Neural BEAGLE

An Implementation of the BEAGLE Algorithm Using The Neural Engineering Framework

Eilene Tomkins-Flanagan

Queen's University

Abstract

The project “Neural BEAGLE” was concerned with the development and implementation of the BEAGLE algorithm (Jones & Mewhort, 2007) within the Semantic Pointer Architecture framework (Eliasmith, 2012). Preliminary experimentation and analysis yielded encouraging results for the future development of models attempting to bridge the psychological/neurobiological divide, however the intricacy and runaway computational complexity of such models may prove a limiting factor in their explorability in the future. This paper examines the theory underlying the interfaced model and the technical hurdles on the road to its implementation.

BEAGLE in Neurons

Jones & Mewhort (2007) first described the BEAGLE algorithm as a simple means of modelling the process of acquiring knowledge of semantic information pertaining to words. Though limited in its descriptive power, the BEAGLE algorithm has succeeded at generating vectors that may be used to capture a variety of semantic effects observed in human subjects (Johns & Jones, 2015; Mewhort & Shabahang, 2017). This project implements the BEAGLE algorithm using the Semantic Pointer Architecture (SPA) described by Eliasmith (2013). The software suite in which the code is implemented is Nengo, a neural simulator which provides libraries for the SPA designed by Eliasmith's own CNRG lab at the University of Waterloo (Version 2.8; 2018).

BEAGLE

BEAGLE is an algorithm that produces Holographic Reduced Representations (HRRs; Plate, 1995). HRRs in general are vectors of high dimensionality (for the purposes of BEAGLE, it is typically in the neighbourhood of 1000-2000) that are derived using a particular set of mathematical operations. The operations permitted on one or more HRRs are as follows: Addition (and subtraction), scalar multiplication (and division), circular convolution (and correlation), permutation, and the dot product and cosine operations.

Of the above operations, circular convolution and correlation are the least conventional, but they are also the most crucial. They are taken from signal processing, and are, respectively, the operation that describes a signal produced by the interaction of two others and the approximate inverse of that operation. Convolution distributes over addition, as does correlation, while convolution alone is commutative. Like addition, convolution and correlation produce a vector of the same dimensionality as each of their operands.

Convolution has a unique quality. If the cosine of two vectors describes how similar those vectors are (a cosine of 1 implying they are identical, and 0 implying they are orthogonal), then the

vector produced by a circular convolution will be dissimilar from either of its operands. Adding two vectors will, on the other hand, produce a vector similar to both. In this way, convolution can be said to "bind" or "associate" two vectors, while adding two vectors connotes their co-occurrence in a memory.

Plate gives the example of binding words with their grammatical function as a possible usage. For instance, taking * and # as our convolution and correlation operators, respectively:

$$PHRASE = SUBJECT * DOG + VERB * CHASE + OBJECT * MAN$$

The vector PHRASE unambiguously represents subject, verb, object structure. Because correlation distributes over addition, we can perform the operation:

$$SUBJECT \# PHRASE = \sim DOG$$

The vector we get is only *similar to* DOG for two reasons. (1) Correlation only yields the approximate inverse of a convolution, and (2) the output we get is the following superposition of terms:

$$SUBJECT \# PHRASE = SUBJECT \# (SUBJECT * DOG) + SUBJECT \# (VERB * CHASE) + [...]$$

Thus, correlation yields the sum of an approximation to our target and two vectors that represent nothing, henceforth called *noise*.

BEAGLE exploits the regularities of natural language and the capacity of HRRs (as opposed to matrices) to tolerate limitless addition of memory episodes without increasing in size. It generates two vectors for each word in a lexicon, respectively the sums of the general context in which a target word has appeared and of the sequence of words preceding and following the target word. They are dubbed "item" and "[associative] order" vectors, following after Murdock's (1982) theory of Storage and Retrieval of memory.

Unlike Murdock, BEAGLE doesn't want to retrieve any individual episodes from memory. Instead, the entire item or order vector is taken as the semantic value of the word it references; it is the *sum total of the word's history of use*. However, Murdock's distinction between item and associative information is useful in this instance. Item information represents the general context in which a word

appears. No particular context is distinguishable from any other, and you cannot tell, even in principle, whether any two words used to generate an item vector ever appeared in the same context. Order information, however, is encoded sequentially, by associating (via a binding operation) each word in the context with the next. This ordered representation doesn't just record the specific contexts in which a word appeared, but also how close or distant it was from the target.

BEAGLE operates, in general, on a Hebbian principle. Words that occur in similar contexts should have similar meanings. Though a simple principle, it is surprisingly powerful when combined with the tools arrayed above.

A run of BEAGLE begins with a lexicon; a dictionary available to you, the programmer, that tells you where the semantic representations for each word will be stored in memory. BEAGLE knows nothing about the sounds of words, nor their spellings, nor their feel in a virtual mouth, only the locations of its representations and the current values of their states (and of course, the memory episodes it will record). It may be the case that BEAGLE can be reasonably extended to include the sound or spelling of words in its semantic representation of them, however that has not yet been attempted.

For simplicity's sake, nothing is alleged by BEAGLE about the mechanism of storage of the vectors. In implementations, we use large matrices of dimensionality N×M, where N is the dimensionality of vectors we are using and M is the number of known words. Each word is mapped to one size-N row of the matrix. The model simply knows the current state of a target word when it needs to, updates that state, and stores it off wherever and however it happens to be stored. When presented with an episode, it simply knows what words are in it and the order in which they occur.

The second step of BEAGLE's operation is to generate visual vectors Viz_{word}. The properties of the visual vectors are crucial to the function of the model. Visual vectors are taken to be

the information the model “sees” for each word when an episode is presented to it. Therefore, in BEAGLE, each word actually has three vectors onto which it is mapped: Item, order, and visual.

The visual vectors are simply randomly oriented vectors of magnitude ~1. They are generated by selecting a random value from a Gaussian distribution on each dimension, with a mean of 0 and a standard deviation of $1/\sqrt{N}$. Because the expected value of a random number selected from a Gaussian distribution is $+/-\sigma$, and the magnitude of a vector $||v||$ is $\text{sum}(v_i^2; i=1..N)$, the expected magnitude of a visual vector is $\text{sum}(1/N; i=1..N) = N/N = 1$. Thus, we start with vectors that are all taken to have roughly equal significance, as, if a vector is identical to any vector with the same orientation, magnitude is just a weight.

The visual vectors are expected to be highly dissimilar from one another. Recalling that two vectors being dissimilar just means that they have a small cosine (or alternately, that they have a relative angle close to 90^o), the property is assured by the vectors’ random orientation and their high dimensionality. Consider the unit circle and the unit sphere. Selecting a 1-vector on an axis as your first vector x, and a randomly oriented 1-vector as your second vector y. On the unit circle, it can be observed that the proportion of the circle closer to 90^o than to either x or -x is 1/2. Therefore, y has a 50% chance of being more dissimilar to x or its negative than similar. On the unit sphere, the section closer to the orthogonal circle than to x or -x is a ring bounded by two circles, centred on $\sqrt{3} \times x$ and $-\sqrt{3} \times x$ and orthogonal to them. Four more more such circles can be drawn on the ring, without intersecting the other two, centred on vectors orthogonal to x and either orthogonal or opposite to one another, leaving area on the ring to spare (*Fig. 1*). By increasing the space by 1 dimension, the probability that a random vector of magnitude 1 will be more different from our chosen vector than similar more than doubled. The probability increases with the number of dimensions. Even if the size of the ring were to decrease (such that anything falling within it would be basically unrelated to x), the probability that any given random vector y would appear on the ring and not outside it would be

overwhelming at thousands of dimensions, tending towards guaranteed as the number of dimensions approaches infinity.

So with a collection of unrelated, unspecial vectors, BEAGLE calculates memory episodes from given phrases. BEAGLE takes in phrases of natural language (as sequences of visual vectors) and calculates item and order vectors like so:

(1) A stop list was declared for item vectors. This list contains all the known stop words (such as “the”, “and”, “to”). For each word in the phrase (including stop words), the item episode for the target word, Itm_{word}, is equal to the sum of visual vectors that occur in the phrase, omitting the target and stop words. So, for the phrase “the dog chased the mailman”:

$$Itm_{dog} = Viz_{chased} + Viz_{mailman}$$

(2) Order vectors are significantly more complicated. Prior to computation, a key vector, PHI, was generated. It is randomly oriented and has a magnitude of 1, but its role is to indicate the presence of a target word. Order vectors use circular convolution to associate words in sequence, but so that order matters in the sequence, convolution’s commutative property must be broken. Two matrices, P_L and P_R were generated. They are permutation matrices; in each row and each column there is exactly 1 randomly placed 1, such that the multiplication of the matrix with any vector will yield a vector with all of the same values, but in a scrambled order. Before convolution occurs, the left operand is permuted according to P_L, and the right according to P_R, performing what is called a “one-way convolution”, which will be denoted as *>. It is called “one way” because the permutation applied to each operand means that the ordering of operands will produce one of two different outputs that do not resemble one another. Any result can only be generated in one way. An order episode, Ord_{word}, is computed for each word by selecting windows of various sizes around a given target (target included), typically spanning all sizes up to 5 words, but in some implementations up to 7. Words inside the

window are one-way convolved in sequence, with the target replaced by PHI. All windows up to the maximum size that include the target are considered. So, for the phrase, “the dog chased the mailman”:

$$Ord_{dog} = Viz_{the} *> PHI + PHI *> Viz_{chased} +$$

$$Viz_{the} *> PHI *> Viz_{chased} + PHI *> Viz_{chased} *> Viz_{the} +$$

$$Viz_{the} *> PHI *> Viz_{chased} *> Viz_{the} + PHI *> Viz_{chased} *> Viz_{the} *> Viz_{mailman} +$$

$$Viz_{the} *> PHI *> Viz_{chased} *> Viz_{the} *> Viz_{mailman}$$

On each line of the above equation, all possible windows of size 2, 3, 4, and 5 are shown. PHI is used to replace the target vector so that the phrase “the dog chased the mailman” for the target “dog” looks identical to “the cat chased the mailman” for the target “cat”. Thus, words that appear in similar contexts resemble one another by virtue of having recorded the same or similar episodes. The use of the key PHI is because a convolution using a visual vector would not strongly resemble a convolution using PHI, and some vector must be used to indicate the position of the target so that its context can find positions relative to it. If we want the episode for “cat” to look like the episode for “dog”, so long as all the same surroundings are there, the same key vector must be used for both of them.

