Sentence processing in spiking neurons: A biologically plausible left-corner parser

Terrence C. Stewart (tcstewar@uwaterloo.ca)
Xuan Choo (fchoo@uwaterloo.ca)
Chris Eliasmith (celiasmith@uwaterloo.ca)
Center for Theoretical Neuroscience, University of Waterloo
Waterloo, ON, Canada N2L 3G1

Abstract

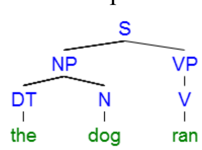
A long-standing challenge in cognitive science is how neurons could be capable of the flexible structured processing that is the hallmark of cognition. We present a spiking neural model that can be given an input sequence of words (a sentence) and produces a structured tree-like representation indicating the parts of speech of that it has identified and their relations to each other. While this system is based on a standard left-corner parser for constituency grammars, the neural natural of the model leads to new capabilities not seen in classical implementations. For example, the model gracefully decays in performance as the sentence structure gets larger. Unlike previous attempts at building neural parsing systems, this model is highly robust to neural damage, can be applied to any binary constituency grammar, and requires relatively few neurons (~150,000).

Keywords: Neural engineering framework; vector symbolic architectures; left-corner parsing; syntax; binary trees; computational neuroscience

Introduction

Human language processing requires not only the ability to represent complex structures, but also the ability to create these representations out of a serial sequence of words. This system is flexible enough to work for a huge variety of possible sentences, including (crucially), sentences that have never been seen before.

Modern linguistics is founded on the principle that these structured representations are tree-like, and that this tree structure imposes useful order on the sentence. For example, "the dog ran" can be parsed as follows:



Here the sentence (S) is divided into a noun phrase (NP) and a verb phrase (VP). The noun phrase is divided into a determiner (DT) "the" and a noun (N) "dog". The verb phrase consists of a single verb (V) "ran".

To describe the space of possible trees, rules as to what constituents can be found within other constituents can be defined. For example, the above sentence is consistent with the following constituency grammar rules:

S → [NP VP]	DT → "the"
NP → [DT N]	N → "dog"
VP → [V]	V → "ran"

By adding more structural rules (on the left), we can build more complex sentences. By adding more words (on the right), we can increase our vocabulary.

Left-Corner Parsing

As more rules are included in the grammar, parsing becomes more complicated. More rules mean more possibilities, and finding a tree that is consistent with the rules can become computationally expensive. For example, there may be the two rules VP → [V] and VP → [V NP], leading to an explosion of possible structures to search through. Furthermore, a single word may have multiple interpretations, such as N → "dog" and V → "dog".

One standard approach to addressing this problem is to use a Left-Corner parsing algorithm. The idea here is to combine bottom-up information with top-down information. The top-down information is what part of speech we are currently looking for (a sentence, a noun phrase, a determiner, etc.). The bottom-up information is the single word we are currently processing. So, if we are currently looking for a noun, we will first try interpreting the word "dog" as a noun, rather than a verb. If this does not lead to a successful parse, then the algorithm will backtrack to this point and try the other option. This approach drastically reduces the amount of backtracking needed.

For "the dog ran", the algorithm proceeds as follows:

- Top-down: look for S
- Bottom-up: see "the"
- Apply rule DT → "the"
- Apply rule NP → [DT N]
- Store the partially completed tree
    - Top-down: look for N for previous rule
    - Bottom-up: see "dog"
    - Apply rule N → "dog"
    - Merge with previously stored tree
- Apply rule S → [NP VP]
- Store the partially completed tree
    - Top-down: look for VP for previous rule
    - Bottom-up: see "ran"
    - Apply rule V → "ran"
    - Apply rule VP → [V]
    - Merge with previously stored tree

As has been pointed out (Johnson-Laird, 1983), this algorithm matches well with observed human sentence comprehension. For example, it has difficulty with "garden-path" sentences such as the classic "the horse raced past the barn fell", which is difficult for most people to interpret even though it has a similar form as the easier sentence "the deer shot by the hunter fell". A left-corner parser has the same tendency as people do to connect the ambiguous word "raced" as part of a S → [NP VP] rule.

