Biologically Inspired Cognitive Architectures (2016) xxx, xxx–xxx

Available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/bica

INVITED ARTICLE

Large-scale cognitive model design using the Nengo neural simulator

Sugandha Sharma [*], Sean Aubin, Chris Eliasmith

Centre for Theoretical Neuroscience, University of Waterloo, Waterloo, ON N2L 3G1, Canada

Received 23 February 2016; received in revised form 4 May 2016; accepted 18 May 2016

KEYWORDS
Neural models;
Neural Engineering Framework;
Biologically plausible spiking networks;
Semantic pointer architecture

Abstract
The Neural Engineering Framework (NEF) and Semantic Pointer Architecture (SPA) provide the theoretical underpinnings of the neural simulation environment Nengo. Nengo has recently been used to build Spaun, a state-of-the-art, large-scale neural model that performs motor, perceptual, and cognitive functions with spiking neurons (Eliasmith et al., 2012). In this tutorial we take the reader through the steps needed to create two simpler, illustrative cognitive models. The purpose of this tutorial is to simultaneously introduce the reader to the SPA and its implementation in Nengo.

1. Introduction

This paper explains, by tutorial demonstration, recent state-of-the-art methods for building large-scale neural models of cognition. The methods we introduce have recently been used to build ''Spaun", which remains the world's largest functional brain model (Eliasmith et al., 2012). Spaun, consists of 2.5-million spiking neurons, includes twenty different brain areas, and is capable of performing eight different cognitive tasks (including digit recognition, list memory, addition, and pattern completion). Spaun receives input through a single eye with a 28 by 28 retina while controlling a simulated, three-joint, six-muscle arm. The flow of information through cortex is controlled internally using a cortex-basal-ganglia-thalamus loop, allowing the same model to perform a variety of tasks.

In this tutorial, we first give an overview of Nengo, the simulation package used to build Spaun. This is followed by a brief introduction to the neural theory underlying Nengo, the Neural Engineering Framework (NEF) (Eliasmith & Anderson, 2004). Next, we describe the Semantic Pointer Architecture (SPA) (i.e., a functional, neuroanatomical, cognitive architecture that exploits the NEF (Eliasmith, 2013)), and its implementation in Nengo. Finally, to demonstrate the utility of SPA, we explain how to implement a simple cognitive model in Nengo in Section 'Question answering model' followed by a more complex SPA model in Section 'Instruction following model'. We conclude by

[*] Corresponding author.
E-mail addresses: s72sharm@uwaterloo.ca (S. Sharma), saubin@uwaterloo.ca (S. Aubin), celiasmith@uwaterloo.ca (C. Eliasmith).

http://dx.doi.org/10.1016/j.bica.2016.05.001
2212-683X/© 2016 Elsevier B.V. All rights reserved.

reiterating the most important concepts and discussing some of the features of Nengo omitted from this tutorial.

2. Nengo

Nengo is a Python-based neural simulation package which uses a small set of simple objects to create functional spiking neural networks. These objects, and their core behaviours implement the NEF, but are also capable of implementing generic spiking neural networks. A detailed description of these objects can be found in Bekolay et al. (2013). In brief, the objects are Ensembles (group of neurons), Nodes (non-neural components, used to control abstract components such as robots arms), Connections (connections between any combination ensembles and nodes) and Networks (groupings of all the aforementioned objects).

To aid in the design and testing of Nengo networks, a Graphical User Interface (GUI) was created with a network visualizer and a live-editing environment as shown in Fig. 1. The examples in this tutorial are constructed in this environment.

This tutorial assumes that you have Python, Nengo and Nengo GUI installed. The latter two packages can be installed on a computer with Python by running `pip install nengo_gui`. Detailed instructions for installing these can be found at the http://python.org, http://github.com/nengo/nengo and http://github.com/nengo/nengo_gui.

3. The Neural Engineering Framework (NEF)

The NEF is based on three principles for the construction of biologically plausible simulations of neural systems (Eliasmith & Anderson, 2004). These are the principles of Representation, Transformation and Dynamics.

NEF models are typically constrained by the available neuroscience evidence to a significant extent. In particular, tuning curves of neurons measured during experiments are typically provided during model construction. Nengo provides flexibility to the modeller to set parameters on ensembles, like the firing rates of neurons, representational radius, intercepts, etc. to capture those tuning curves. It also provides an option to set the synaptic time constants on connections, to reflect known biological constraints regarding neurotransmitters. As well, anatomical constraints can be enforced by structuring the model to respect known connectivity. It is up to the modeller to choose these parameters and methods of organization to enforce constraints based on neuroscience evidence for a given model or part of the brain which they are modelling. The benefits of biological plausibility are described in more detail in Section 'SPA models with the NEF'.

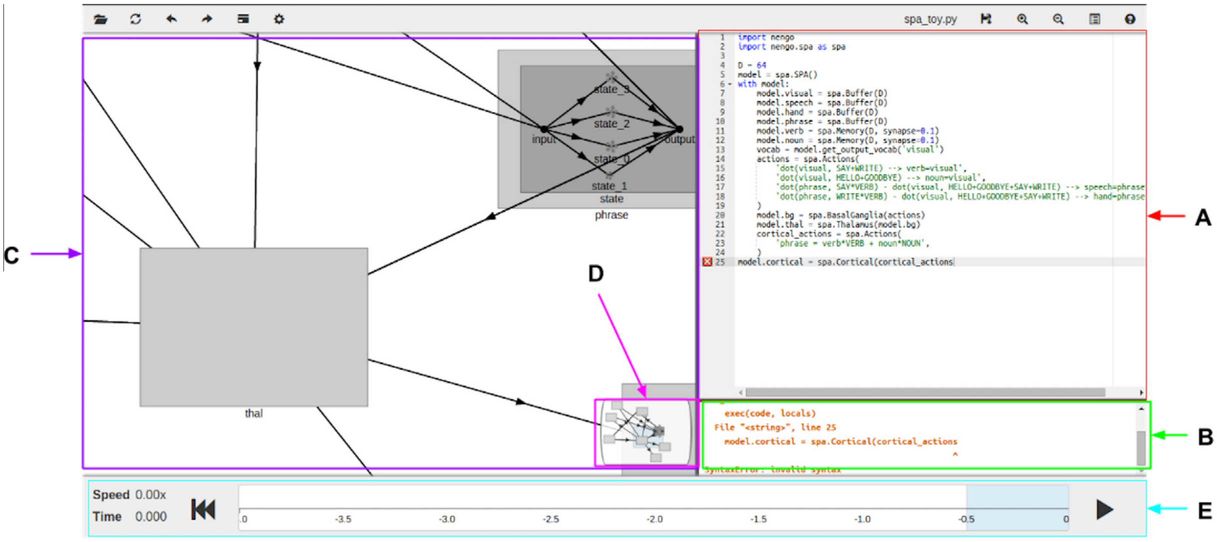
This section briefly demonstrates each of the three NEF principles through basic examples. Additional pre-built examples can be found in the nengo_gui/examples/tutorial folder. Relevant examples contained in that folder are mentioned in each of the following sections with the heading *Relevant Examples*.

3.1. Representation

Relevant Examples: 01-one-neuron.py, 02-two-neurons.py, 03-many-neurons.py, 07-multiple-dimensions.py.

Neural representations in NEF are defined by the combination of nonlinear encoding and weighted linear decoding. These representations are considered to be encoding relatively low-dimensional state vectors into high-dimensional neural populations. Fig. 2 shows a group of 20 neurons representing a scalar value. To construct a simulation of this representation in Nengo, follow these steps:

- Create an empty Python file and open it with the Nengo GUI.
- Type the code shown in Fig. 2 into the text editor of the GUI. Notice that the corresponding objects start to appear in the network visualizer as you create them in the text editor.

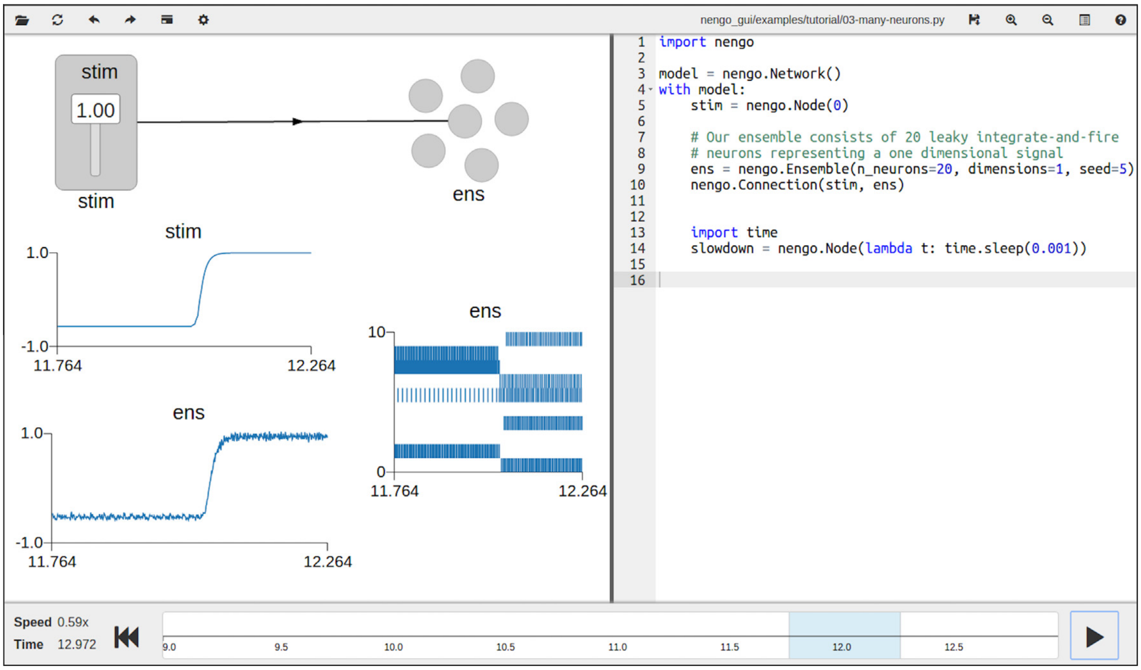


spa_toy.py

```
import nengo
import nengo.spa as spa

D = 64
model = spa.SPA()
with model:
    model.visual = spa.Buffer(D)
    model.speech = spa.Buffer(D)
    model.hand = spa.Buffer(D)
    model.phrase = spa.Buffer(D)
    model.verb = spa.Memory(D, synapse=0.1)
    model.noun = spa.Memory(D, synapse=0.1)
    vocab = model.get_output_vocab('visual')
    actions = spa.Actions(
        'dot(visual, SAY+WRITE) --> verb=visual',
        'dot(visual, HELLO+GOODBYE) --> noun=visual',
        'dot(phrase, SAY*VERB) - dot(visual, HELLO+GOODBYE+SAY+WRITE) --> speech=phrase
        'dot(phrase, WRITE*VERB) - dot(visual, HELLO+GOODBYE+SAY+WRITE) --> hand=phrase
    )
    model.bg = spa.BasalGanglia(actions)
    model.thal = spa.Thalamus(model.bg)
    cortical_actions = spa.Actions(
        'phrase = verb*VERB + noun*NOUN',
    )
model.cortical = spa.Cortical(cortical_actions
```

```
  exec(code, locals)
 File "<string>", line 25
  model.cortical = spa.Cortical(cortical_actions
                                                ^
SyntaxError: invalid syntax
```