BEAGLE simply sums the item and order vectors over all episodes in order to get its final representations. While words are the primary subject of work using BEAGLE, the algorithm itself is more general. As the vectors it uses have no particular semiotics, and in fact, Murdock’s theory is completely general, variants of BEAGLE might be used to encode objects of relative position in space, or other perceptual information (including, due to the capacity of HRRs to recursively store information, perception of one’s own thoughts).

Semantic Pointer Architecture

The SPA is a tool that is broadly useful for the construction of HRRs. It is designed according to the principles of the Neural Engineering Framework (NEF; Eliasmith, 2012), which uses

control theory to interface populations of Leaky Integrate-and-Fire (LIF) neurons and floating point numbers, such that the floating point number represents the state of the population, and the connections between two populations A and B can cause the value B is representing to be the approximate result of a mathematical operation performed on the value A is representing. The floating point value is thus encoded by a neural population, and can be decoded from the state of the population using a particular operation.

The LIF neuron is a fairly simple, and importantly, biologically-motivated model of a neuron. While not all neurons in actual brains resemble LIF neurons, they model some well-understood types adequately (Usher et al., 1993). An LIF neuron behaves something like a bad capacitor. Incoming current causes a voltage to build up in the LIF neuron, until a threshold is reached. At the threshold, the neuron releases a spike, and its capacitance returns to 0. After a refractory period, it begins to accumulate charge again. Importantly, the neuron's membrane of some resistance is considered in the model, and should the neuron not fire, it will discharge through its membrane over time (*Fig. 2*).

Usher et al. first described the use of population coding in sets of LIF neurons. Their fully-connected networks exhibited the capacity to synchronize and hold a steady state. The NEF's insight was to take the state of such a neural population to represent (encode) a value, and notice that the code was invertible. That is, one could set the value encoded in a population's activity as easily as one could decode and read it. In neural networking, it is ordinary practice to encode a linear function in a set of weighted connections between groups of neurons. The NEF takes a linear approximation to any given function between two neural populations and encodes it in the weights between them such that given a function f(x) where x is the value encoded in a population A, the system comprised by A, a network B can be said to represent y = f(x), where y is the value encoded by B. As such, the NEF abstractly represents functions on scalar values in neural populations as groups of models of biological neurons and the connections between them, without the function or the scalar values ever explicitly influencing

the function of the network; they are only ever available as a decoding of the network's activity and connections.

The NEF has a key limitation: It does not provide a mechanism for learning in its connection weights. It alleges that, at the time a model is run, particular functions between populations of neurons are represented in the connections present between them. However, it does not propose an account of how those functions got there, nor how they can change after the model starts running. In this, it is extremely unconventional among contemporary neural networks, not considering weight adjustment after initialization at all. Although a sound scientific choice (the causes of myelination and synaptic pruning – two mechanisms said to mediate learning in connections – are not extraordinarily well-understood; Barres, 2008), the deficiency outright prohibits the use of memory models that rely on learning in the connections between neurons (because BEAGLE operates on Hebbian principles, a Hebbian learning rule might be used in an extension of this project, however that's out of scope for now). Any mechanism for recording or retrieving memory has to rely on the present state of neural populations and fixed operations between populations.

The SPA extends the NEF by implementing tools for the representation of HRRs in neural networks generated under the NEF. Linear algebra provides all the mathematics needed to develop an HRR, expressing any function on a vector as a set of functions on each of the values of the vector. Thus, the SPA can exploit the learning mechanism provided by HRRs: Holographic superposition; simply letting the memory be the sum total of recorded episodes. It is straightforward to implement the addition of two vectors using the NEF (just add their individual values) and the convolution of two vectors (as the circular convolution is just a periodic sum of an outer product matrix).

More complex is the implementation of control systems in the SPA. While Eliasmith has devised a very simple system that can switch tasks or recall short term memories according to instructions passed to it via pre-encoded environmental cues (he calls it SPAUN, short for Semantic

Pointer Architecture Unified Network; 2012), it is not at all obvious how an elaboration on SPAUN might intuit when it is time to attend to the next word in a phrase or conditionally overlook a word when recording an episode.

Integrating BEAGLE with a Neural Model

The components are, alone, elaborate and interesting. Their integration would be more elaborate still, but imagining it leaves open the question of whether the task would be as interesting. What motivates the embedding of BEAGLE into a neural model? The SPA specifically?

The appeal of BEAGLE to the SPA is relatively obvious, on inspection. If BEAGLE, or something very similar to it, can be feasibly implemented using the SPA, then the SPA's claim to biological plausibility can be bolstered with an account of the acquisition of semantic knowledge that has already been shown to be psychologically plausible. The result is (the beginnings of) a unified theory of psychology and neuroscience (even if it is outrageously speculative).

The appeal of the SPA to BEAGLE is less clear. BEAGLE was never intended to really represent the process for word learning, or any other learning process, for that matter. It generates vectors that seem to possess similar semantic properties to those used by humans, but it is not so bold as to make claims about an actual mechanism of learning that it expects to find in an organism. Extending BEAGLE by embedding it in the SPA, or any neural model, doesn't have much significance as far as the progress of research into the algorithm is concerned. Whether it is possible or not isn't at-issue.

The cognitive sciences are a relatively young and highly interdisciplinary family of fields. Researchers approach the study of cognition from many different directions (linguistics, neuroscience, mathematical psychology, and artificial intelligence, to name a few), but share little in terms of the questions they ask and what standards of evidence they rely upon for generating theory. Infamously, even the object of study is the topic of intense philosophical debate (to list some positions: Dreyfus,

2007; Haraway, 1985; Harnad, 1990; Newell, 1980; Searle, 1980; Sutton et al., 2010; Wilson, 2002). "The study of cognition" is insufficient, as while "cognition" grounds our science in apparent phenomena that can be measured (learning, language, and so on) it attributes those phenomena to an entity called a "mind". Where the mind exists in the physical universe is *usually* said to be the brain or the system comprised by the whole body, some cognitive scientists even attributing particular cognitive functions to brain regions. However, there is rarely a precise ontology for linking mind and brain.

This sloppy metaphysics is not the scientifically prudent disregard for the unfalsifiable, but rather the implicit adoption of non-rigourous metaphysical views. An artificial intelligence researcher can claim that the Physical Symbol System Hypothesis (Newell, 1980) has been categorically disproven (Nilsson, 2007), while a linguist might claim that a grammar (a kind of Physical Symbol System; Chomsky, 2006) is indisputably necessary for language production. Clearly, they use different standards of evidence, and may even be understanding the meaning of the mind's alleged implementation as a physical symbol system differently, but it is not obvious how.

A precise ontology allows the unambiguous and predictive specification of theory. One can ask the question "is my theory consistent with the ontology I am working in?" An ontology determines the meaning of data and the statistical interpretation thereof with respect to a descriptive model generated by the theory. The theory is situated in a universe that permits the alleged theory-statistics-data relationship. Without adopting the constraint of a known ontology, researchers devise theory according to their intuitions about the world, often without well-formed ideas about what is possible, how it is possible, how a given possible thing is meaningfully distinguishable from another, and what standards of evidence count for determining which possible thing is happening. The result, tragically, is fields generating work that is incompatible with and *doesn't care about its compatibility with* the findings of other fields. Theories between one family of cognitive sciences and another display sharp inconsistencies, theorists talk past one another in arguments, the definition of technical terms being

unstable (and sometimes not well-defined) between and even within fields, dogs and cats living together, mass hysteria.

A remedy is the adoption of a good set of ontological positions. Manuel De Landa's (2013) "speculative realism" suits what intuition says we are trying to accomplish in the cognitive sciences.

Putting the work of Gilles Deleuze into scientific terms, De Landa accepts the problem of induction (Popper & Miller, 1953) and adopts a speculative position. De Landa's position is somewhat weaker than Popper's, but because it allows itself to make weaker claims, it more precisely specifies the relationship between descriptive theory and actual physical entities in the world. Whereas Popper holds that the adoption of scientific theory is always *provisional* (implicitly, that a hypothetical master theory, predictive of all possible observations, is operating in a platonic capacity, unobservable and underneath everything), De Landa claims that the mathematical theories that characterize systems apparently actualized in the world are abstract or virtual expressions of those systems. For De Landa, the virtual is not in any sense more or less real than the actual (in computing terms, a floating point representation is no more or less real than a corresponding byte representation, or the set of transistors that correspond to the byte and their states, or the atoms that make them up, and so on, it is just the case that each of those entities can affect and be affected by other entities in different ways; e.g., the bytes can be logically operated on while a floating point number can be added). Rather, the virtual is *immanent* to the actual (literally a part of it, expressive of the same entity), but not directly present in a way one could observe (i.e., we cannot peer into the universe and find the Schrodinger equation written out in front of us; it had to be induced from observation).