Previous Models

Cognitive models of left-corner parsing already exist. For example, Lewis and Vasishth (2005) present an implementation using the ACT-R cognitive architecture. In that work, a set of IF-THEN production rules (for selecting grammar rules to apply) are combined with a declarative memory system (for storing the partially completed parts of the tree). The result is a computational model that shows how existing cognitive modules (for which there is strong prior evidence they exist in the human brain) can be re-purposed to perform left-corner parsing. The model's reaction times and error patterns match well to human subjects. However, this work does not offer a neural explanation of how those particular modules and structures could be physically instantiated within the human brain. Indeed, the question of how language could be represented and manipulated by interacting neurons is a long-standing question in cognitive science (e.g. Jackendoff, 2002).

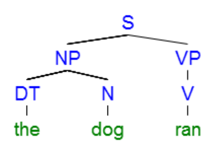
For a neural explanation of this process, van der Velde and de Kamps (2006) offer their Neural Blackboard Architecture. Here, a set of specific neural groups can temporarily come to represent different nouns, verbs, adjectives, and so on. This makes it possible to represent structured information such as sentences. While we have previously argued (Stewart & Eliasmith, 2012) that the neural structures proposed by this approach are inefficient and do not correspond to those seen in real brains, the core difference here is that they have only shown how specific cases of sentence patterns might be parsed, rather than presenting a general method for parsing arbitrary compositional grammar rules, as is attempted here.

Finally, we previously presented a system for parsing highly specific sentence patterns, but using a biologically realistic neural model (Stewart & Eliasmith, 2013). These parsed commands, such as "write two" or "if see eight write three", could be successfully interpretted as commands and used to guide action (Choo & Eliasmith, 2013). However, as with the work by van der Velde and de Kamps, this model was restricted to parsing highly specific grammatical forms. The purpose of this paper is to generalize to significantly more complex grammars and to connect this neural work to the grammatical structures seen in established linguistic theory.

Neural Engineering Framework

Vector Symbolic Architectures

For example, we can represent the tree for "the dog ran" as follows:



$$\mathtt{S} = \mathtt{the} \circledast \mathtt{DT} \circledast \mathtt{NP}_L \circledast \mathtt{S}_L + \mathtt{dog} \circledast \mathtt{N} \circledast \mathtt{NP}_R \circledast \mathtt{S}_L + \mathtt{ran} \circledast \mathtt{V} \circledast \mathtt{VP} \circledast \mathtt{S}_R$$

This computes a single vector S which store that particular tree. The terms `the`, `dog`, and `ran` are randomly chosen vectors for each of those words. The terms $\mathtt{DT}$, $\mathtt{N}$, $\mathtt{NP}_L$, $\mathtt{S}_R$, and so on are also randomly chosen vectors. These are used to indicate the structure of the tree. Note that there are different vectors for taking the left or right branch of the tree (the `R` or `L` subscript). This is so that the vector for a version of this tree with swapped branched ("ran the dog") would be different than the vector for this tree ($\mathtt{S}_L$ and $\mathtt{S}_R$ would be swapped).

A similar approach could be used for trees with more than just binary branching (i.e. trees where there can be more than two children of a node). However, for the purposes of the left-corner algorithm implemented here, we only consider binary rules. Note that binary constituency rules (e.g. S → [NP VP]) are sufficient to implement the widely used standard X-bar grammars.

Show extracting

Mention vocab sizes

Semantic Pointer Architecture

Mention number of neurons per rule: 300

Left-Corner Parsing in SPA

To develop a neurally plausible implementation of left-corner parsing within the Semantic Pointer Architecture, we need to define cortical modules, their functions, and the basal ganglia/thalamus rules that coordinate the flow of information between these modules.

For cortical modules, we only need one basic component: a *buffer* capable of storing a semantic pointer. This is a module that stores a single high-dimensional vector (for this model, 1000 dimensions is sufficient) over time. We need three of these: one to store the tree being built (*tree*), one to store the current top-down goal (*goal*; what part of speech we are looking for), and one to store the partially completed trees (*partial*). Note that the neural modules needed to visually recognize words, or to move visual attention from one word to the next, are not considered here (although see [ref:vision] and [ref:attention] for potential modules).

Since these buffers store vectors, we can use the approach of Vector Symbolic Architectures to store a tree within them. We assume words are presented sequentially, their vectors being stored in the buffer called *tree*.