Fig. 1 Nengo GUI. (**A**) A live model script editor. (**B**) A console showing compile errors. (**C**) The explorable network visualizer, zoomed. (**D**) A minimap showing the whole network. (**E**) The simulator controls.



```
import nengo

model = nengo.Network()
with model:
    stim = nengo.Node(0)

    # Our ensemble consists of 20 leaky integrate-and-fire
    # neurons representing a one dimensional signal
    ens = nengo.Ensemble(n_neurons=20, dimensions=1, seed=5)
    nengo.Connection(stim, ens)


    import time
    slowdown = nengo.Node(lambda t: time.sleep(0.001))

```

Fig. 2 Ensemble of neurons representing a scalar value.

- Now right click on both *stim* and *ens* objects and select *Value* to display their corresponding value plots. Also display the *Slider* from the right-click menu of *stim* node, and the *Spikes* from the right-click menu of the *ens* node.
- Run the simulation by pressing the play button and use the slider to change the input value provided by the *stim* node.

Notice that the value represented by the ensemble tracks the input value. However, as you reduce the number of neurons to 1, the representation becomes worse (i.e., the value represented by the ensemble doesn't always follow the input). As you increase the number of neurons, the representation will improve. In this example, the non-linear encoding of the first NEF principle is determined by the spiking of the default leaky integrate-and-fire (LIF) neurons generated in the ensemble. The neural spikes are encoding the *stim* input. The linear decoding is automatically computed for you, and used to generate the Value plots. This principle naturally extends to multi-dimensional vector representations (see the 07-multiple-dimensions.py example).

The default parameters of LIF neurons in the ensemble are randomly sampled from a uniform distribution for each neuron. Specifically, encoders are sampled from a uniform hypersphere i.e., an *n*-dimensional unit hypersphere, intercepts and firing rates are sampled from a uniform distribution from $[-1, 1]$ and $[200, 400]$ respectively. The modeller has the ability to chose these distributions according to the empirical data specific to the region of the brain they want to model. They can set the range of these distributions or even use other distributions which are built into Nengo (or define their own distribution using the Nengo interface).

3.2. Transformation

Relevant Examples: 10-transforms.py, 05-computing.py, 09-multiplication.py. The second principle of the NEF states that both linear and non-linear functions can be computed on the represented values by linear decodings. These linear decodings are combined with the neural encodings to determine the connection weights between populations of neurons. The function to be computed by the weights is specified in Nengo when making a connection between ensembles (the default function is identity, *i.e.* $f(x) = x$).

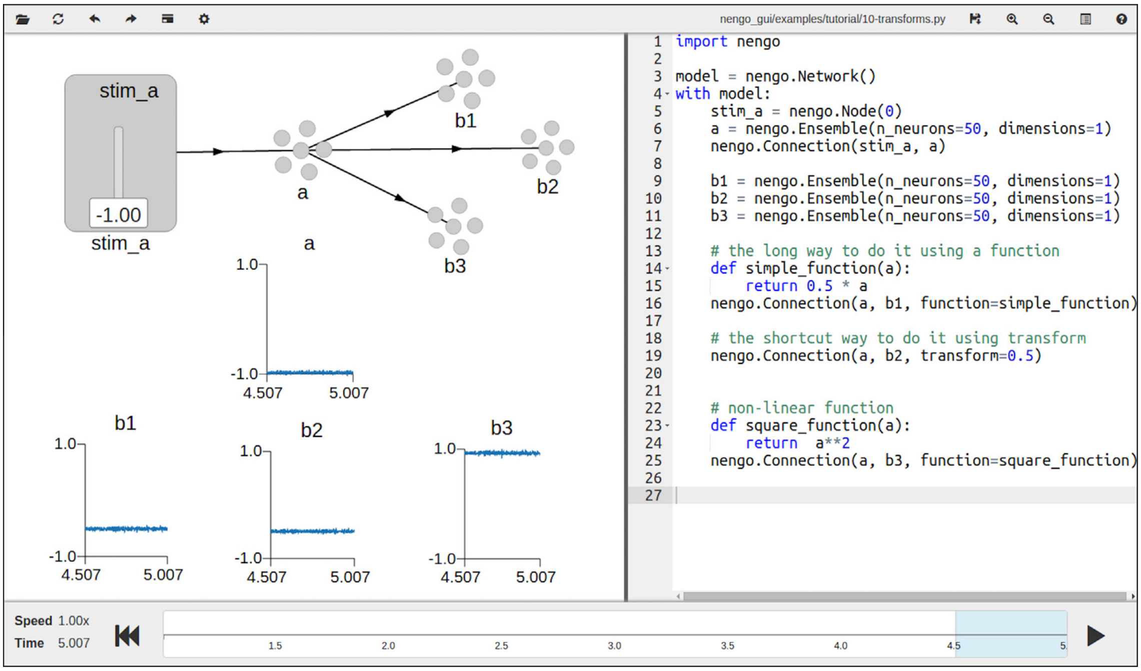
Fig. 3 demonstrates two ways of specifying these functions in Nengo. The first method allows any linear or nonlinear function to be specified, the second allows only linear transforms to be specified. It is often useful to specify linear and non-linear transformations independently.

Because linear transforms, such as scaling, rotating, or shearing, are common in neural models, Nengo has a ''transform'' parameter defined on the Connection object for the purpose of defining these transforms. The example in Fig. 3 shows the equivalence of these approaches. In this example, b1 and b2 represent the same value i.e., the value of ensemble a scaled by 0.5 as shown in their value plots. However, b3 represents the square of the value of the ensemble, illustrating a non-linear transformation.

3.3. Dynamics

Relevant Examples: 11-memory.py, 12-differential-eqns.py, 13-oscillators.py, 14-controlled-oscillator.py, 15-lorenz.py.

The third principle of the NEF states that the values represented by neural populations can be considered state variables in a (linear or non-linear) dynamical system. These



```python
import nengo

model = nengo.Network()
with model:
    stim_a = nengo.Node(0)
    a = nengo.Ensemble(n_neurons=50, dimensions=1)
    nengo.Connection(stim_a, a)

    b1 = nengo.Ensemble(n_neurons=50, dimensions=1)
    b2 = nengo.Ensemble(n_neurons=50, dimensions=1)
    b3 = nengo.Ensemble(n_neurons=50, dimensions=1)

    # the long way to do it using a function
    def simple_function(a):
        return 0.5 * a
    nengo.Connection(a, b1, function=simple_function)

    # the shortcut way to do it using transform
    nengo.Connection(a, b2, transform=0.5)


    # non-linear function
    def square_function(a):
        return  a**2
    nengo.Connection(a, b3, function=square_function)

```

Fig. 3 Two ways of implementing transformations in Nengo. The first is demonstrated by the definition and use of the simple_function. The second is demonstrated by setting the value of the transform parameter. The square_function shows a way of implementing a non-linear transformation.

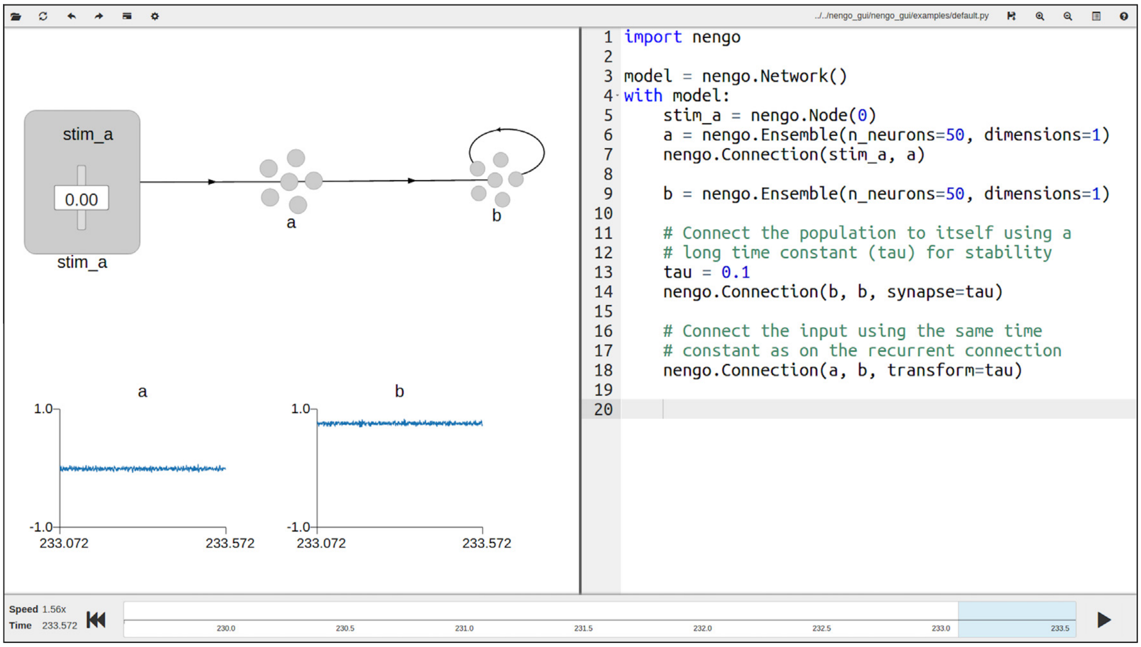
dynamical systems are built using recurrent connections, which can be computed using the second principle. Fig. 4 shows an example of a recurrent network, where ensemble b is connected back to itself, which means that it can store data (i.e., values) over time. If the input is positive, the stored value in b increases, and it decreases if the input is negative (i.e. it integrates (sums) its input). When the input is turned to zero, the value represented by b stays constant thus indicating that neurons can be used to implement stable dynamics. In general, the NEF provides a method for directly converting any set of differential equations into a spiking network approximation of the dynamics described by those equations (Eliasmith & Anderson, 2004). However, describing those methods is beyond the scope of this tutorial.