De Landa's gift to the cognitive sciences is his analysis of abstraction. De Landa's virtual contains many valid abstractions of the same system, encompassing differently-relevant information. Each is a model in state space of the system with one or more attractor-states. The state of the system tends towards the attractor-state exerting the strongest influence over it, but never perfectly aligns with

the attractor. A sufficiently forceful displacement from an “external” (from the perspective of the given model) system can move the state towards another attractor. The model itself can be likewise displaced, undergoing a transition into a different model. In this sense, each model exists in a state space of its own, thus abstraction is nested recursively.

De Landa reconciles the problem of how an entity like the “mind” could correspond, in a fuzzy way, to the brain, body, and whole embodied system of an organism. The mind is not a singular, discrete entity, but rather statistically constituted (present as patterns in data about the world), and abstractly describable.

Immanence is thus the answer to BEAGLE’s uncaring attitude towards its presence in a real system. BEAGLE’s predictive power tells us that, in some sense, it is already there. The serious question is “in what sense?” It is almost certainly insufficient to describe learning in general (it is only a model of implicit learning and has no idea how the words it learns about came to be, after all), but it is nonetheless the case that it provides a description of semantic learning phenomena. How does a physical entity like a brain, or possibly a larger system, produce such a phenomenon? Knowing precisely *how* BEAGLE is descriptive, precisely *what* it describes, is deeply important to the furtherance of research into the algorithm and its family.

A place to start is attempting to implement BEAGLE (or as near as one can get) in the SPA. First showing that it is feasible, the model should then be further constrained by the known properties of the regions of brain tissue in which it is alleged to be implemented. The theory should predict as much existing data as possible, and further, allow for the prediction of as-yet unobserved phenomena. Through this process, the model should be revised as it is found to fail to generate predictions for real-world data. The model is taken to represent the process of word learning as-present in the brain, and both the abstract holographic model and the neural model which actualizes it should leave their scope

and properties open to revision as failures of correspondence between model and organism are discovered.

Feasibility

In the first stage of development, a report was written assessing Nengo's suitability for the task of embedding BEAGLE. Vectors in two sets were randomly initialized and artificially associated with one another. Pairs of vectors were selected from the sets and passed through both a Nengo implementation of key mathematical operations (convolution, one-way convolution, correlation, addition), and an implementation of the same in NumPy. The result of the report showed that with default settings, Nengo's ability to perform the selected key operations accurately is nearly perfect (*Fig. 3*).

The report also timed the run of both convolution and correlation for one-way and ordinary cases, however, a software update that greatly improved the performance of Nengo on lab hardware was released during development, and the data was rendered obsolete. There was no opportunity to re-run the test before submission, so the original diagnostic (using version 2.6) is documented in *Fig. 4*.

The original report made significant oversights that ultimately led to challenges in implementation of the project.

- The dot product operation was not examined; the neuronal implementation of the dot product ultimately used in the final project is less stable than expected
- Scalar division was not examined; it also turned out to be less stable than expected
- The cosine operation was not examined; this proved an especially significant hurdle, as neural populations in Nengo do not represent values outside of their radius very well
- Control subsystems that proved essential to the final implementation were overlooked; a cursor that selects the target word, a network that tracks the variance of a value over time, switching mechanisms, and networks that apply a mask to a set of words so that values can be

conditionally skipped were all important to the final design, and did not factor into the analysis at all. They had to be designed during the implementation step.

- BEAGLE has one significant stumbling block that prevents perfect implementation in Nengo. The growth of BEAGLE vectors is unbounded. As the number of episodes recorded increases, so the magnitude of item and order vectors tends to increase. Nengo neurons are only capable of representing a small range of values well for even a relatively large population size (radius 1, population size 50 tends to be the most successful). If any value of a vector exceeds +/- the radius of the population representing it, the quality of representation quickly diminishes to uselessness. So, a feasible model must constrain the growth of BEAGLE vectors somehow. Normalization as a method was considered, but was not analyzed along with the other operations.

The report examined vectors of size 1024, a common size used for BEAGLE. The following is an excerpt of the report, describing its method:

> *The properties of a State subnetwork (a holographic neural store of inputs) showed that the quality of the stored values would degrade in small measure as the number of traces present in the hologram increases. As States are the standard way of storing function outputs, the output of a State storing the result of a convolution, a one-way convolution, a cosine, and an addition were measured over 5000 time steps in order to gather output samples. The tests consisted of the presentation of one (of 100) vectors for each of two operands to the input functions, waiting for approximately 50 time steps (0.05 simulated seconds) for the output to stabilize, and then collecting the output for the next approximately 50 time steps, then presenting it with a new pair of vectors. Final values were compared to the results of their precise mathematical analogues.*

Four statistics were collected: The deviation of the grand mean from the value of the true operation, the grand variance of the sample, the mean of the variances of each sample, and the variances of the sample means about the value of the true operation. Each statistic was gathered by operation type and by synthetic similarity of the operands. The statistics measured the stability of the calculations as well as the quality of their approximation to the true operations they are intended to represent.

[…]

The five synthetic similarity conditions were: cos(a, b) = [0.0, 0.3, 0.5, 0.7, 0.9]; where a and b were the operands of the operation being performed. Vectors were synthetically related in order to determine whether the operations became unstable under successively closer relationships between their operands. Synthetic relating of vectors consisted of setting, for each pair a and b in the lexicon, a equal to a random Gaussian vector, and b equal to the normalized hologram of a random Gaussian vector and some proportion of a, where that proportion would produce approximately the desired cosine between the two vectors. The necessity of evaluating numeric instability under conditions of highly similar operands is due to the nature of the BEAGLE algorithm, which constructs interrelationships between vectors in the lexicon, with similarities of a range up to approximately 0.9, but usually not less than 0. If under any similarity condition Nengo became unstable, it may become impossible to use some BEAGLE vectors reliably.

[…]

For each statistic, 0 is a desirable value, indicating perfect accuracy and stability. Accuracy and stability to at least two decimal places was a desired quality of the Nengo software.

Architecture

Neural BEAGLE contains three core modules. Each module interacts with the others and with visual and memory nodes (*Fig. 5*):

1. **The processing subsystem:** Given a phrase of visual vectors and the location of a target word in that phrase, computes the item or order information for that word.
2. **The recording subsystem:** Given the existing state of a word and the new episode to record, calculates the weighted sum of the two, giving the new state. Yields a control vector whose stable value over time indicates that the state is no longer changing.
3. **The control subsystem:** Monitors the current state of the control vector yielded by the recording subsystem and the current target in the phrase. When the state of the control vector stops changing, selects the next word as the target and sends the signal to record a current encoding.

The core modules of BEAGLE are implemented exclusively using neurons (and nodes that do nothing but help organize interactions between neuron populations), while the visual and memory nodes are just functions that interact with the external state (stored vectors, corpus, stop list), providing an interface between the neural modules and the parts of the system that are necessary, but not part of the model.

Several subnetwork types did not exist in Nengo prior to this project and are crucial to its function. They are:

- **The zipper network:** Works like a conventional zipper function. Taking two vectors x and y as inputs, returns a set of paired values $[x_i, y_i]$. A function may be applied such that the output is a single vector of the same size as x and y and each value i in the vector is a function of $[x_i, y_i]$. Typically used for masking; a vector of either all 1s or all 0s is multiplied with the input vector in order to either completely include or completely omit it from an episode (*Fig. 6*)
- **The variance switch:** Tracks a value over a number of time steps. A series of neural populations each slowly take on the value of the last population in the sequence, and the variance of those populations is recorded. When the variance is below a threshold, the variance switch takes on a value of [1, 0]. Above the threshold, it has the value [0, 1]. A thalamus network, from Nengo, was used to implement switching (*Fig. 7)*
- **The rotating cursor:** A series of neural populations in which exactly one population has a value of 1, the rest being 0. The populations are all self-stimulating, according to a function that forces them towards either 1 or 0. Inhibitory connections exist between all populations except between a given population A and the population B that follows it in sequence, such that the dominant population will tend to force others towards 0. The connection from A to B is excitatory, but weak enough that the state of B will not change unless A grows greater than 1. In order to cause the cursor to advance, an external signal stimulates ("floods") all populations. A thus excites B, causing B to inhibit A and become the dominant item. The flood is not sufficient for any other value to overtake B. Once the external stimulus fades, B will be the dominant item and the cycle can repeat. The final item in the list has an excitatory connection with the first, hence the cursor "rotates" once the end is reached. A reset signal causes the cursor to reset to the initial value, [1, 0, 0…] (*Fig. 8*)

Two versions of the **processing subsystem** were written: One for item vectors (*Fig. 9*), one for order vectors (*Fig 10*). The item version is considerably simpler. As the order vector version contains a nested control subsystem, discussion will be held off for the moment.

The **item information processing subsystem** takes in a phrase as a set of visual vectors (excess vectors are assumed to be 0) and adds them together, omitting the target and stop words. A binary stop vector (a vector that indicates whether a word is a stop; 0 for stop words, 1 for non-stops) and the cursor are taken as inputs, in addition to the phrase. The cursor is inverted, such that it has a value of 1 everywhere except for the target. It is then multiplied with the stop vector in order to make a mask. The mask multiplies the phrase, and the result is summed in order to create the episode.

