2008.

[29] Benjamin Liu, Marcelo Caetano, Nandakumar Narayanan, Chris Eliasmith, and Mark Laubach. A neuronal mechanism for linking actions to outcomes in the medial prefrontal cortex. In *Computational and Systems Neuroscience 2011*, 2011.

[30] David MacNeil and Chris Eliasmith. Fine-Tuning and the Stability of Recurrent Neural Networks. *PLoS ONE*, 6(9):e22885, September 2011.

[31] Brian McLaughlin. Classical constituents in smolenskys ics architecture. In MariaLuisaDalla Chiara, Kees Doets, Daniele Mundici, and Johan Van Benthem, editors, *Structures and Norms in Science*, volume 260 of *Synthese Library*, pages 331–343. Springer Netherlands, 1997.

[32] G Miller, R Beckwith, C Fellbaum, G Gross, and K Miller. Introduction to WordNet: An on-line lexical database. *International Journal of Lexicography*, 3:235–244, 1990.

[33] Erkki Oja. Simplified neuron model as a principal component analyzer. *Journal of Mathematical Biology*, 15(3), 1982.

[34] G Ojemann, J Ojemann, E Lettich, and M Berger. Cortical language localization in left, dominant hemisphere. An electrical stimulation mapping investigation in 117 patients. *Journal Of Neurosurgery*, 71(3):316–326, 1989.

[35] Randall C O'Reilly and Yuko Munakata. *Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain*. The MIT Press, 1 edition, 2000.

[36] A Paivio. *Mental representations: A dual coding approach.* Oxford University Press, New York, 1986.

[37] Bente Pakkenberg and Hans Jørgen G Gundersen. Neocortical neuron number in humans: Effect of sex and age. *The Journal of comparative neurology*, 384(2):312–320, July 1997.

[38] A Peters and E G Jones. *Cerebral Cortex*, volume 1. Plenum Press, New York, 1984.

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

[40] Tony A Plate. *Holographic reduced representations*. CSLI Publication, Stanford, CA, 2003.

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

[42] Z Pylyshyn. *Computation and cognition: Toward a foundation for cognitive science.* MIT Press, Cambridge, MA, 1984.

[43] L Shastri and V Ajjanagadde. From simple associations to systematic reasoning: A connectionist representation of rules, variables, and dynamic bindings. *Behavioral and Brain Sciences*, 16:417–494, 1993.

[44] W Kyle Simmons, Stephan B Hamann, Carla L Harenski, Xiaoping P Hu, and Lawrence W Barsalou. fMRI evidence for word association and situated simulation in conceptual processing. *Journal of physiology, Paris*, 102:106–119, 2008.

[45] R Singh and Chris Eliasmith. Higher-dimensional neurons explain the tuning and dynamics of working memory cells. *Journal of Neuroscience*, 26:3667–3678, 2006.

[46] Paul Smolensky. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence*, 46:159–217, 1990.

[47] K O Solomon and Lawrence W Barsalou. Perceptual simulation in property verification. *Memory and Cognition*, 32:244–259, 2004.

[48] Terrence Stewart, T Bekolay, and Chris Eliasmith. Learning to select actions with spiking neurons in the basal ganglia. *Frontiers in Decision Neuroscience*, 6, 2012.

[49] Terrence Stewart, Trevor Bekolay, and Chris Eliasmith. Neural representations of compositional structures: Representing and manipulating vector spaces with spiking neurons. *Connection Science*, 2011.

[50] Terrence Stewart, Xuan Choo, and Chris Eliasmith. Dynamic Behaviour of a Spiking Model of Action Selection in the Basal Ganglia. In D. D. Salvucci and G. Gunzelmann, editors, *10th International Conference on Cognitive Modeling*, 2010.

[51] Terrence Stewart and Chris Eliasmith. *Compositionality and biologically plausible models*. Oxford University Press, 2012.

[52] Terrence Stewart, Yichuan Tang, and Chris Eliasmith. A biologically realistic cleanup memory: Autoassociation in spiking neurons. *Cognitive Systems Research*, 12:84–92, 2011.

[53] Frank van der Velde and Marc de Kamps. Neural blackboard architectures of combinatorial structures in cognition. *Behavioral and Brain Sciences*, 29(29):37–108, 2006.

[54] Aaron Voelker, Eric Crawford, and Chris Eliasmith. Learning large-scale heteroassociative memories in spiking neurons. In *13th International Conference, UCNC 2014. London, Canada, July 2014 Poster Proceedings*, 2014.

[55] Cristoph von der Malsburg. The correlation theory of brain function, 1981.