3.4. SPA models with the NEF

Provided the resources of the NEF as encapsulated by Nengo, it is possible to build large-scale neural models. We have recently suggested a general architecture that aids in structuring such models in a biologically constrained manner (Eliasmith, 2013). We call this architecture the Semantic Pointer Architecture (SPA), and have provided a Python module in Nengo to assist in the construction of SPA-based models. The SPA module adds a higher-level set of objects to Nengo, as well implementing a domain-specific language (discussed in greater detail in Section 'In struction following model') designed for cognitive modelling. The SPA describes cognitive function in terms of vector-based algorithms allowing for structured information in a representation distributed across neurons. In doing so, the SPA employs a specific compression operator for cognitive representations (that is distinct from the compression operator for vision, motor control, etc.). Specifically, the SPA uses circular convolution Plate (1991) coupled with neural saturation to fulfill this role. Circular convolution is one of a class of operators that have been proposed to be able to fill such a role (including XOR, multiplication, and others). As a group, approaches using such operators for modelling cognitive representations are often referred to as Vector Symbolic Architectures (VSAs). In short, the SPA adopts and slightly modifies the VSA that uses circular convolution for the purposes of structured cognitive representation. However, it also greatly expands the intended target of VSAs, providing characterizations of perceptual and motor representations, cognitive action selection, and so on.

SPA is based on the ''semantic pointer hypothesis" which suggests that higher level cognitive functions in biological systems are made possible by Semantic Pointers (SPs). SPs are neural representations that carry partial semantic content and are composable into the representational structures necessary to support complex cognition. We have argued elsewhere that the SPA combines scalability, integration of perception, cognition, and action, and biological plausibility in a more convincing manner than past approaches (Eliasmith, 2013). The Spaun model is a testament to these claims (Eliasmith et al., 2012).

In Nengo, the SPA module is used to create networks by referring directly to SPs, while abstracting away from the specifics of the NEF (although NEF details can be adjusted later). In the cases we will consider here, this abstraction allows the cognitive model designer to focus on symbol-like information processing, while the overall model remains implemented in a spiking neural network.



```python
import nengo

model = nengo.Network()
with model:
    stim_a = nengo.Node(0)
    a = nengo.Ensemble(n_neurons=50, dimensions=1)
    nengo.Connection(stim_a, a)

    b = nengo.Ensemble(n_neurons=50, dimensions=1)

    # Connect the population to itself using a
    # long time constant (tau) for stability
    tau = 0.1
    nengo.Connection(b, b, synapse=tau)

    # Connect the input using the same time
    # constant as on the recurrent connection
    nengo.Connection(a, b, transform=tau)

```

Fig. 4 A Nengo network capable of storing information over time.

Staying faithful to a neural substrate (e.g., by building models using spiking neurons with appropriate synaptic time constants between connections) beneficially applies biological constraints to the kind of computations that can be performed effectively in such models. Clearly, such models are an abstraction since there are typically many missing details about the brain regions being captured by the model. However, the additional details which must be added by the modeller are often fertile ground for determining specific predictions that come out of the proposed model Eliasmith and Trujillo (2014).

Specifically, in larger models outside of the scope of this tutorial, the neural substrate can manifest itself by generating experimental firing patterns (Stewart, Choo, & Eliasmith, 2010b), physiological predictions (Stewart et al., 2010b), behavioural predictions (Eliasmith, 2013), allowing ablation experiments (Leigh, Danckert, & Eliasmith, 2014), and other biological manipulations (Rasmussen, 2010), while generating a variety of physiological data, such as EEG, LFP, and fMRI data (Stewart & Eliasmith, 2011).

4. Question answering model

In our first example demonstrating the SPA, we create a model that is able to examine a simple visual scene represented with SPs and answer questions, formulated as other SPs, about this scene. For example, imagine a person being asked questions about a scene containing a blue square and red triangle. If we ask: what colour is the square? Our model, like a person, should answer blue. Note that this example solves a simple instance of the so-called 'binding problem' (Jackendoff, 2002).

What structure should we use to represent this visual scene, assuming that the shape and colour vectors arrive as separate vectors from our visual system? There are two different compression operators defined for symbol-like representation in the SPA, vector addition and circular convolution. Given that the SPA defines the components of our visual scene as vectors (in all caps below), we could represent the scene with addition.

$$BLUE + SQUARE + RED + TRIANGLE$$

However, this representation suffers from the binding-problem: it does not distinguish between ''blue square and red triangle'' and ''red square and blue triangle''. We need a new binding operator that associates two vectors, by producing a third vector that is very dissimilar to the original inputs (as opposed to vector addition which produces an output that is highly similar to the inputs). The SPA uses the aforementioned circular convolution to fulfill this role (Plate, 1991).

Denoting circular convolution as $\circledast$, we can represent the visual scene S as follows:

$$S = BLUE \circledast SQUARE + RED \circledast TRIANGLE$$

It is also important to be able to decompress these representations to determine the elements of the original structure. For example, given the sentence S, we need to be able to identify what the attribute of the $SQUARE$ is. The SPA uses circular correlation (i.e., circular convolution with an inverse of the desired vector) for this operation. Thus the colour of the $SQUARE$ can be extracted from the sentence S as follows:

$$S \circledast SQUARE' \approx BLUE$$

where $'$ indicates the approximate inverse (here, involution, a linear operation).

Circular convolution also has the benefit of being an instance of a linear transformation followed by multiplication. Specifically, circular convolution in frequency domain is element-wise product of two vectors. Transforming to the frequency domain is a linear transform (e.g. the discrete

Fourier transform matrix). As described in Section 'Transformation', smooth nonlinear functions of this sort can be naturally and efficiently implemented in populations of neurons (also see the 09-multiplication.py example mentioned in Section 'Transformation'; see Plate (1991) for details).

Given we now know the representations to be used in our model, let's open a new blank file in the Nengo GUI and start building a model.

The results of the code that we walk through in the next three paragraphs are shown in Fig. 5. You can type the code into the text editor in the newly opened Nengo GUI browser window. We start by creating the required networks.

First, we need to create a vocabulary for the components to share. A vocabulary is a collection of SPs. By default, the entire model has a common vocabulary (unless you explicitly create separate vocabularies for different components). Since we want to quickly prototype our network and our vocabulary is limited, 16 dimensional vectors should be enough (Fig. 5, note A). Our vocabulary is created and populated in Fig. 5, note B. Vocabularies can (and often) grow as a model runs, so it is not necessary to define them ahead of time, but it can help others understand your code.

In Nengo, a SPA network can be created by using `model = spa.SPA()`, as shown in Fig. 5, note C. We have also provided this network with the vocabulary just created for initialization purposes, although this is not essential. This line creates a network which supports modules that use SPs and associated vocabularies in their definition.

Next, we require inputs for the question being asked, as well as the visual scene being represented. Since all inputs are assumed to arrive simultaneously, we do not require any memory or a basal-ganglia-thalamus loop for action selection, although we will require both in the next example in Section 'Instruction following model'. In this simple case, we only need circular convolution (denoted as `spa.Bind` in the code) and state networks for representing input and output. We assume that the visual input arrives as two distinct feature sets ''colour'' and ''shape'', which we need to bind together (Treisman & Schmidt, 1982). So, in total we need three state networks for the inputs (`colour_in`, `shape_in` and `question`), and one state network for the output (Fig. 5, note D). Additionally, we need two convolution networks: one to bind the colour and shape together (Fig. 5, note E); and one convolution network to answer the query (Fig. 5, note F).

The connections for this model are simple. The connection code and it's results are shown in Fig. 6. The colour and shape inputs are connected to the binding population (Fig. 6, note A). The bound value and the question are passed to the answering population (Fig. 6, note B), whose results are forwarded to the output (Fig. 6, note C).

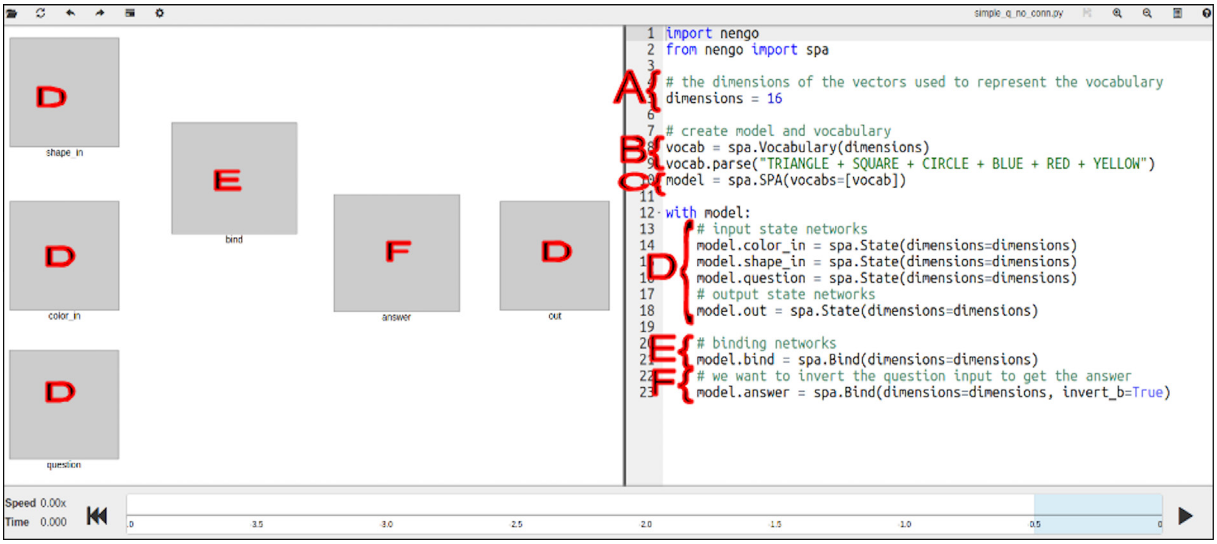
We can now simulate the model to check it is working correctly:

- Right-click on all state networks individually and select *Semantic Pointer Cloud.*
- Right-click on all the input-related SP Clouds to enter a SP that will result in that ensemble representing the specified SP. Set the shape value to TRIANGLE. Set both the colour and query value to RED.
- For the `bind` network, right click on the SP Cloud and select *Show Pairs* so we can see the binding results.
- Hit the play button in the bottom right corner of the window.