The **recording subsystem** simply takes the episode and the current state of the (item or order) BEAGLE vector as inputs. It normalizes both, and multiplies them by scalars, respectively 0.1 and 0.9 to keep the rate of forgetting relatively low, then it adds them together. It yields the normalized episode as its control vector, as it is the most variable vector in the subsystem, and stabilizes second-to-last (before the updated memory vector) (*Fig. 11*)

The **control subsystem** is the most intricate and important component. A variance switch monitors the control vector (given by the recording subsystem) dotted with its own previous state. The actual similarity of the two values doesn't matter, only that the dot product between them changes to indicate the control vector is changing. The variance switch triggers the cursor to select the next value when the monitored value stabilizes. It takes the "next phrase" signal as input from the external environment (*Fig. 12*)

The **order information processing subsystem** uses a nested control subsystem in order to slide an encoding window across the given phrase, replacing the target with PHI. Items in the window are sequentially convolved, with every set of convolutions containing the target. The target-selecting vector is used to create a mask that corresponds to a window onto the phrase. The window shows the 1-

4 vectors to the left of the target, PHI, and the 1-4 vectors to the right of the target. The second control subsystem selects a vector from which to start one-way convolutions, and the following 4 vectors are each used as operands in the convolutions (again, controlled using a mask). The results of all convolutions are summed (*Fig. 13*)

It's in the order information processing subsystem that the zipper mask as a tool for conditional omission starts to show its cracks. A phrase-sized mask is needed for each value in the window, and a window-sized mask is needed for each operand of the convolutions (that isn't just a previous convolution). As yet, there does not seem to a more efficient way of conditionally routing data in Nengo.

Implementation

The project was implemented in Nengo 2.8 under Python 3.6, using library functions from NumPy. Not all implementation goals were met by the time of the project's conclusion.

The following are design goals of the project:

- To fully implement the BEAGLE algorithm using the Nengo framework
- Keep the code to a high standard using modularization, enclosure of networks in instantiable objects, and ample documentation
- Perform piecewise tests on modules to ensure correct function
- Provide a simple shell interface for running Neural BEAGLE over a corpus of text
- Provide a graphical user interface in the Nengo visualizer demonstrating the semantic relations between words
- Parallelize the program to improve its runtime on capable hardware
- Statistically evaluate vectors computed by Neural BEAGLE in comparison to vectors computed by the Fortran implementation of BEAGLE in order to gauge the comparative performance

Only the first goal was met over the course of the project. The Fortran implementation of BEAGLE (not contained in this report) was heavily revised in order to facilitate testing, a process that proved time-consuming. The sheer unanticipated scale of the task played a major role. However, the project did get off to a good start and continued development will, hopefully, bear fruit. Code for the project is located in Appendix II.

Discussion

An ambition of this project was to use BEAGLE to begin to develop a dynamic, neurally embedded lexicon. The feasibility report contained a discussion of how that goal was revealed not to be accommodated by the SPA tools currently available. Above, there is a discussion of the NEF's inability to adjust weights after initialization.

The initial preoccupation with the neurally embedded lexicon was over the concept of a cleanup memory. A traditional cleanup memory, used to take the "noisy" trace retrieved from a correlation and turn it into its "true" representation, is just a matrix containing all available words in the lexicon; the trace is associated with the vector most similar to it, and that's taken to be the target. This model expands linearly with vocabulary size and is intolerant to damage, properties HRRs are supposed to fix!

The early motivation of the project was in search of a *holographic lexicon*. A method for item storage and retrieval in memory that would be denser, more robust and that wouldn't involve a linear search of vocabulary in order to clean up a given trace. Before the properties of the SPA were fully understood, it was thought that the properties of its neurons could be exploited in some way to create a holographic lexicon, initially via the associative memory network created by Eliasmith, and later by other means.

That search continues. It may turn out quixotic, but the imagined properties of a fully holographic memory system create interesting avenues of research.

BEAGLE is, fundamentally, a prototype theory of memory. In it remain gaping holes like "what are words?", "where do they come from?", "how do you figure out affixation?", "how do you know when to initialize a new prototype?". A fully holographic memory storage system might begin to answer these questions. A hologram can be analyzed such that two items in it which are extremely similar can also be viewed as an aggregate, single item.

A hologram has emergent characteristics that make it a fertile site for informal rule induction, creativity; a mostly-unexplored avenue in machine learning. Rasmussen and Eliasmith (2011) explored such a model.

Possible avenues for further work here include:

- Applying Hebbian learning to associative memory networks to adjust knowledge of the lexicon with time
- Using the encoding and decoding of neuron tuning curves (Eliasmith, 2012) to instantiate neural populations only capable of responding with 1 of 2 or more encoded values, else 0, and in this sense to holographically superpose vectors in long-term memory
- Returning to De Landa and using his mathematical notion of "progressive differentiation" to analyze the emergence of semantic objects (while De Landa's idea is fundamentally *physical*, describing how real space could emerge from symmetry-breaking transitions of simpler topological spaces, physics is also what provided us with the hologram, and tells us that real space could also be a hologram; Bousso, 2002)

This past year, Jamieson et. al. (2018) showed that semantic information of much the same sort BEAGLE encodes can be generated by simply storing episodes in memory and adding them up later on. Their model, however, does not solve the problem of unbounded lexicon growth, or create a particularly robust memory system. It does, however, show that an aggregated memory system that

only stores episodes can exhibit semantic qualities. This is exactly the property we would be looking for if our goal was to devise a fully holographic memory system.

References

Barres, B. A. (2008). The mystery and magic of glia: a perspective on their roles in health and disease. *Neuron*, *60*(3), 430-440.

Bousso, R. (2002). The holographic principle. *Reviews of Modern Physics*, *74*(3), 825.

Chomsky, N. (2006). *Language and mind*. Cambridge University Press.

DeLanda, M. (2013). *Intensive science and virtual philosophy*. Bloomsbury Publishing.

Dreyfus, H. L. (2007). Why Heideggerian AI failed and how fixing it would require making it more Heideggerian. *Philosophical psychology*, *20*(2), 247-268.

Eliasmith, Chris. *How to build a brain: A neural architecture for biological cognition*. Oxford University Press, 2013.

Haraway, D. J. (1985). *A manifesto for cyborgs: Science, technology, and socialist feminism in the 1980s* (pp. 173-204). San Francisco, CA: Center for Social Research and Education.

Harnad, S. (1990). The symbol grounding problem. *Physica D: Nonlinear Phenomena*, *42*(1-3), 335-346.

Jamieson, R. K., Avery, J. E., Johns, B. T., & Jones, M. N. (2018). An instance theory of distributional semantics. In *Proceedings of the 39th Conference of the Cognitive Science Society. Austin TX: Cognitive Science Society. Google Scholar*.

Johns, B. T., & Jones, M. N. (2015). "Generating structure from experience: A retrieval-based model of language processing". *Canadian Journal of Experimental Psychology, 69*, 233-251.

Jones, M. N., & Mewhort, D. J. K. (2007). "Representing word meaning and order information in a composite holographic lexicon". *Psychological Review, 114*(1), 1-37.

Mewhort, D. J. K., Kevin D. Shabahang, and D. R. J. Franklin. "Release from PI: An analysis and a model." *Psychonomic bulletin & review* (2017): 1-19.

Murdock, B. B. (1982). A theory for the storage and retrieval of item and associative information. *Psychological Review*, *89*(6), 609.

Nengo [Computer software]. (2018). Retrieved from https://github.com/nengo/nengo

Newell, A. (1980). Physical symbol systems. *Cognitive science*, *4*(2), 135-183.

Nilsson, N. J. (2007). The physical symbol system hypothesis: status and prospects. In *50 years of artificial intelligence* (pp. 9-17). Springer, Berlin, Heidelberg.

Plate, T. A. (1995). Holographic reduced representations. *IEEE Transactions on Neural networks*, *6*(3), 623-641.

Popper, K. R., & Miller, D. (1953). The problem of induction.

Rasmussen, D., & Eliasmith, C. (2011). A neural model of rule generation in inductive reasoning. *Topics in Cognitive Science*, *3*(1), 140-153.

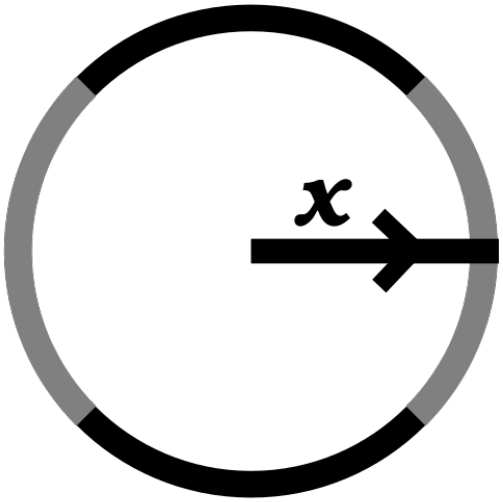
Searle, J. R. (1980). Minds, brains, and programs. *Behavioral and brain sciences*, *3*(3), 417-424.