We now need rules for the basal ganglia and thalamus system to cause the model to implement the parsing algorithm. We start with simple rules of the form X → [Y]. For each of these, we need a system that says if we're currently parsing a Y, we should build a tree that consists of an X connected to a Y. In terms of vectors, this means building a new vector that is X + *tree*⊛X. For example, if the *tree* buffer contains `ran` and we have a rule `V → ran`, we want to compute `V + ran⊛V` and store that in the *tree* buffer. This can be written in SPA form as follows:

U_i: *tree* • Y
E_i: *tree* ← X + *tree*⊛X

Note that the utility of this rule is the dot product of whatever is in the *tree* buffer and the Y part of the rule, so it will only be active when the *tree* looks like Y.

To see how this helps to build up the desired tree, consider what happens if there is another similar rule implementing `VP` → [`V`]. Once the first rule is active, the *tree* buffer will contain `V + ran⊛V`. This will cause the rule for `VP` → [`V`] to have a high utility (since its utility is *tree* • `V`). The effect of the rule is *tree* ← `VP` + *tree*⊛`VP`, so the result will be `VP + (V + ran⊛V) ⊛VP`. This vector is highly similar to `ran⊛V⊛VP`, which is part of what we need for parsing the complete tree for “the dog ran”.

We also need to handle rules of the form X → [Y Z], which gives the branching capability to the tree. Here we need to not only build up a tree, but we also need to set a new top-down goal to find a Z. The utility is the same as in the previous rule (we want the rule to be active when *tree* contains a Y), but we also want to store the partially completed tree and go on to processing the next word. An initial version of this rule's effect would be:

E_i: *partial* $\leftarrow X + tree \circledast X_L$
tree $\leftarrow$ the next word
goal $\leftarrow Z$

This rule would successfully store the partially completed tree in the *partial* buffer, and then would go on to start trying to parse the next word, trying to find a Z. However, setting this new *goal* in this way would completely replace the old value in *goal*. Indeed, setting the new *partial* tree would completely erase any previous *partial* tree.

To deal with this, we exploit the vector-based representation to build an approximation of a "stack". For example, if the goal currently contains the vector for `S` (a sentence), but due to a rule like `NP` → [`DT N`] we now need to look for an `N` (a noun), we can set the new *goal* to be $\mathtt{N}+goal\circledast\mathtt{STACK}$, where `STACK` is another random vector. In this case, the *goal* will now be $\mathtt{N}+\mathtt{S}\circledast\mathtt{STACK}$. If we now need to look for a `V` (a verb), we would compute $\mathtt{V}+goal\circledast\mathtt{STACK}$, giving $\mathtt{V}+(\mathtt{N}+\mathtt{S}\circledast\mathtt{STACK})\circledast\mathtt{STACK}$, or $\mathtt{V}+\mathtt{N}\circledast\mathtt{STACK}+\mathtt{S}\circledast\mathtt{STACK}\circledast\mathtt{STACK}$. Note that to remove an item from the stack, we can compute $goal\circledast\mathtt{STACK}^{-1}$, which would give us back an approximation of $\mathtt{N}+\mathtt{S}\circledast\mathtt{STACK}$. This approximation will gradually become worse as the number of items in the stack increases.

The resulting rule for X → [Y Z] is as follows:

U_i: $tree \bullet Y$
E_i: *partial* $\leftarrow X + tree\circledast X_L + partial\circledast\mathtt{STACK}$
tree $\leftarrow$ the next word
goal $\leftarrow Z + goal\circledast\mathtt{STACK}$

Finally, we need a top-down rule to recognize when we have found a part of speech we are looking for. When it does so, we combine it with the partial tree on the stack, remove it from the stack, and continue. For a rule of the form X → [Y Z], we get:

U_i: $(partial \bullet X)(goal \bullet tree)$
E_i: *tree* $\leftarrow partial + X_R\circledast tree$
goal $\leftarrow goal\circledast\mathtt{STACK}^{-1}$
partial $\leftarrow partial\circledast\mathtt{STACK}^{-1}$