A successful test is shown in Fig. 7.

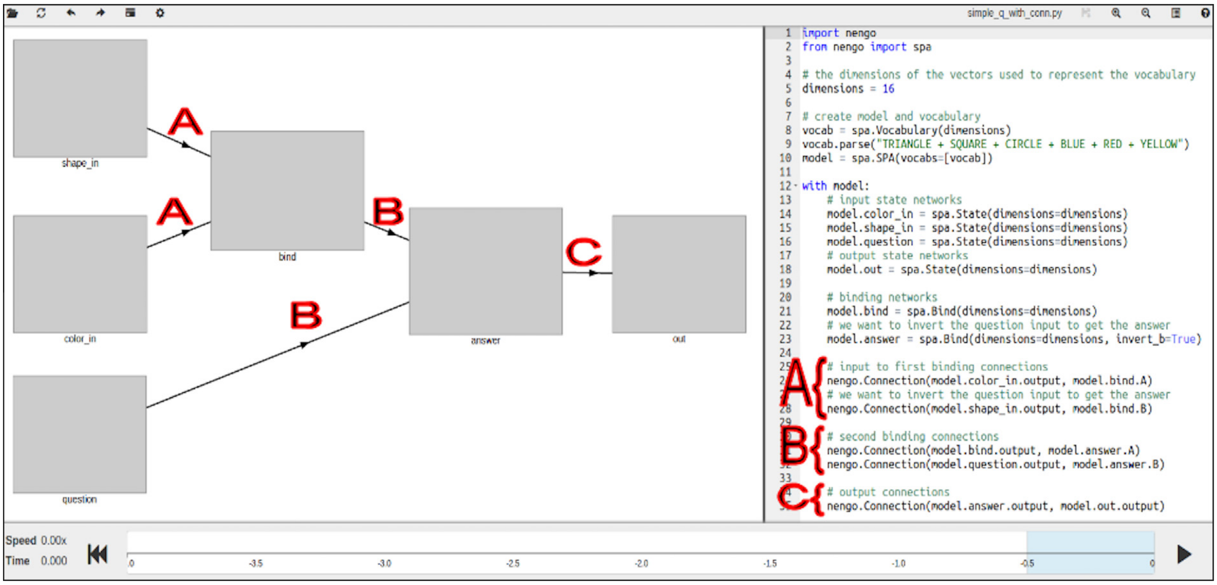
To make the validation of our model easier, we need to automatically provide input to the model, which we can accomplish by using the `spa.Input` object. This object associates state networks with input functions that change over time. To ensure that the model is responding to both colour and shape related queries, by getting both features from RED*SQUARE and RED*TRIANGLE. The code to be appended to your previous model code, is shown in Fig. 8, note A.

- Right click on each of the SP Cloud plots and choose *Remove* to remove them.



```
import nengo
from nengo import spa

# the dimensions of the vectors used to represent the vocabulary
dimensions = 16

# create model and vocabulary
vocab = spa.Vocabulary(dimensions)
vocab.parse("TRIANGLE + SQUARE + CIRCLE + BLUE + RED + YELLOW")
model = spa.SPA(vocabs=[vocab])

with model:
    # input state networks
    model.color_in = spa.State(dimensions=dimensions)
    model.shape_in = spa.State(dimensions=dimensions)
    model.question = spa.State(dimensions=dimensions)
    # output state networks
    model.out = spa.State(dimensions=dimensions)

    # binding networks
    model.bind = spa.Bind(dimensions=dimensions)
    # we want to invert the question input to get the answer
    model.answer = spa.Bind(dimensions=dimensions, invert_b=True)
```

Fig. 5 The code and resulting network graph. Note that the SPA module is imported in the line `from Nengo import spa`. The network elements have been manually re-arranged to enhance understanding. See text for explanation of lettered annotations.



```
import nengo
from nengo import spa

# the dimensions of the vectors used to represent the vocabulary
dimensions = 16

# create model and vocabulary
vocab = spa.Vocabulary(dimensions)
vocab.parse("TRIANGLE + SQUARE + CIRCLE + BLUE + RED + YELLOW")
model = spa.SPA(vocabs=[vocab])

with model:
    # input state networks
    model.color_in = spa.State(dimensions=dimensions)
    model.shape_in = spa.State(dimensions=dimensions)
    model.question = spa.State(dimensions=dimensions)
    # output state networks
    model.out = spa.State(dimensions=dimensions)

    # binding networks
    model.bind = spa.Bind(dimensions=dimensions)
    # we want to invert the question input to get the answer
    model.answer = spa.Bind(dimensions=dimensions, invert_b=True)

    # input to first binding connections
    nengo.Connection(model.color_in.output, model.bind.A)
    # we want to invert the question input to get the answer
    nengo.Connection(model.shape_in.output, model.bind.B)

    # second binding connections
    nengo.Connection(model.bind.output, model.answer.A)
    nengo.Connection(model.question.output, model.answer.B)

    # output connections
    nengo.Connection(model.answer.output, model.out.output)
```

Fig. 6 The network graph with connections. See text for explanation of lettered annotations.

- Replace the removed plots by right clicking on each network and choosing the *Semantic Pointer Plot* option from the menu.
- You can now run the model to see the result.

Fig. 9 shows the SP plots, which display the similarity (dot product) between the output vector and the original vectors defined in the vocabulary.

To display a full second of data, instead of the default 0.5 s:

- Pause the simulator after running the model for at least a simulation-time second by clicking the pause button in the bottom right corner.
- Drag the left edge of the transparent blue box (contained in the timeline at the bottom of the screen) until it reaches 1 s in width.

As shown in Fig. 9, the representations were bound and unbound as expected.

This completes the first SPA tutorial in this paper. While simple, this example has demonstrated how to construct a vocabulary of semantic pointers for a model, how to bind SPs into a syntactic structure, and how to unbind syntactic structures into their constituent SPs. All of this is done in a spiking neural network (to observe spikes in the model, double-click on network components until you see ensembles, and right-click on the ensembles to select the spike

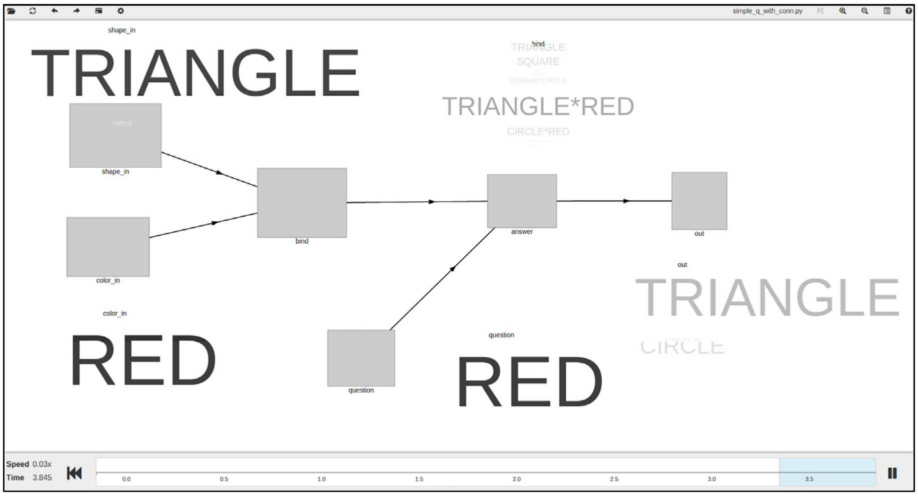
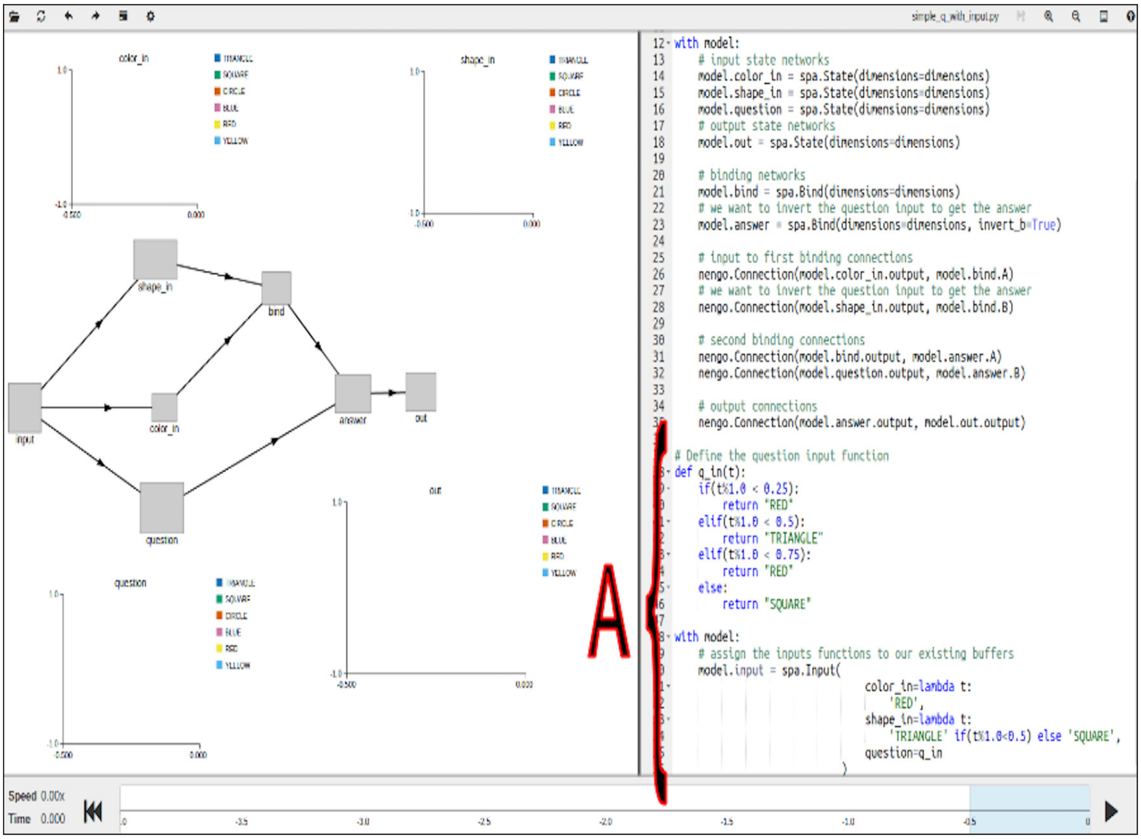