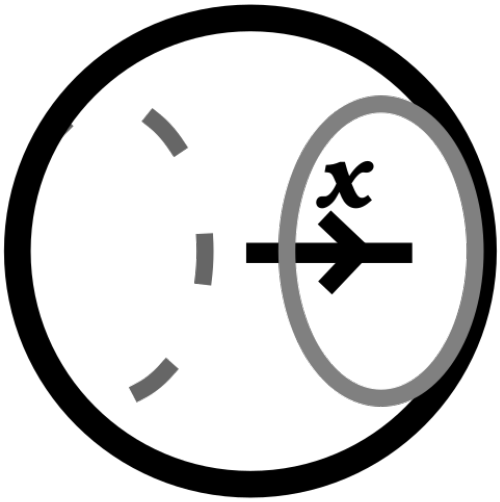
Sutton, J., Harris, C. B., Keil, P. G., & Barnier, A. J. (2010). The psychology of memory, extended cognition, and socially distributed remembering. *Phenomenology and the cognitive sciences*, *9*(4), 521-560.

Usher, M., Schuster, H. G., & Niebur, E. (1993). Dynamics of populations of integrate-and-fire neurons, partial synchronization and memory. *Neural Computation*, *5*(4), 570-586.

Wilson, M. (2002). Six views of embodied cognition. *Psychonomic bulletin & review*, *9*(4), 625-636.

Figures





$p(\cos(x, y) \in [-0.5, 0.5]) = 1/2$ $p(\cos(x, y) \in [-0.5, 0.5]) > 2/3$

Fig. 1: The unit circle and unit sphere, with areas more alike +/-x than unlike it enclosed in grey.

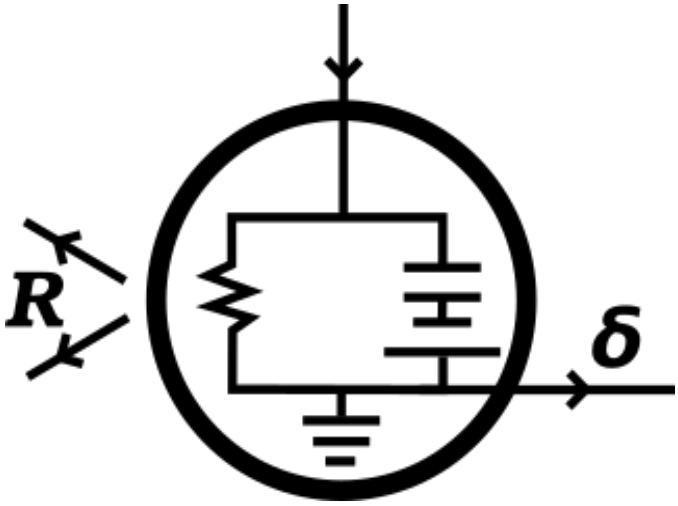


Fig. 2: A circuit diagram visualization of a LIF neuron. There are several models of LIF neuron, but in general, they contain a leakage parameter (a resistor), and a capacitance parameter (the capacitor) that accumulates charge. When it discharges, the neuron spikes at a fixed voltage (represented by the battery), usually represented as a Dirac delta δ

Fig. 3: Statistical Analysis of Nengo

Neural analogue to NumPy task for synthetic similarity conditions 0.0, 0.3, 0.5, 0.7, 0.9