So, if we are on the last word of "the dog ran", the stored value in *partial* will be close to $\mathtt{the}\circledast\mathtt{DT}\circledast\mathtt{NP_L}\circledast\mathtt{S_L} + \mathtt{dog}\circledast\mathtt{N}\circledast\mathtt{NP_R}\circledast\mathtt{S_L}$ and the value in *tree* will get built up to approximately $\mathtt{ran}\circledast\mathtt{V}\circledast\mathtt{VP}$, as indicated previously. The goal will be `VP`. This means that the utility for this top-down version of the `S` → [`NP VP`] rule will be high (since $partial \bullet \mathtt{S}$ will be large and $goal \bullet tree$ will be large). The resulting tree will be $partial + \mathtt{S_R}\circledast tree$, or approximately $\mathtt{the}\circledast\mathtt{DT}\circledast\mathtt{NP_L}\circledast\mathtt{S_L} + \mathtt{dog}\circledast\mathtt{N}\circledast\mathtt{NP_R}\circledast\mathtt{S_L} + \mathtt{ran}\circledast\mathtt{V}\circledast\mathtt{VP}\circledast\mathtt{S_R}$. This is the desired vector for the correct parse of the tree.

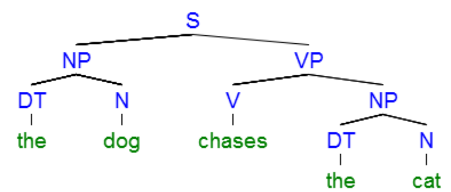
This algorithm will work for arbitrary binary construction grammar rules. However, it needs to be extended to deal with multiple possible rules that could be applied at once (ambiguous sentences), as is discussed below.

Parsing Results

This biologically plausible left-corner parsing algorithm is capable of computing the vector-based tree representation of a sentence, given a set of constituency grammar rules. These rules are converted into connections between the cortex, basal ganglia, and thalamus, as per the Semantic Pointer Architecture. That neural structure is then capable of parsing sentences consistent with those rules.

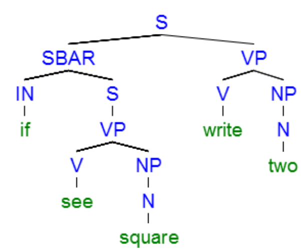
In addition to the "the dog ran" example given previously, the following set of rules allows the model to parse "the dog chases the cat".

VP → [V NP] NP → [DET N]
S → [VP] NP → [N]
S → [NP VP]



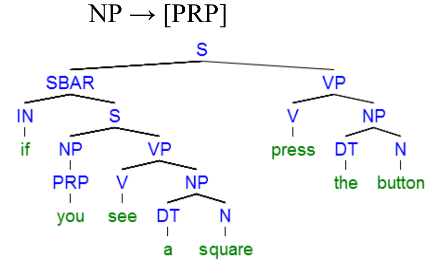
To demonstrate more complex parsing, we add rules for subordinate clauses. In previous work (Choo & Eliasmith, 2013) we had built neural models that could follow orders of the form "if see square, press button". Rather than using the special-case neural parser we had built previously (Stewart & Eliasmith, 2013), the parser presented here can be extended by adding the following rules, where IN is a subordinating conjunction (such as "if"), and SBAR is a subordinating conjunctive phrase ("if see square").

S → [SBAR VP] SBAR → [IN S]



Importantly, this system can also parse more grammatically complete sentences. Indeed, if we also add the following rule for PRP (personal pronouns), then the system is able to successfully parse "if you see a square, press the button".

NP → [PRP]



Parsing Accuracy

While the model is capable of parsing the previous sentences, its capabilities are not perfect. Indeed, the accuracy of the parsing is dependent on the accuracy of the neural representation. This is a concrete example of the competence vs. performance distinction: while the underlying algorithm may be have the competence to parse those sentences, the neural implementation may not match that in terms of actual performance.

We can analyze this by determining the probability of correctly parsing the sentence as we increase the amount of noise. A word is considered to be correctly parsed if we can extract it from the vector for the sentence, using the standard approach to extracting in Vector Symbolic Architectures. For example, for the "the dog ran" the ideal resulting vector is $\mathtt{S} = \mathtt{the}\circledast\mathtt{DT}\circledast\mathtt{NP}_L\circledast\mathtt{S}_L + \mathtt{dog}\circledast\mathtt{N}\circledast\mathtt{NP}_R\circledast\mathtt{S}_L + \mathtt{ran}\circledast\mathtt{V}\circledast\mathtt{VP}\circledast\mathtt{S}_R$. To determine if "the" was parsed correctly, we compute $\mathtt{S}\circledast(\mathtt{DT}\circledast\mathtt{NP}_L\circledast\mathtt{S}_L)^{-1}$. If this vector is closer to the vector for `the` than to any other vector in the vocabulary of 10,000 words, then the parse is considered to be correct. Noise is adjusted by adding random values to the stored *tree*, *partial*, and *goal* vectors after every rule. Finally, if the algorithm fails (i.e. no words can be extracted), then we retry it until it succeeds. Figure 1 shows that performance gets much worse with noise values of 0.15 or more, and that even small values of noise require an average of 0.6 retries.