Fig. 7 Simulating the network, with a RED TRIANGLE as the visual scene, RED as the question and TRIANGLE as the answer. The code pane has been hidden to show the result better. Confidence of results are shown by the size and opaqueness of the text. Note that the answer is paler than the question components. This is because of the lossiness of decompressing a SP.



```
with model:
    # input state networks
    model.color_in = spa.State(dimensions=dimensions)
    model.shape_in = spa.State(dimensions=dimensions)
    model.question = spa.State(dimensions=dimensions)
    # output state networks
    model.out = spa.State(dimensions=dimensions)

    # binding networks
    model.bind = spa.Bind(dimensions=dimensions)
    # we want to invert the question input to get the answer
    model.answer = spa.Bind(dimensions=dimensions, invert_b=True)

    # input to first binding connections
    nengo.Connection(model.color_in.output, model.bind.A)
    # we want to invert the question input to get the answer
    nengo.Connection(model.shape_in.output, model.bind.B)

    # second binding connections
    nengo.Connection(model.bind.output, model.answer.A)
    nengo.Connection(model.question.output, model.answer.B)

    # output connections
    nengo.Connection(model.answer.output, model.out.output)

# Define the question input function
def q_in(t):
    if(t%1.0 < 0.25):
        return "RED"
    elif(t%1.0 < 0.5):
        return "TRIANGLE"
    elif(t%1.0 < 0.75):
        return "RED"
    else:
        return "SQUARE"

with model:
    # assign the inputs functions to our existing buffers
    model.input = spa.Input(
                          color_in=lambda t:
                              'RED',
                          shape_in=lambda t:
                              'TRIANGLE' if(t%1.0<0.5) else 'SQUARE',
                          question=q_in
                    )
```

Fig. 8 The code to provide automated input to the model and the new plots. See text for explanation of lettered annotations.

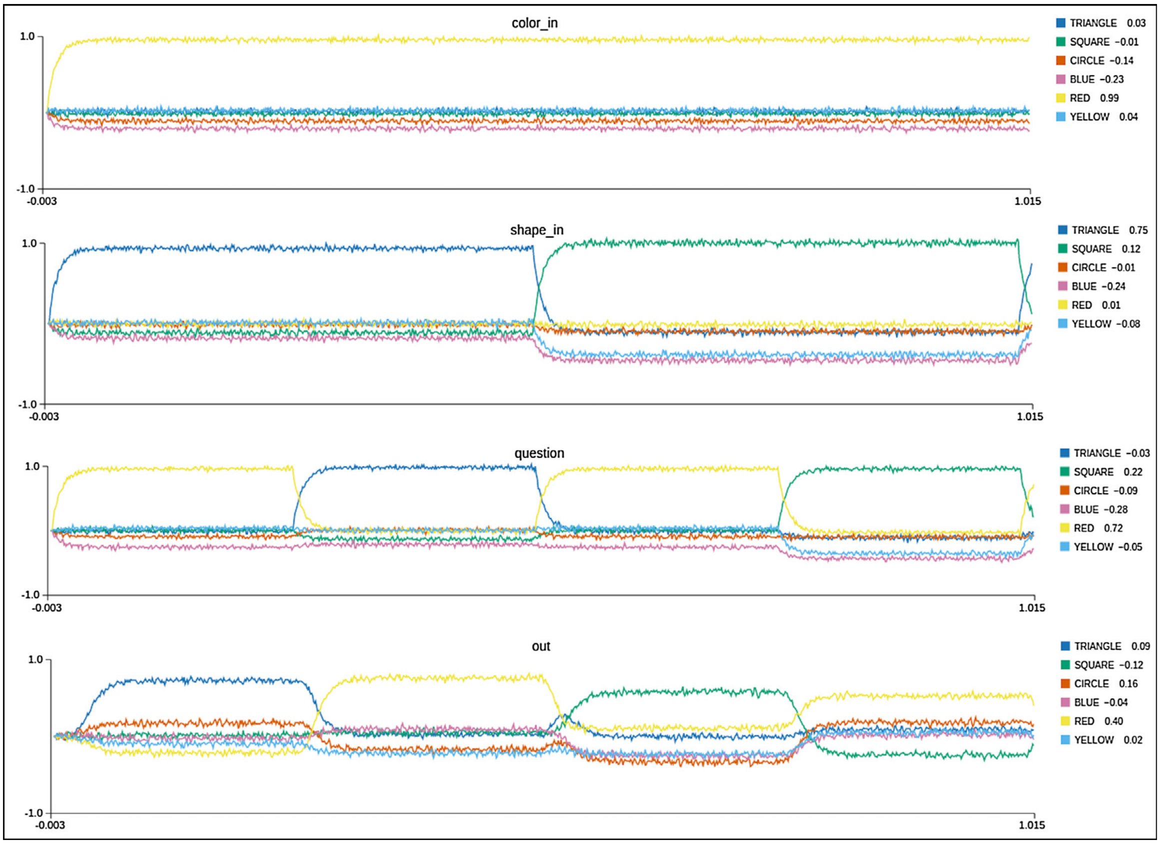


Fig. 9 The similarity plots after the model is run for one simulation-time second. The plots have been re-arranged for legibility.

graphs). These basic operations will be expanded on and integrated with new operations in the next tutorial example.

5. Instruction following model

In this section we describe a more complex example, that is capable of processing simple language commands and then executing actions based on these commands. In this model, the cortex-basal ganglia-thalamus loop is used to control information flow through the model, much as it is thought to in mammalian cortex (Redgrave, Prescott, & Gurney, 1999). Specifically, in this model information stored in cortex is used by the basal ganglia as the basis for selecting between a set of actions. When an action is selected by the basal ganglia, it causes the thalamus to modify the transmission of information between areas of the cortex (Stewart, Choo, & Eliasmith, 2010a).

5.1. Model description

The function of our model is to obey sequential language commands arriving as verb—noun pairs, by executing the appropriate actions. The model receives language commands via visual input and possible outputs are speech or writing. For this tutorial, we will use a very limited vocabulary: The verb is either SAY or WRITE; and the noun is either HELLO or GOODBYE. For example, ''SAY HELLO" is a verb—noun pair that is supplied to the visual input of the model one word at a time. This is followed by a period of no input during which the model is expected to execute this command. Based on the visual input and the state of the cortex, there are four actions (two internal and two physical) that the model can perform.

Basal ganglia selects one of these four actions and the thalamus routes them as shown in Fig. 10. We describe these four actions next.

Internal actions: Internal actions modify the memory contents of the cortex and do not cause any external behaviour. They are executed when the visual input is either a noun or a verb as follows:

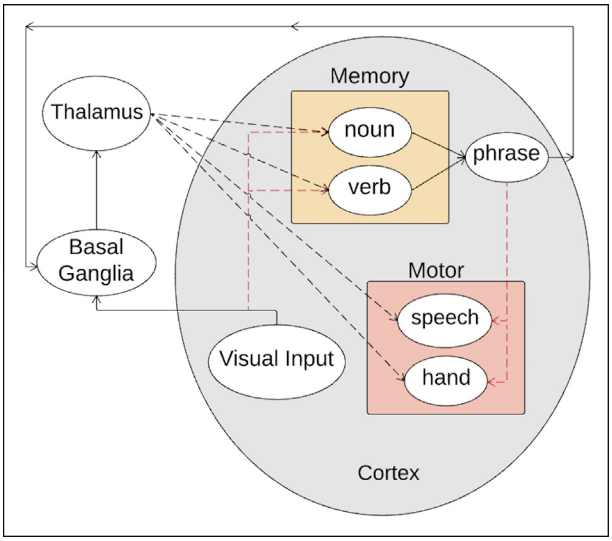


Fig. 10 Action selection by the basal ganglia and routing by the thalamus.

1. If the visual input is a verb, the model identifies and stores the verb in the *verb* memory.
2. If the visual input is a noun, the model identifies and stores the noun in the *noun* memory.

Physical actions: Physical actions are executed when the visual input is neither a noun nor a verb (this is realized by setting the visual input to NONE). The chosen action is dependent on the current state of the cortex, which is determined by the most recent language command supplied to the model. Physical actions affect the motor areas of the cortex, and cause the model to exhibit external behaviour through either speaking or writing as follows:

1. If the verb was SAY, the *speech* network outputs the noun in the command.
2. If the verb was WRITE, the *hand* network outputs the noun in the command.

Once the basal ganglia selects one of the four actions, based on the visual input, it sends the result to the thalamus. The thalamus then routes the information in cortex (in this case, the phrase or the visual input) to implement the effects of the selected action. The black dotted lines in Fig. 10 show the routing connections from thalamus to cortex. Consequently, some information is either sent to a memory area or to a motor area in the cortex, leading to information flow between cortical areas as shown by the red dotted lines in Fig. 10. Note that the *phrase* network in cortex is connected back to the basal ganglia, thus completing the cortex-basal-ganglia-thalamus loop. The phrase network provides information that helps determine action selection and routing.

Provided this description of the function of the model, we can now build it using Nengo. We use the following SPA modules:

- `State` — Represents a semantic pointer with optional recurrent connection. It is used for representing and transmitting information. It acts as a simple working memory if the feedback parameter is set to 1.0, while other non-zero values create a more rapidly decaying or saturating recurrent memory. By setting the feedback to 1, Nengo will introduce recurrent connections into the population of neurons. These connections can then act to preserve the activity of the population over time, providing a kind of memory. This is demonstrated further in Section 'Dynamics'.
- `BasalGanglia` — Performs action selection on a set of given actions.
- `Thalamus` — Implements the effects of action selection for an associated basal ganglia.
- `Cortical` — Used to form cortical connections between other SPA modules.
- `Actions` — A collection of Actions which are commonly used in two ways: 1. specified as actions to choose between for the basal ganglia 2. connections to implement for the cortical modules that cause specific actions (i.e., computations) to occur in cortex.