0.0

+

Deviation of grand mean: μ((μ$\bar{\bar{x}}$ – μ)2) = 0.0000

Grand variance: μ(σ2$\bar{\bar{x}}$) = 0.0000

Mean of sample variances: μ(μ(σ2$\bar{x}$)) = 0.0015

Variance of sample means: σ2(μ, μ$\bar{x}$) = 0.0000

*

Deviation of grand mean: μ((μ$\bar{\bar{x}}$ – μ)2) = 0.0000

Grand variance: μ(σ2$\bar{\bar{x}}$) = 0.0000

Mean of sample variances: μ(μ(σ2$\bar{x}$)) = 0.0001

Variance of sample means: σ2(μ, μ$\bar{x}$) = 0.0000

*>

Deviation of grand mean: μ((μ$\bar{\bar{x}}$ – μ)2) = 0.0019

Grand variance: μ(σ2$\bar{\bar{x}}$) = 0.0020

Mean of sample variances: μ(μ(σ2$\bar{x}$)) = 0.0010

Variance of sample means: σ2(μ, μ$\bar{x}$) = 0.0000

0.3

+

Deviation of grand mean: μ((μ$\bar{\bar{x}}$ – μ)2) = 0.0001

Grand variance: μ(σ2$\bar{\bar{x}}$) = 0.0001

Mean of sample variances: μ(μ(σ2$\bar{x}$)) = 0.0017

Variance of sample means: σ2(μ, μ$\bar{x}$) = 0.0000

*

Deviation of grand mean: μ((μ$\bar{\bar{x}}$ – μ)2) = 0.0000

Grand variance: μ(σ2$\bar{\bar{x}}$) = 0.0000

Mean of sample variances: μ(μ(σ2$\bar{x}$)) = 0.0001

Variance of sample means: σ2(μ, μ$\bar{x}$) = 0.0000

*>

Deviation of grand mean: μ((μ$\bar{\bar{x}}$ – μ)2) = 0.0019

Grand variance: μ(σ2$\bar{\bar{x}}$) = 0.0019

Mean of sample variances: μ(μ(σ2$\bar{x}$)) = 0.0010

Variance of sample means: σ2(μ, μ$\bar{x}$) = 0.0000

0.5

+

Deviation of grand mean: μ((μ$\bar{\bar{x}}$ – μ)2) = 0.0002

Grand variance: μ(σ2$\bar{\bar{x}}$) = 0.0002

Mean of sample variances: μ(μ(σ2$\bar{x}$)) = 0.0020

Variance of sample means: σ2(μ, μ$\bar{x}$) = 0.0000

*

Deviation of grand mean: μ((μ$\bar{\bar{x}}$ – μ)2) = 0.0000

Grand variance: μ(σ2$\bar{\bar{x}}$) = 0.0000

Mean of sample variances: μ(μ(σ2$\bar{x}$)) = 0.0001

Variance of sample means: σ2(μ, μ$\bar{x}$) = 0.0000

*>

Deviation of grand mean: μ((μ$\bar{\bar{x}}$ – μ)2) = 0.0019

Grand variance: μ(σ2$\bar{\bar{x}}$) = 0.0019

Mean of sample variances: μ(μ(σ2$\bar{x}$)) = 0.0010

Variance of sample means: σ2(μ, μ$\bar{x}$) = 0.0000

0.7

+

Deviation of grand mean: μ((μ$\bar{\bar{x}}$ – μ)2) = 0.0002

Grand variance: μ(σ2$\bar{\bar{x}}$) = 0.0002

Mean of sample variances: μ(μ(σ2$\bar{x}$)) = 0.0021

Variance of sample means: σ2(μ, $\mu\bar{x}$) = 0.0000

*

Deviation of grand mean: μ(($\mu\bar{\bar{x}}$ – μ)2) = 0.0000

Grand variance: μ($\sigma 2\bar{\bar{x}}$) = 0.0001

Mean of sample variances: μ(μ($\sigma 2\bar{x}$)) = 0.0001

Variance of sample means: σ2(μ, $\mu\bar{x}$) = 0.0000

*>

Deviation of grand mean: μ(($\mu\bar{\bar{x}}$ – μ)2)= 0.0019

Grand variance: μ($\sigma 2\bar{\bar{x}}$) = 0.0019

Mean of sample variances: μ(μ($\sigma 2\bar{x}$)) = 0.0010

Variance of sample means: σ2(μ, $\mu\bar{x}$) = 0.0000

0.9

+

Deviation of grand mean: μ(($\mu\bar{\bar{x}}$ – μ)2) = 0.0003

Grand variance: μ($\sigma 2\bar{\bar{x}}$) = 0.0003

Mean of sample variances: μ(μ($\sigma 2\bar{x}$)) = 0.0022

Variance of sample means: σ2(μ, $\mu\bar{x}$) = 0.0000

*

Deviation of grand mean: μ(($\mu\bar{\bar{x}}$ – μ)2) = 0.0002

Grand variance: μ($\sigma 2\bar{\bar{x}}$) = 0.0002

Mean of sample variances: μ(μ($\sigma 2\bar{x}$)) = 0.0001

Variance of sample means: σ2(μ, $\mu\bar{x}$) = 0.0000

*>

Deviation of grand mean: μ(($\mu\bar{\bar{x}}$ – μ)2) = 0.0019

Grand variance: μ($\sigma 2\bar{\bar{x}}$) = 0.0020

Mean of sample variances: μ(μ($\sigma 2\bar{x}$)) = 0.0010

Variance of sample means: σ2(μ, $\mu\bar{x}$) = 0.0000

Cleanup Memory task for synthetic similarity conditions 0.0, 0.3, 0.5, 0.7, 0.9

0.0

*

Deviation of grand mean: $\mu((\mu\bar{\bar{x}} - \mu)2) = 0.0005$

Grand variance: $\mu(\sigma 2\bar{\bar{x}}) = 0.0010$

Mean of sample variances: $\mu(\mu(\sigma 2\bar{x})) = 0.0010$

Variance of sample means: $\sigma 2(\mu, \mu\bar{x}) = 0.0000$

*>

Deviation of grand mean: $\mu((\mu\bar{\bar{x}} - \mu)2) = 0.0010$

Grand variance: $\mu(\sigma 2\bar{\bar{x}}) = 0.0014$

Mean of sample variances: $\mu(\mu(\sigma 2\bar{x})) = 0.0010$

Variance of sample means: $\sigma 2(\mu, \mu\bar{x}) = 0.0000$

0.3

*

Deviation of grand mean: $\mu((\mu\bar{\bar{x}} - \mu)2) = 0.0005$

Grand variance: $\mu(\sigma 2\bar{\bar{x}}) = 0.0010$

Mean of sample variances: $\mu(\mu(\sigma 2\bar{x})) = 0.0010$

Variance of sample means: $\sigma 2(\mu, \mu\bar{x}) = 0.0000$

*>

Deviation of grand mean: $\mu((\mu\bar{\bar{x}} - \mu)2) = 0.0010$

Grand variance: $\mu(\sigma 2\bar{\bar{x}}) = 0.0014$

Mean of sample variances: $\mu(\mu(\sigma 2\bar{x})) = 0.0010$

Variance of sample means: $\sigma 2(\mu, \mu\bar{x}) = 0.0000$

0.5

*

Deviation of grand mean: $\mu((\mu\bar{\bar{x}} - \mu)2) = 0.0005$

Grand variance: $\mu(\sigma 2\bar{\bar{x}}) = 0.0010$

Mean of sample variances: μ(μ(σ2$\bar{x}$)) = 0.0010

Variance of sample means: σ2(μ, μ$\bar{x}$) = 0.0000

*>

Deviation of grand mean: μ((μ$\bar{\bar{x}}$ – μ)2) = 0.0010

Grand variance: μ(σ2$\bar{\bar{x}}$) = 0.0014

Mean of sample variances: μ(μ(σ2$\bar{x}$)) = 0.0010

Variance of sample means: σ2(μ, μ$\bar{x}$) = 0.0000

0.7

*

Deviation of grand mean: μ((μ$\bar{\bar{x}}$ – μ)2) = 0.0005

Grand variance: μ(σ2$\bar{\bar{x}}$) = 0.0010

Mean of sample variances: μ(μ(σ2$\bar{x}$)) = 0.0010

Variance of sample means: σ2(μ, μ$\bar{x}$) = 0.0000

*>

Deviation of grand mean: μ((μ$\bar{\bar{x}}$ – μ)2) = 0.0010

Grand variance: μ(σ2$\bar{\bar{x}}$) = 0.0014

Mean of sample variances: μ(μ(σ2$\bar{x}$)) = 0.0010

Variance of sample means: σ2(μ, μ$\bar{x}$) = 0.0000

0.9

*

Deviation of grand mean: μ((μ$\bar{\bar{x}}$ – μ)2) = 0.0005

Grand variance: μ(σ2$\bar{\bar{x}}$) = 0.0010

Mean of sample variances: μ(μ(σ2$\bar{x}$)) = 0.0010

Variance of sample means: σ2(μ, μ$\bar{x}$) = 0.0000

*>

Deviation of grand mean: μ((μ$\bar{\bar{x}}$ – μ)2) = 0.0010

Grand variance: μ(σ2$\bar{\bar{x}}$) = 0.0014

Mean of sample variances: μ(μ(σ2$\bar{x}$)) = 0.0010

Variance of sample means: σ2(μ, μx̄) = 0.0000

Fig. 4	*	*>
Convolution	30m	38m
Correlation	71m	73m

Fig. 4: Time (minutes) to complete 100 iterations of the designated operation for 0.1 simulated seconds per iteration in Nengo 2.6

Appendix I

```
// BEAGLE in pseudocode
// Based on the current Fortran implementation of BEAGLE
// let target be an integer that indexes phrase
// let phrase be a list of integers that index visual, item, order, stops
// let stops be a boolean array in which a value of 1 at index i
// indicates that word i is a stop word

function item_info(target, phrase, stops, visual):
      work = [0., ..]
      for i=1..phrase.len:
            if i != target and not stops[phrase[i]]:
                  work += visual[phrase[i]]

      return work

function order_info(target, phrase, visual)
      work = [0., ..]
      for i=target – MAXWINDOW + 1..target:
            if i > 0: ! lower bound check
                  if i == target:
                        tmp = PHI
                  else:
                        tmp = visual[phrase[i]]

      for j=i+1..i + MAXWINDOW – 1:
            if i <= phrase.len: ! Upper bound check
                  if i == target:
                        tmp = one_way_convolve(tmp, PHI)
                  else:
                        tmp = one_way_convolve(tmp, visual[phrase[j]])
            if i >= target:
                  work += tmp

      return work

function beagle(corpus, stops, visual):
      items = [[0., ..], ..]
      order = [[0., ..], ..]

      foreach phrase in corpus:
            for target=1..phrase.len:
                  items[phrase[target]] += item_info(target, phrase, stops, visual)
                  order[phrase[target]] += order_info(target, phrase, visual)

      return items, order
```