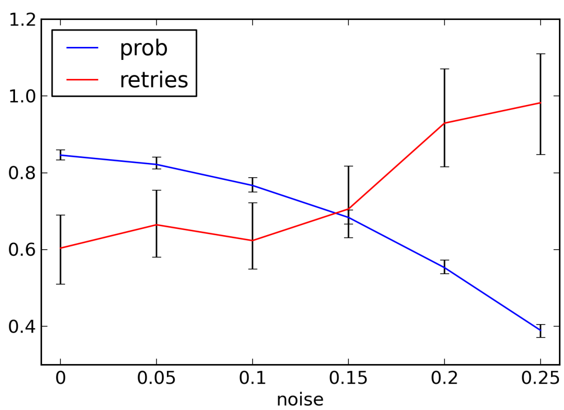


Figure 1: The probability of successful parsing and the number of retries needed as the random noise varies. 95% confidence intervals are shown.

These results give us a lower bound on the accuracy of the neural representation that is needed for the buffers. Since the Neural Engineering Framework indicates that accuracy is increased as more neurons are added, we can find how many neurons are needed for each buffer to achieve this level of accuracy. We do this by creating groups of neurons with connection weights between them optimized via the NEF to compute the function $\frac{dx}{dt} = 0$ (i.e. the value x that is being stored should not change). The network is initialized to contain a randomly chosen vector x, and the model is run for 50 milliseconds. We have previously shown that 50 milliseconds is the average amount of time taken between rule activation (Stewart, Choo, & Eliasmith, 2010).

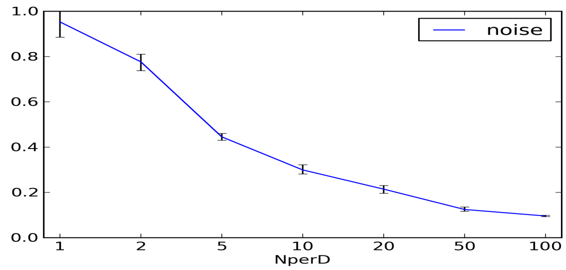


Figure 2: Amount of representational noise for different numbers of neurons. 95% confidence intervals are shown.

Figure 2 shows that 50 to 100 neurons are needed per dimension to achieve noise levels around 0.15. Since Figure 1 indicates that performance is much improved with noise below 0.15, and since we are using vectors with 1000 dimension, we build our final model with 50,000 neurons per buffer. The complete model has 3 buffers and 18 basal ganglia/thalamus/cortex rules (300 neurons each), resulting in a parsing system with 155,400 neurons. Importantly, Figures 1 and 2 also show that the model exhibits graceful degradation. If neurons are removed (or die), accuracy will gradually decrease.

Using Parsed Commands

While the model presented thus far is capable of parsing sentences, the real test is to be able to parse a sentence and then make use of that information. In previous work (Choo & Eliasmith, 2013), we presented a spiking neural model that is capable of following instructions of the form "If <condition>, <do action>". However, that model did not perform the parsing itself. It relied on a special-case neural parser that only worked on particular word patterns (Stewart & Eliasmith, 2013). Since they both make use of the Semantic Pointer Architecture, the general-purpose left-corner parser presented here can be combined with the instruction-following model.

In the instruction-following model, the instruction "if see square, write two" was encoded as:

```
CONDITION⊛(SENSE⊛VISION + SENSE_DATA⊛SQUARE)
 + ACTION⊛(MOTOR⊛WRITE + MOTOR_DATA⊛TWO)
```