We will now proceed to use these SPA modules to build the components of our model. After building the individual components, we will put them together to implement our full model (the final code is shown in Fig. 15). By using the methods for setting inputs and visualizing outputs presented in Section 'Question answering model', each code block can be tested separately.

As mentioned, the model receives a visual language command as an input (Fig. 11, Note X), which the basal ganglia uses to select one of the actions. This implies that we need a State component to represent the visual input (`visual`; see Fig. 13). We also need a BasalGanglia (`bg`) with a list of specified actions and a Thalamus (`thal`) to implement the effects of those actions. When the input to the model is either a verb or a noun, the input should be stored in the memory area of the cortex (Fig. 11, Note A). Thus we need two State components with `feedback = 1` for storing the noun (`noun`) and verb (`verb`). Splitting the language command into a noun and a verb enables us to create a structured representation of the phrase using binding, as explained in Section 'Question answering model'. This form of representation makes it easier to later decompress the phrase for extracting the noun or verb (an example of this is shown later).

In the code shown in Fig. 12, the State components that have `feedback = 1` (line 2, 3) act as a working memory. The Actions module takes a string definition of an action where '−−>' is used to split the action into condition and effect (line 6, 7), otherwise it is treated as having no condition and just an effect. The conditions are processed by the basal ganglia to select the action with the maximum utility (i.e., the action having the maximum value of the dot product in its condition). The effects are used by the thalamus to control routing within the cortex. In this example (line 6), the basal ganglia checks to see whether the `visual` representation is similar to either of the verbs, SAY or WRITE. If it is similar to either of them, the thalamus routs the visual representation to the `verb` memory network.

Similarly, if the input is similar to one of the nouns, the visual representation is routed to the `noun` memory network (line 7). The list of these actions is passed to the basal ganglia which performs action selection on these actions (line 9). The basal ganglia (`bg`) is passed to the thalamus (`thal`) which implements the effects for the basal ganglia (line 10).

In the code block shown in Fig. 13, we construct a State component that represents a `phrase` formed out of the noun and verb representations. The `noun` and `verb` networks need to be connected to the `phrase` network so

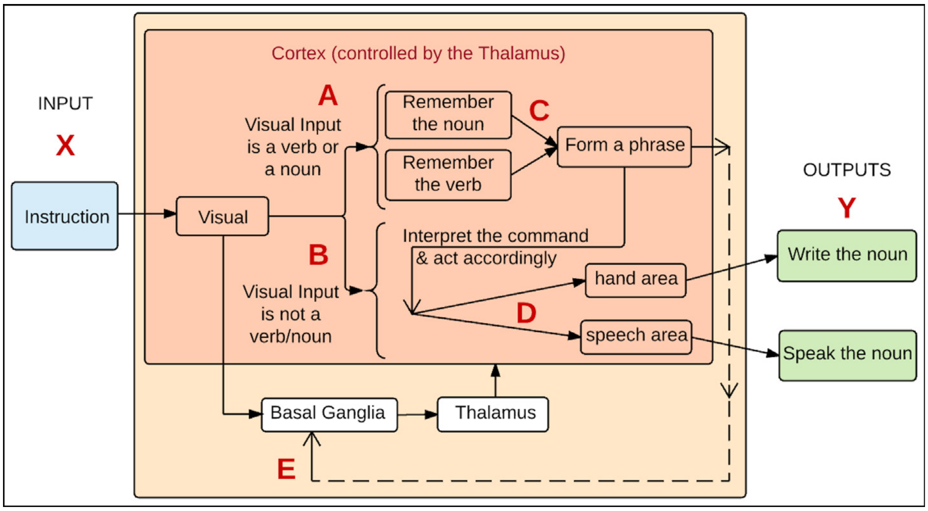
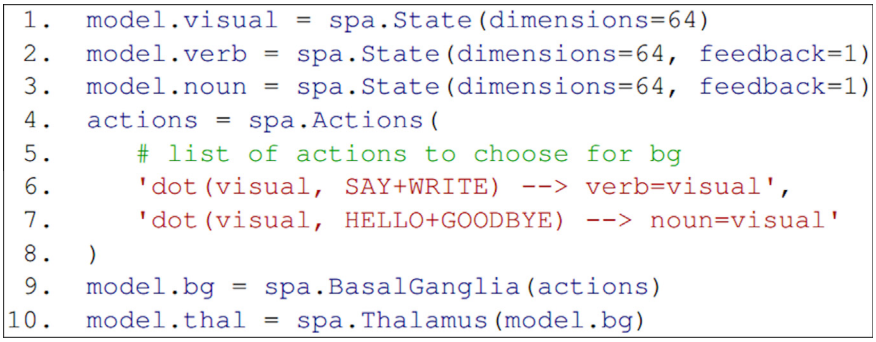


Fig. 11 Block diagram showing the input, outputs and key components of the model. See text for explanation of lettered annotations.



```
 1.  model.visual = spa.State(dimensions=64)
 2.  model.verb = spa.State(dimensions=64, feedback=1)
 3.  model.noun = spa.State(dimensions=64, feedback=1)
 4.  actions = spa.Actions(
 5.      # list of actions to choose for bg
 6.      'dot(visual, SAY+WRITE) --> verb=visual',
 7.      'dot(visual, HELLO+GOODBYE) --> noun=visual'
 8.  )
 9.  model.bg = spa.BasalGanglia(actions)
10.  model.thal = spa.Thalamus(model.bg)
```

Fig. 12 Code for building state components (`visual`, `verb` and `noun`), BasalGanglia (`bg`) and Thalamus (`thal`).

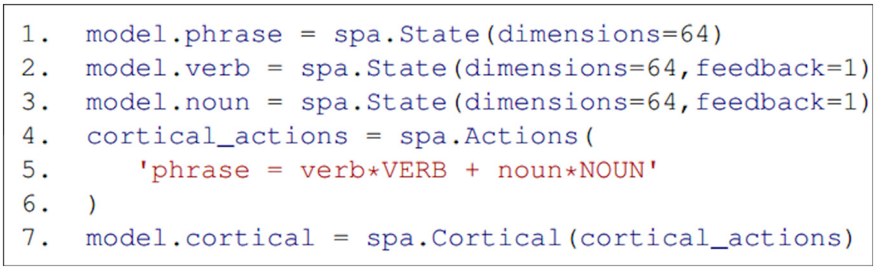
that the full structure of the command is represented (Fig. 11, Note C). We can build these connections using the `cortical`. These cortical connections are the same as standard Nengo connections, although they are simpler to construct. In the code in Fig. 13, cortical actions are passed to the Cortical module to form connections between the state components (line 7). Cortical actions (lines 4–6) are connections used for directing the flow of information within the cortex. In this case, the contents of `verb` are bound with constant SP *VERB*, and the contents of `noun` are bound with constant SP *NOUN*. These bound representations are then added and passed to the `phrase` network (line 5). This creates a structured representation of the phrase, and clarifies the need to store the noun and verb in separate State components. Thus, multiple operations including binding, addition and forming connections are all done in a single command using the Cortical module. Note that the cortical actions cannot have conditions (line 5).

Recall that when the input to the model is not a verb or a noun (Fig. 11, Note B), the command in `phrase` should be interpreted to determine whether the model was instructed to SAY or to WRITE the noun. Depending on the instruction, the noun from `phrase` should be sent to either the speech or the hand area of the cortex, causing a physical action (Fig. 11, Note D). Thus we need two more State components for representing the speaking (`speech`) and writing (`hand`) outputs. The code in Fig. 14 provides an example of a basal ganglia rule for executing these physical actions (lines 6–8). In this example, the basal ganglia checks whether the instruction was to SAY, by computing the dot product between the contents of `phrase` and the semantic pointer SAY*VERB (line 6). Recall that the basal ganglia has access to the contents of `phrase` through a feedback connection from the cortex (Fig. 11, Note E). The basal ganglia also ensures that the `visual` network is not currently receiving any input that needs processing (line 7). If these conditions are met, the thalamus unbinds the noun from `phrase` and passes it to the `speech` network (line 8). The symbol '~' is used to compute the inverse of *NOUN* for the de-convolution or unbinding (line 8).

Note the use of the constant SP *NOUN* in facilitating the extraction of the noun from `phrase` (line 8). A similar rule can be used to check for the WRITE command and direct it to the `hand` network. The `speech` or `hand` network then outputs the noun passed to it, simulating a physical action (Fig. 11, Note Y).

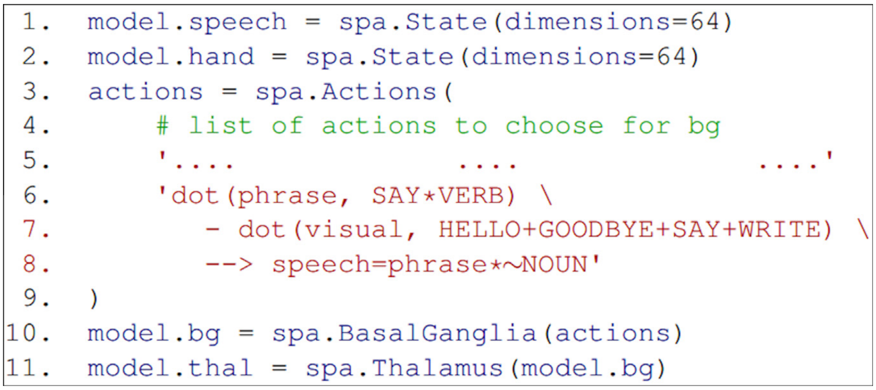
We can now combine these code blocks into a single model, removing any redundancies that were introduced for the purposes of testing (e.g. have multiple basal ganglia). Fig. 15 shows the full model implementation in Nengo using SPA. To build and simulate the model using Nengo GUI, follow these steps:

- Type the model implementation (Fig. 15) into the text editor. Fig. 16 shows the network connections of the model which appear in the network visualizer.
- Provide input to the model. Recall from Section 'Question answering model', that you can either directly provide



```
1.  model.phrase = spa.State(dimensions=64)
2.  model.verb = spa.State(dimensions=64,feedback=1)
3.  model.noun = spa.State(dimensions=64,feedback=1)
4.  cortical_actions = spa.Actions(
5.      'phrase = verb*VERB + noun*NOUN'
6.  )
7.  model.cortical = spa.Cortical(cortical_actions)
```