Appendix II

FILE: NETWORKS.PY

```
# networks
# a library of Nengo-network generating functions

import nengo
from nengo import spa
from nengo.dists import Exponential
import numpy as np

###### Functions applied to connections
# Applied by an ensemble onto itself, hovers about either 0 or 1(ish) and resists
being changed
def hold(x):
    return x - 16 * x * (x - 1) * (x - .5)

# Used in a Connection from an Ensemble onto an output,
# sigmoid distribution forces value closer to either 0 or 1
# approximates step function
def binarize(x):
    return 1/(np.exp((1 - 2*x) * 32) + 1)

# Multiplies two zipped values together
# Useful together with zipper network
def prod(x):
    return x[0] * x[1]

def sq(x):
    return x**2

###### Supporting networks

# calculates variance over a given number of intervals (each interval connects
# to the next with a synapse=0.05 and that does not work in an obvious way)
def variance_calculator(intervals):
    var = nengo.Network("variance")

    with var:
        var.input = nengo.Node(size_in=1, label='input')

        # Collect snapshots of recent values in sequence
        time_arr = []

        time_out = nengo.Node(size_in=intervals, label='time_out')

        for i in range(intervals):
            e = nengo.Ensemble(n_neurons=50, dimensions=1, label="t-{}".format(i))
            time_arr.append(e)

            if i > 0: nengo.Connection(time_arr[i-1], e, synapse=0.05)

            nengo.Connection(e, time_out[i])

        nengo.Connection(var.input, time_arr[0])
```

```
        # Feed snapshots into a time ensemble to hold the state
        time = nengo.Ensemble(n_neurons=50, dimensions=intervals, label='time')
        nengo.Connection(time_out, time)

        # Take the mean of snapshots in an ensemble
        mean = nengo.Ensemble(n_neurons=50, dimensions=1, label='mean')
        for i in range(intervals):
            nengo.Connection(time[i], mean, transform=1/intervals)

        # Calculate (x[n] - mu)^2 for each x[n]
        amms_in = nengo.Node(size_in=intervals, label='(x-mu)^2')
        nengo.Connection(mean, amms_in, transform=np.ones((intervals, 1)) * -1)
        nengo.Connection(time, amms_in)

        # output will just be the sum of (x[n] - mu)^2
        var.output = nengo.Ensemble(n_neurons=50, dimensions=1)
        amms_arr = []

        for i in range(intervals):
            e = nengo.Ensemble(n_neurons=50, dimensions=1, label="(x[{}]-
mu)^2".format(i))
            amms_arr.append(e)
            nengo.Connection(amms_in[i], e)
            nengo.Connection(e, var.output, function=sq)

    return var

# A switch whose value 'reset' indicates when a value has stabilized
def variance_switch(intervals, thr):
    switch = nengo.Network('variance switch')

    with switch:
        variance = variance_calculator(intervals)
        switch.input = variance.input

        thresh = nengo.Node([thr], label='threshold')

        # Basal ganglia + thalamus combo to calculate dominant value
        bg = nengo.networks.BasalGanglia(dimensions=2)
        th = nengo.networks.Thalamus(dimensions=2)
        nengo.Connection(bg.output, th.input)
        nengo.Connection(thresh, bg.input[0])
        nengo.Connection(variance.output, bg.input[1])

        switch.output = th.output

        # reset is nonzero when variance is small
        switch.reset = th.output[0]

    return switch

def switch_mask(dimensions, target_state, label='switch mask'):
    sw_mask = nengo.Network(label)

    with sw_mask:
        sw_mask.target = nengo.Node(target_state)
```

```
        sw_mask.dot = dot_product(len(target_state))
        nengo.Connection(sw_mask.target, sw_mask.dot.input[0])
        sw_mask.switch = sw_mask.dot.input[1]

        sw_mask.zip = zipper(dimensions)
        nengo.Connection(sw_mask.dot.output, sw_mask.zip.input[0],
transform=np.ones((dimensions, 1)))
        sw_mask.input = sw_mask.zip.input[1]

        sw_mask.output = nengo.Node(size_in=dimensions)
        for i in range(dimensions):
            nengo.Connection(sw_mask.zip.output[i], sw_mask.output[i],
function=prod)

    return sw_mask

# Yields zipper network:
# analogous to a zip function, takes two (or more) arrays and pairs them item-wise
# Necessary for applying transformations to paired values
# Optional function value enables the specification of many-to-one function maps
# for each zipper index
def zipper(dimensions, label='zipper', inputs=2, function=None):
    zip_net = nengo.Network(label)

    with zip_net:
        zip_net.input = [nengo.Node(size_in=dimensions) for _ in range(inputs)]

        zip_net.zip = []

        for i in range(dimensions):
            z = nengo.Ensemble(n_neurons=50*inputs, dimensions=inputs, radius=1.5)
            zip_net.zip.append(z)
            for inp in range(inputs):
                nengo.Connection(zip_net.input[inp][i], z[inp])

        if function:
            zip_net.output = nengo.Node(size_in=dimensions)
            for i, z in enumerate(zip_net.zip):
                nengo.Connection(z, zip_net.output[i], function=function)

        else:
            zip_net.output = zip_net.zip


    return zip_net

def dot_product(dimensions, n_neurons=100, radius=3, label='dot product'):
    dot = nengo.Network(label)

    with dot:
        dot.zip = zipper(dimensions, function=prod)
        dot.output = nengo.Ensemble(n_neurons=n_neurons, dimensions=1,
radius=radius)
        dot.input = dot.zip.input

        for i in range(dimensions):
            nengo.Connection(dot.zip.output[i], dot.output)
```

```
    return dot

def cosine(dimensions, n_neurons=100, radius=3):
    cos = nengo.Network('cosine')

    with cos:
        cos.num = dot_product(dimensions, n_neurons=n_neurons, radius=radius,
label='numerator')
        cos.input = cos.num.input

        cos.sq_norms = [dot_product(dimensions, n_neurons=n_neurons, radius=3,
label='dot {}'.format(i)) for i in range(len(cos.input))]

        cos.output_zip = zipper(1, label='output zip', inputs=3, function=lambda x:
x[0]/(np.sqrt(x[1]) * np.sqrt(x[2])) \
                                if x[1] > 0 and x[2] > 0 else 0)
        cos.output = cos.output_zip.output

        for i, inp in enumerate(cos.input):
          for sqno in cos.sq_norms[i].input:
            nengo.Connection(inp, sqno)
          nengo.Connection(cos.sq_norms[i].output, cos.output_zip.input[i+1])

        nengo.Connection(cos.num.output, cos.output_zip.input[0])

    return cos


# Yields an array of zipper networks
# index the 0-indices with [:dimensions*phrase_length] and the 1-indices with
[dimensions*phrase_length:]
def zipper_array(dimensions, phrase_length, label='zipper array'):
    zarr = nengo.Network(label)

    with zarr:
        zarr.input = nengo.Node(size_in=phrase_length*2*dimensions)
        zarr.zippers = []
        zarr.output = []
        for i in range(phrase_length):
            z = zipper(dimensions)
            zarr.zippers.append(z)
            nengo.Connection(zarr.input[i*dimensions:(i+1)*dimensions], z.input[0])
            nengo.Connection(zarr.input[i*2*dimensions:(i*2+1)*dimensions],
z.input[1])

            zarr.output += z.output

    return zarr

# Yields inverter network:
# What it says on the tin: Given x in [0, 1], yields 1 - x for all values
def inverter(dimensions, label='inverter'):
    inv = nengo.Network(label)

    with inv:
        inv.input=nengo.Node(size_in=dimensions)
```

```
        inv.output=nengo.Node(size_in=dimensions)

        inv.ensembles = []
        for i in range(dimensions):
            e = nengo.Ensemble(n_neurons=50, dimensions=1)
            inv.ensembles.append(e)
            nengo.Connection(inv.input[i], e)
            nengo.Connection(e, inv.output[i], function=lambda x: 1-x)

    return inv

# Yields rotating cursor network:
# An array with exactly one subscript that holds a value of 1, the rest are 0
# If you flood the network and then return the flood to 0, it will select the next
item,
# looping around at the end of the list
def rotating_cursor(phrase_length, label='rotating cursor', function=binarize):
    cursor = nengo.Network(label)

    with cursor:
        # Generate interface objects in Network container
        cursor.input = nengo.Node(size_in=phrase_length)
        cursor.flood = nengo.Node(size_in=1)
        cursor.reset = nengo.Node(size_in=1)
        cursor.output = nengo.Node(size_in=phrase_length)

        # Generates an addressable list of Ensembles to store values
        # in the cursor
        cursor.indices = []


        for i in range(phrase_length):
            e = nengo.Ensemble(n_neurons=50, dimensions=1,
label='cursor[{}]'.format(i),
                intercepts=Exponential(.15, 0, 1))
            cursor.indices.append(e)
            nengo.Connection(e, e, function=hold, synapse=.1)
            nengo.Connection(cursor.flood, e, transform=[[.8]])
            nengo.Connection(e, cursor.output[i], function=function)
            nengo.Connection(cursor.input[i], e)
            if i > 0:
                nengo.Connection(e, cursor.indices[i-1], transform=[[-.8]],
synapse=.1)
                nengo.Connection(cursor.indices[i-1], e, transform=[[.2]],
synapse=.1)
                nengo.Connection(cursor.reset, e, transform=[[-1.5]])
            else:
                nengo.Connection(cursor.reset, e)
        nengo.Connection(cursor.indices[0], cursor.indices[-1], transform=[[-.8]],
synapse=.1)
        nengo.Connection(cursor.indices[-1], cursor.indices[0], transform=[[.2]],
synapse=.1)

    return cursor

# an array of SPA State objects (Networks that retain a state vector)
def state_array(dimensions, phrase_length, label='state array'):
```

```
    arr = spa.SPA(label)

    with arr:
        arr.states = []

        arr.input = nengo.Node(size_in=dimensions * phrase_length)
        arr.output = nengo.Node(size_in=dimensions * phrase_length)

        for i in range(phrase_length):
            setattr(arr, 'state_{}'.format(i), spa.State(dimensions,
subdimensions=dimensions))
            arr.states.append(getattr(arr, 'state_{}'.format(i)))
            nengo.Connection(arr.input[i*dimensions:(i+1)*dimensions],
arr.states[i].input)

            nengo.Connection(arr.states[i].output, arr.output[i*dimensions:
(i+1)*dimensions])

    return arr

def normalizer(dimensions, label='normalize'):
    normalize = nengo.Network(label)

    with normalize:
        normalize.vec = [nengo.Ensemble(n_neurons=50, dimensions=1) for _ in
range(dimensions)]
        normalize.input = nengo.Node(size_in=dimensions)

        normalize.magnitude = nengo.Ensemble(n_neurons=500, dimensions=1, radius=4)

        normalize.zip = zipper(dimensions)

        nengo.Connection(normalize.magnitude, normalize.zip.input[1],
transform=np.ones((dimensions, 1)))

        normalize.normvec = nengo.networks.EnsembleArray(n_neurons=50,
n_ensembles=dimensions)
        normalize.output = normalize.normvec.output

        for i in range(dimensions):
            nengo.Connection(normalize.input[i], normalize.vec[i])
            nengo.Connection(normalize.vec[i], normalize.zip.input[0][i])
            nengo.Connection(normalize.vec[i], normalize.magnitude, function=sq)
            nengo.Connection(normalize.zip.output[i], normalize.normvec.input[i],
function=lambda x: x[0] / x[1] if x[1] != 0 else 0)

    return normalize
```