With the parser presented here, we can directly use the parsed sentence as a command to the instruction-following model by making the following definitions:

$$\begin{aligned}
&\mathtt{CONDITION} = \mathtt{S}_L\\
&\mathtt{SENSE} = \mathtt{V}\circledast\mathtt{VP}_L\circledast\mathtt{S}\circledast\mathtt{SBAR}_R\\
&\mathtt{SENSE_DATA} = \mathtt{N}\circledast\mathtt{NP}\circledast\mathtt{VP}_R\circledast\mathtt{S}\circledast\mathtt{SBAR}_R\\
&\mathtt{ACTION} = \mathtt{S}_R\\
&\mathtt{MOTOR} = \mathtt{V}\circledast\mathtt{VP}_L\\
&\mathtt{MOTOR_DATA} = \mathtt{N}\circledast\mathtt{NP}\circledast\mathtt{VP}_R
\end{aligned}$$

The result is a biologically plausible neural model where we can feed in a set of commands as sentences, and the model can parse those sentences, remember them, and apply them to incoming stimuli. For example, Figure 3 shows the effect of a system configured to follow these instructions:

P1. If see square write 1
P2. If see circle write 2
P3. If hear one press button1
P4. If hear two press button2

The the visual and sensory inputs to the model change (top four rows of Figure 3), the model successfully responds as appropriate (bottom two rows). Of course, the model presented here does not include all the details necessary to actually perform the motor actions or the visual processing necessary to complete these tasks. The purpose here is to show that the model is capable of correctly selecting the task to perform.

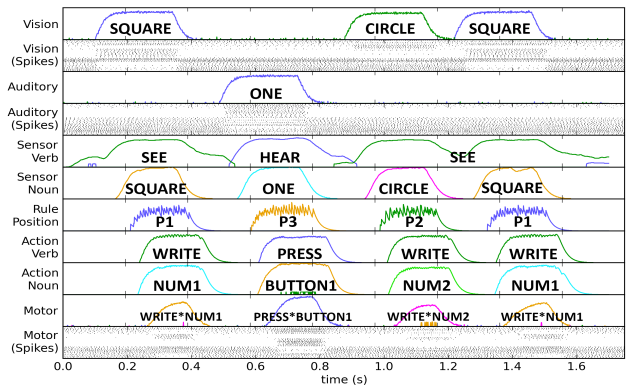


Figure 3: The instruction-following model. Each row is a different group of neurons. Labelled lines indicate the value represented by this group of neurons is close to that vector. First the model is visually shown a SQUARE (top row), and it responds correctly with the motor action of WRITE⊛NUM1 (write the number 1; bottom row). It then hears a ONE, is shown a CIRCLE, and a SQUARE again. The correct motor response is given each time.

Conclusions and Future Directions

Automatically learning utilities to handle repair/recovery
Explore dimensions/accuracy tradeoff
Timing of ambiguous sentences
Context-dependency

References

Ball, J. (2011). A Pseudo-Deterministic Model of Human Language Processing. *33rd Cog. Sci. Society Conference.*

Choo, X and Eliasmith, C. (2013). General Instruction Following in a Large-Scale Biologically Plausible Brain Model. *35th Cog. Sci. Society Conference.*

Eliasmith, C. (2013). *How to build a brain*. Oxford University Press, New York, NY.

Eliasmith, C. & Anderson, C. (2003). *Neural Engineering.* Cambridge: MIT Press.Eliasmith et al., 2012

Gayler, R. (2003). Vector Symbolic Architectures Answer Jackendoff's Challenges for Cognitive Neuroscience, in Slezak, P. (ed). *Int. Conference on Cognitive Science,* Sydney: University of New South Wales, 133–138.

Jackendoff, R. (2002). *Foundations of language: Brain, meaning, grammar, evolution.* Oxford, UK.

Johnson-Laird, P. N. (1983). *Mental models*. Cambridge, MA: Harvard University Press.

Lewis and Vasishth (2005)

Plate, T. (2003). *Holographic Reduced Representations*, CSLI Publications, Stanford, CA.

Stewart, T.C., Choo, X., and Eliasmith, C. (2010). Dynamic Behaviour of a Spiking Model of Action Selection in the Basal Ganglia. *10th Int. Conf. on Cognitive Modeling.*

Stewart, T.C., and Eliasmith, C. (2012). Compositionality and biologically plausible models. In *Oxford Handbook of Compositionality*. Oxford University Press, 2012.

Terrence C. Stewart and Chris Eliasmith. Parsing sequentially presented commands in a large-scale biologically realistic brain model. In 35th Annual Conference of the Cognitive Science Society, 3460-3467. Cognitive Science Society, 2013.

van der Velde and de Kamps (2006)