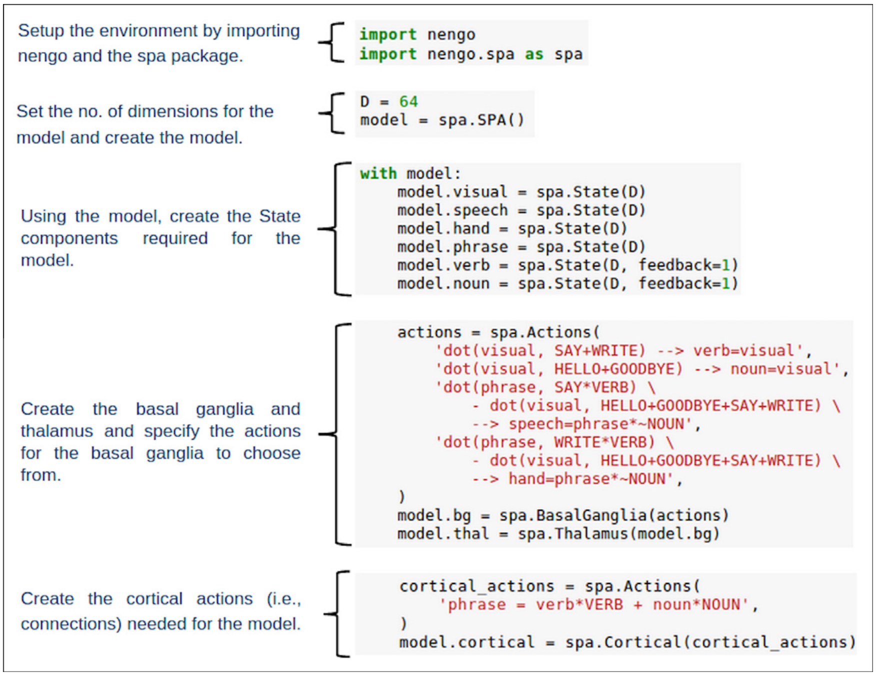
Fig. 13 Code for building state components (`phrase`, `verb` and `noun`) and the cortical connections.



```
1.  model.speech = spa.State(dimensions=64)
2.  model.hand = spa.State(dimensions=64)
3.  actions = spa.Actions(
4.      # list of actions to choose for bg
5.      '....            ....            ....'
6.      'dot(phrase, SAY*VERB) \
7.         - dot(visual, HELLO+GOODBYE+SAY+WRITE) \
8.         --> speech=phrase*~NOUN'
9.  )
10. model.bg = spa.BasalGanglia(actions)
11. model.thal = spa.Thalamus(model.bg)
```

Fig. 14 Code for building state components (`speech` and `hand`), BasalGanglia (`bg`) and Thalamus (`thal`).

Please cite this article in press as: Sharma, S et al., Large-scale cognitive model design using the Nengo neural simulator ..., *Biologically Inspired Cognitive Architectures* (2016), http://dx.doi.org/10.1016/j.bica.2016.05.001



Setup the environment by importing nengo and the spa package.

```
import nengo
import nengo.spa as spa
```

Set the no. of dimensions for the model and create the model.

```
D = 64
model = spa.SPA()
```

Using the model, create the State components required for the model.

```
with model:
    model.visual = spa.State(D)
    model.speech = spa.State(D)
    model.hand = spa.State(D)
    model.phrase = spa.State(D)
    model.verb = spa.State(D, feedback=1)
    model.noun = spa.State(D, feedback=1)
```

Create the basal ganglia and thalamus and specify the actions for the basal ganglia to choose from.

```
    actions = spa.Actions(
        'dot(visual, SAY+WRITE) --> verb=visual',
        'dot(visual, HELLO+GOODBYE) --> noun=visual',
        'dot(phrase, SAY*VERB) \
            - dot(visual, HELLO+GOODBYE+SAY+WRITE) \
            --> speech=phrase*~NOUN',
        'dot(phrase, WRITE*VERB) \
            - dot(visual, HELLO+GOODBYE+SAY+WRITE) \
            --> hand=phrase*~NOUN',
    )
    model.bg = spa.BasalGanglia(actions)
    model.thal = spa.Thalamus(model.bg)
```

Create the cortical actions (i.e., connections) needed for the model.

```
    cortical_actions = spa.Actions(
        'phrase = verb*VERB + noun*NOUN',
    )
    model.cortical = spa.Cortical(cortical_actions)
```

Fig. 15 Python code to construct the complete model using Nengo and SPA.

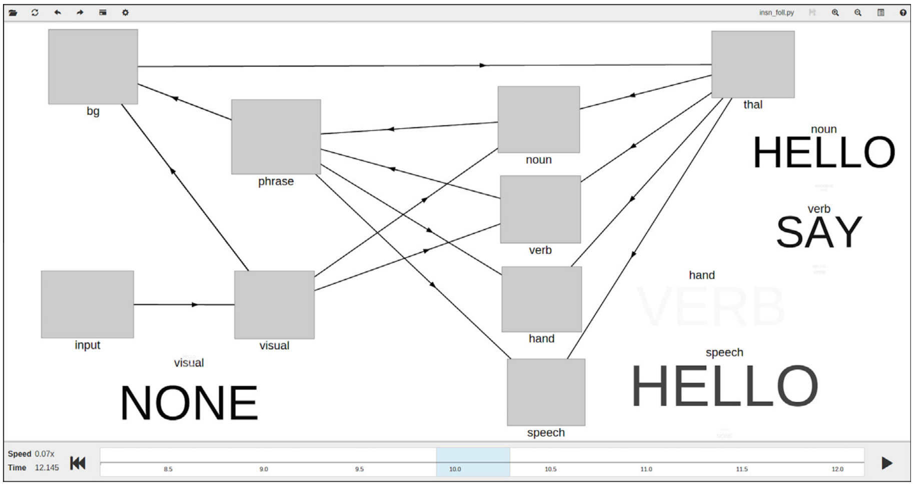


Fig. 16 Network layout and connectivity of the model. The *Semantic pointer Clouds* are shown from the time when the model is instructed to SAY HELLO followed by a period of NONE input, during which the speech State outputs HELLO.

input to the model via the SP Cloud plot, or provide an automatic input using a `spa.Input` object as shown below:

```
def input_vision(t):
    sequence = ‘SAY HELLO NONE WRITE GOODBYE \
                NONE GOODBYE SAY NONE’.split()
    index = int(t / 0.5) % len(sequence)
    return sequence[index]

with model:
    model.input = spa.Input
(visual = input_vision)
```

The above code provides input to the model which changes after every 0.5 s (refer to Table 1 for details). Additionally, the input sequence cycles repeatedly over time (i.e., the same input sequence is repeated after every 4.5 s). This code can be added to the text editor immediately after the code in Fig. 15.

- Display the similarity plots of the `visual`, `speech`, `hand`, `verb` and `noun` networks in the model, by choosing *Semantic pointer plot* option from their right-click menu. These plots (shown in Fig. 17) show the similarity of the contents of a particular component in the model to all the semantic pointers in the model’s vocabulary while the model is running.
- Run the simulation by pressing the play button.

Table 1 Model behaviour over time when the model is run for 4.5 simulation-time seconds.

Simulation time	Visual input	Model behaviour
$0 < t \leqslant 0.5$	SAY	Action: *verb* = *visual* BG determines that the input is a verb and thalamus routes it to `verb` State
$0.5 < t \leqslant 1$	HELLO	Action: *noun* = *visual* BG determines that the input is a noun and thalamus routes it to `noun` State
$1 < t \leqslant 1.5$	NONE	Action: $speech = phrase * \sim NOUN$ BG determines that the model has been instructed to speak HELLO, so `speech` outputs HELLO
$1.5 < t \leqslant 2$	WRITE	Action: *verb* = *visual* BG determines that the input is a verb and thalamus routes it to `verb` State
$2 < t \leqslant 2.5$	GOODBYE	Action: *noun* = *visual* BG determines that the input is a noun and thalamus routes it to `noun` State
$2.5 < t \leqslant 3$	NONE	Action: $hand = phrase * \sim NOUN$ BG determines that the model has been instructed to write GOODBYE, so `hand` outputs GOODBYE
$3 < t \leqslant 3.5$	GOODBYE	Action: *noun* = *visual* BG determines that the input is a noun and thalamus routes it to `noun` State
$3.5 < t \leqslant 4$	SAY	Action: *verb* = *visual* BG determines that the input is a verb and thalamus routes it to `verb` State
$4 < t \leqslant 4.5$	NONE	Action: $speech = phrase * \sim NOUN$ BG determines that the model has been instructed to say GOODBYE, so `speech` outputs GOODBYE

5.2. Results

The first 4.5 s of the model behaviour is shown in Table 1 along with the visual inputs. It is clear that the model is behaving as expected. The model ''says hello'' and then ''writes goodbye'' for the first two sets of verb—noun pairs presented to it. However it is worth noting that for the third set of verb—noun pair, the noun was presented to the model before the verb. In other words, instead of providing the command ''SAY GOODBYE'', command is provided as ''GOODBYE SAY''. However the model is still able to interpret the command and perform the appropriate closest action it knows of, i.e., it ''says goodbye''.

The same results are shown using the similarity plots in Fig. 17. The plots indicate that the model successfully parses the language commands and executes the corresponding actions. For example, when the model receives the `visual` input SAY (Fig. 17, Note A), the action *verb* = *visual* is executed, and the verb plot shows that verb network is now storing SAY (Fig. 17, Note B). Similarly, when the model receives the input HELLO (Fig. 17, Note C), the action *noun* = *visual* is executed and the noun plot shows that noun network is storing HELLO (Fig. 17, Note D). When the model receives the input NONE (Fig. 17, Note E), the action $speech = phrase * \sim NOUN$ (i.e., unbind the noun from the phrase and speak it) is executed and the speech plot shows HELLO (Fig. 17, Note F), indicating that the model ''says hello''.

This concludes our second example of constructing a simple cognitive model using the SPA in Nengo. While both examples are highly simplified, they have allowed us to demonstrate several of the features of the SPA module. One especially promising aspect of these features is that they have proven to be highly scalable, allowing the representation of the structured relations in the more than 100,000 concepts in WordNet (Crawford, Gingerich, & Eliasmith, 2015), and allowing many tasks to be executed by a single model, as demonstrated by Spaun (Eliasmith et al., 2012).

6. Features of Nengo/SPA

This tutorial has scraped the surface of all that Nengo and the SPA are capable of. In particular, the Nengo GUI was exclusively used to interface with Nengo, however it is possible to use Nengo from the command line and in iPython notebooks (Pérez & Granger, 2007) to run repeated and longer term experiments for empirical validation (timing, spiking data, etc.). This is discussed in greater detail in the Nengo documentation (https://pythonhosted.org/nengo/), as well as in the ''Best Practices'' repository (https://github.com/ctn-waterloo/best-practices) where we are currently in the process of establishing standards for this type of model use.

There were also many components of the SPA that were not used in this tutorial, such as Associative Memories and networks that output similarity between SPs. More importantly, the connections in this tutorial were all static. However, Nengo gives the ability to learn associations (Voelker, Crawford, & Eliasmith, 2014) and other functions (Bekolay & Eliasmith, 2011) on the connection weights between modules using BCM, Oja and Prescribed Error Sensitivity learning rules. Additionally, most of the computation in these examples focused on manipulations of symbolic vectors, but neural ensembles are also capable of representing function spaces. This includes probability distributions, such as

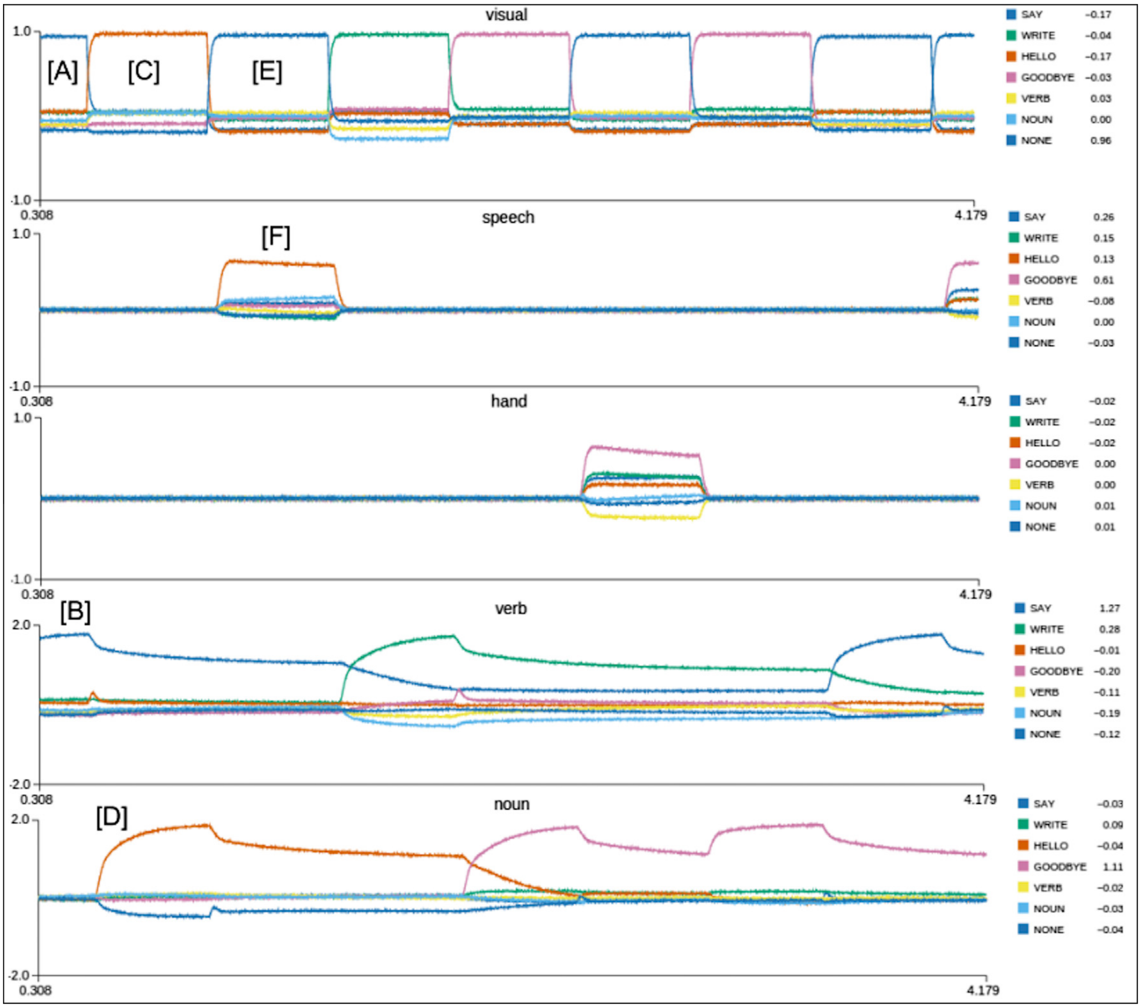


Fig. 17 Simulation results showing the SPA similarity plots after the model is run for 4.5 simulation-time seconds.

Gaussians, exponentials or any other function space that is capable of being represented by vectors.

In this tutorial, all simulations were run on your computer's CPU, but Nengo supports a variety of different backend hardware including GPUs via OpenCL (Bekolay et al., 2014) and neuromorphic hardware such as SpiNNaker (Mundy, Knight, Stewart, & Furber, 2015) and Neurogrid (Choudhary et al., 2012). These backends allow for much faster simulation, and often with considerably lower power consumption. In addition to these functional features, Nengo also includes more detailed neuron models than the LIFs and more detailed synaptic models than are used in this tutorial. Recently, Nengo has been connected to Neuron (i.e., a simulator that supports extremely detailed single cell models) allowing for models with even greater biological realism and more complex dynamics (Eliasmith, Gosmann, & Choo, 2016).

7. Conclusion

In this tutorial, we demonstrated building two Nengo models: a simple question answering model; and a slightly more complex instruction following model. Each of these leveraged elements of the SPA to quickly prototype simple cognitive behaviours (i.e., binding and rule following). We believe that, as demonstrated by models like Spaun, these tools can provide effective means of building large, biologically based cognitive models that capture a wide variety of neural and behavioural data. Moreover, we hope that these tools prove useful to others in the field for addressing interesting tasks and cognitive behaviours that we have not yet considered. To this end, we invite users to join the Nengo mailing list (http://www.nengo.ca/contact), and participate in a vibrant community devoted to improving both these tools and the models they are used to build. Additionally, some of the existing Nengo models can be found at http://models.nengo.ca/.

Acknowledgments

This work was supported by CFI and OIT infrastructure funding, the Canada Research Chairs program, NSERC Discovery Grant 261453, ONR Grant N000141310419, AFOSR Grant FA8655-13-1-3084 and OGS graduate funding.

Please cite this article in press as: Sharma, S et al., Large-scale cognitive model design using the Nengo neural simulator ..., *Biologically Inspired Cognitive Architectures* (2016), http://dx.doi.org/10.1016/j.bica.2016.05.001

References

Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T. C., Rasmussen, D., ... Eliasmith, C. (2013). Nengo: A Python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics, 7*.

Bekolay, T., Bergstra, J., Hunsberger, E., DeWolf, T., Stewart, T. C., Rasmussen, D., ... Eliasmith, C. (2014). Nengo: A Python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics, 7*.

Bekolay, T., & Eliasmith, C. (2011). A general error-modulated STDP learning rule applied to reinforcement learning in the basal ganglia. *Cognitive and Systems Neuroscience*.

Choudhary, S., Sloan, S., Fok, S., Neckar, A., Trautmann, E., Gao, P., ... Boahen, K. (2012). Silicon neurons that compute. *International conference on artificial neural networks* (vol. 7552, pp. 121–128).

Crawford, E., Gingerich, M., & Eliasmith, C. (2015). Biologically plausible, human-scale knowledge representation. *Cognitive Science, 40*(4), 782–821.

Eliasmith, C. (2013). *How to build a bra: A neural architecture for biological cognition*. Oxford University Press.

Eliasmith, C., & Anderson, C. H. (2004). *Neural engineering: Computation, representation, and dynamics in neurobiological systems*. MIT Press.

Eliasmith, C., Gosmann, J., & Choo, X. (2016). Biospaun: A large-scale behaving brain model with complex neurons. http://arxiv.org/abs/1602.05220.

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

Eliasmith, C., & Trujillo, O. (2014). The use and abuse of large-scale brain models. *Current Opinion in Neurobiology, 25*, 1–6.

Jackendoff, R. (2002). Foundations of language: Brain, meaning, grammar. *Evolution*.

Leigh, S., Danckert, J., & Eliasmith, C. (2014). Modelling the differential effects of prisms on perception and action in neglect. *Experimental Brain Research, 233*(3), 751–766.

Mundy, A., Knight, J., Stewart, T. C., & Furber, S. (2015). An efficient spinnaker implementation of the neural engineering framework. In *IJCNN*.

Pérez, F., & Granger, B. E. (2007). IPython: A system for interactive scientific computing. *Computing in Science & Engineering, 9*, 21–29.

Plate, T. (1991). Holographic reduced representations: Convolution algebra for compositional distributed representations. In *IJCAI* (pp. 30–35). Citeseer.

Rasmussen, D. (2010). *A neural modelling approach to investigating general intelligence* (Masters thesis).Waterloo, ON: University of Waterloo.

Redgrave, P., Prescott, T. J., & Gurney, K. (1999). The basal ganglia: A vertebrate solution to the selection problem? *Neuroscience, 89*(4), 1009–1023.

Stewart, T. C., Choo, X., & Eliasmith, C. (2010a). Symbolic reasoning in spiking neurons: A model of the cortex/basal ganglia/thalamus loop. In *Proceedings of the 32nd annual conference of the cognitive science society* (pp. 1100–1105). TX: Cognitive Science Society Austin.

Stewart, T. C., Choo, X., & Eliasmith, C. (2010b). Dynamic behaviour of a spiking model of action selection in the basal ganglia. In *Proceedings of the 10th international conference on cognitive modeling* (pp. 235–240). Citeseer.

Stewart, T. C., & Eliasmith, C. (2011). Neural cognitive modelling: A biologically constrained spiking neuron model of the Tower of Hanoi task. In *33rd annual conference of the cognitive science society*.

Treisman, A., & Schmidt, H. (1982). Illusory conjunctions in the perception of objects. *Cognitive Psychology, 14*, 107–141.

Voelker, A. R., Crawford, E., & Eliasmith, C. (2014). Learning large-scale heteroassociative memories in spiking neurons. In *Unconventional Computation and Natural Computation, London, Ontario*.