FILE: NEURAL_BEAGLE.PY

```
import nengo
from nengo import spa
from nengo.dists import Exponential
import numpy as np
from networks import *
```

```
###### To be enclosed in a class
dimensions = 4
phrase_length = 3
window_size = 5
retention = 0.1
min_variance = 0.2
phi = np.random.normal(dimensions)
phi = phi/np.linalg.norm(phi)
left_perm = np.array(range(dimensions))
right_perm = np.array(range(dimensions))
np.random.shuffle(left_perm)
np.random.shuffle(right_perm)
# declare items, order, visual, phi, p1, p2, ready file

# returns the item vector of the current target
# takes concatenation [word ids] + [cursor] + [updated value]
# yields concatenation [old value for target] + [next phrase signal]
def memory_access(_, inp):
    # selects the value in the phrase index of the current cursor position
    trg = int(inp[np.argmax(inp[phrase_length:phrase_length*2])])

    # Falling edge only
    if trg != store_trg and not last_word_setting:
        last_word_setting = True
        items[trg] = inp[phrase_length*2:]
        store_trg = trg
    elif trg == store_trg and last_word_setting:
        last_word_setting = False

    if trg == -1:
        trg_vector = np.zeros((dimensions+1))
        trg_vector[-1] = 1. # send reset signal on end of list
    else:
        trg_vector[:-1] = items[trg]
        trg_vector[-1] = 0.

    return trg_vector


# uses global ready file as list of phrases, containing integers
# inp = 0/1 "next phrase prompt"
def process_input(_, inp):
    # next phrase on rising edge only
    if inp[0] and not last_phrase_setting:
        last_phrase_setting = inp[0]
        current_phrase += 1
        if current_phrase >= len(ready):
            complete = True
            pass # terminate processing
    elif not inp[0] and last_phrase_setting:
        last_setting = inp[0]

    viz_phrase = [visual[i] for i in ready[current_phrase] if i >= 0]
    stops_phrase = [stops[i] for i in ready[current_phrase] if i >= 0]
    void = phrase_length - len(stops_phrase)
```

```
    return np.concatenate((viz_phrase, np.zeros((dimensions, void)), \
                           stops_phrase, np.ones(void), \
                           ready[current_phrase]))


# includes cursor, variance switch, last vector storage, update detection dot
product,
# reinforce and replace switch masks
def control_block(dimensions, phrase_length):
    control = spa.SPA('control block')

    with control:
        ##### declare elements
        # one-hot encoding of current item in list
        control.rot_cursor = rotating_cursor(phrase_length)
        # variance switch detects that an episode's value has stabilized and
signals
        # the cursor to select the next item
        control.sw = variance_switch(10, 0.4)
        # stores the previous episode for detection stabilization
        # (its actual value isn't all that important it's just a stable benchmark
        # to measure change against)
        control.last = spa.State(dimensions)
        # dot product of last vector + current value, we measure this thing's
variance
        # to tell when the episode is changing and when it's stable
        control.dot = dot_product(dimensions)
        # this is the most awkward part of the whole thing
        # this is a switched feedback/replacement mechanism - we use the variance
        # switch to either feed back the value of the last vector or replace it
with a
        # new value
        control.reinforce = switch_mask(dimensions, [0, 1], label='reinforce')
        control.replace = switch_mask(dimensions, [1, 0], label='replace')

        ##### declare interface
        # passthrough node for the control vector because it needs to go to 2
places
        # the vector used to control the system (the value of the current episode)
        ### inputs
        control.control_vector = nengo.Node(size_in=dimensions)
        control.next_phrase = control.rot_cursor.reset

        ### outputs
        control.cursor = control.rot_cursor.output

        ##### wire critical connections
        nengo.Connection(control.last.output, control.reinforce.input)
        nengo.Connection(control.control_vector, control.replace.input)
        nengo.Connection(control.control_vector, control.dot.input[0])
        nengo.Connection(control.last.output, control.dot.input[1])
        nengo.Connection(control.dot.output, control.sw.input)

        nengo.Connection(control.reinforce.output, control.last.input)
        nengo.Connection(control.replace.output, control.last.input)
```

```
        nengo.Connection(control.sw.output, control.reinforce.switch)
        nengo.Connection(control.sw.output, control.replace.switch)
        nengo.Connection(control.sw.reset, control.rot_cursor.flood)


    return control

# includes inverter, stoplist, cursor+stop zipper, mask,
# masked off phrase representation, sum of phrase
# stoplist is just a dummy here, it is not used
# calculates order vector, intended to create an object that can be drop-in
replaced
# by an item vector function
def processing_block_order(dimensions, phrase_length, window_size, phi, left_perm,
right_perm):
    left_transform = np.zeros((dimensions, dimensions))
    right_transform = np.zeros((dimensions, dimensions))

    for i in range(dimensions):
        left_transform[i, left_perm[i]] = 1
        right_transform[i, right_perm[i]] = 1

    processing = spa.SPA('order vectors processing block')

    with processing:
        ##### declare elements
        # zips input phrase with cursor
        processing.phrase_zip = [zipper_array(dimensions, phrase_length,
label='phrase mask[{}]'.format(i)) for i in range(2*(window_size - 1))]
        processing.masked_phrase = nengo.Node(size_in=dimensions*2*(window_size -
1), size_out=dimensions*(2*window_size-1), output=lambda
x:np.concatenate((x[:window_size-1], phi, x[window_size-1:])))
        # for when we just need a value to be 0
        processing.zero = nengo.Node([0]*dimensions)
        # summed phrase elements
        processing.sum = spa.State(dimensions)

        # A second zipper array masks off our window
        processing.window_zip = [zipper_array(dimensions, window_size*2-1,
label='window mask[{}]'.format(i)) for i in range(window_size)]
        processing.scramble = [nengo.Node(size_in=dimensions) for _ in
range(window_size)]
        # selects the start position of the computation
        processing.window_cursor = rotating_cursor(window_size)

        processing.inputs = []
        processing.convolutions = []
        processing.outputs = []

        # Running the convolutions through an SPA cortical
        for i in range(window_size-1):
            setattr(processing, 'inputs_{}'.format(i), spa.State(dimensions))
            setattr(processing, 'convolutions_{}'.format(i), spa.State(dimensions))
            processing.inputs.append(getattr(processing, 'inputs_{}'.format(i)))
            processing.convolutions.append(getattr(processing,
'convolutions_{}'.format(i)))
```

```
            if i < window_size - 2:
                setattr(processing, 'outputs_{}'.format(i), spa.State(dimensions))
                processing.outputs.append(getattr(processing,
'outputs_{}'.format(i)))

        setattr(processing, 'inputs_{}'.format(i+1), spa.State(dimensions))
        processing.inputs.append(getattr(processing, 'inputs_{}'.format(i+1)))

        processing.conv_zip = zipper(window_size-1, label='convolution zip')
        processing.conv_mask = nengo.Node(size_in=window_size-1)

        processing.sum_zip = zipper_array(dimensions, window_size-1, label='sum
mask')

        ##### declare interface
        ### inputs
        processing.stops = nengo.Node(size_in=phrase_length)
        processing.cursor = nengo.Node(size_in=phrase_length)
        processing.phrase = nengo.Node(size_in=dimensions*phrase_length)

        ##### wire critical connections
        # copy phrase across all masking zippers
        [nengo.Connection(processing.phrase, pz.input[dimensions*phrase_length:])
for pz in processing.phrase_zip]
        # connect cursor to each of the 8 masks, left-shifting or right-shifting
based on each
        # word's distance from the target
        # this procedure contains two steps; i counts the number of positions to
shift, and j counts the value in the cursor
        # being shifted
        # a slice [k*dimensions:(k+1)*dimensions] specifies the application of the
mask to the kth vector in the phrase,
        # a transformation matrix (dimensions, 1) specifies an expansion of the
mask such that it applies to (and will multiply)
        # all the values of the kth vector
        # Moving from the inside, outwards, we perform a left shift of 1 on the
cursor to obtain the mask for mask[3],
        # and a right shift of 1 on the cursor to obtain the mask for mask[4].
Thus, mask[3] and [4] represent the vectors
        # directly to the left and right of the target, respectively
        # If the cursor falls off the edge, the value of mask[n] will simply be a 0
vector
        # In our second step for i=1, want to map cursor position 1 to the 0th word
in the phrase for mask[3],
        # and cursor position -2 to the -1st word in the phrase for mask[4], and so
on
        # For i=2 we want cursor[2] -> phrase[0] | mask[2], cursor[-3] -> phrase[-
1] | mask[5], and so on
        # thus, we calculate l = j-i and m = (l, phrase_length-l-1) for each j for
each i, giving us m = (0, phrase_length-1) for cursor[1] and [-2], respectively
        # Omitted values are connected to the 0 vector
        for i in range(1, min(phrase_length, window_size)):
            for j in range(i):
                nengo.Connection(processing.zero,
processing.phrase_zip[window_size-i-1].input[(phrase_length-j-1)*dimensions:
(phrase_length-j)*dimensions])
```

```
                nengo.Connection(processing.zero,
processing.phrase_zip[window_size+i-2].input[j*dimensions:(j+1)*dimensions])
            for j in range(i, phrase_length):
                l = j-i
                nengo.Connection(processing.cursor[j],
processing.phrase_zip[window_size-i-1].input[l*dimensions:(l+1)*dimensions],
transform=np.ones((dimensions, 1)))
                nengo.Connection(processing.cursor[-j-1],
processing.phrase_zip[window_size+i-2].input[(phrase_length-l-1)*dimensions:
(phrase_length-l)*dimensions], transform=np.ones((dimensions, 1)))

        # Say our phrase is shorter than our window size, we want to explicitly 0
everything that won't be used
        # In practice this should never happen
        for i in range(phrase_length, window_size):
            nengo.Connection(processing.zero, processing.phrase_zip[window_size-i-
1].input[:dimensions*phrase_length],
transform=np.ones((dimensions*phrase_length,dimensions)))
            nengo.Connection(processing.zero, processing.phrase_zip[window_size+i-
2].input[:dimensions*phrase_length],
transform=np.ones((dimensions*phrase_length,dimensions)))

        # Next, we need to take the cursor that measures the start position we're
working from
        # and connect it to each window_zip[i], right shifted by i positions
        for i in range(window_size):
            for j in range(i):
                nengo.Connection(processing.zero,
processing.window_zip[i].input[j*dimensions:(j+1)*dimensions])
            for j in range(window_size):
                nengo.Connection(processing.window_cursor.output[j],
processing.window_zip[i].input[(i+j)*dimensions:(i+j+1)*dimensions],
transform=np.ones((dimensions, 1)))
            for j in range(i+window_size, window_size*2-1):
                nengo.Connection(processing.zero,
processing.window_zip[i].input[j*dimensions:(j+1)*dimensions])

        # Connect up convolutions
        actionstr = ['convolutions_0 = inputs_0 * inputs_1'] + \
                    ['convolutions_{} = outputs_{} * inputs_{}'.format(i, i-1, i+1)
for i in range(1, window_size-1)]
        processing.actions = spa.Actions(*actionstr)
        processing.cortical = spa.Cortical(processing.actions)


        for i, pz in enumerate(processing.phrase_zip):
            # Connect input phrase to first mask
            nengo.Connection(processing.phrase,
pz.input[dimensions*phrase_length:])
            # each phrase mask should map onto one vector in masked_phrase
            for j in range(phrase_length):
                for k in range(dimensions):
                    nengo.Connection(pz.output[j*dimensions+k],
processing.masked_phrase[i*dimensions+k], function=prod)

            # Copy output of first mask to input of each window mask
            for j in range(window_size):
```

```
                nengo.Connection(processing.masked_phrase,
processing.window_zip[j].input[(window_size*2-1)*dimensions:])

        # Connect window zips to inputs and convolutions to outputs
        nengo.Connection(processing.scramble[0], processing.inputs[0].input,
transform=left_transform)
        for i, wz in enumerate(processing.window_zip):
            if i > 0:
                nengo.Connection(processing.scramble[i],
processing.inputs[i].input, transform=right_transform)
                if i < window_size - 1:
                    nengo.Connection(processing.convolutions[i-1].output,
processing.outputs[i-1].output, transform=left_transform)
            for j in range(window_size*2-1):
                for k in range(dimensions):
                    nengo.Connection(wz.output[j*dimensions+k],
processing.scramble[i][k], function=prod)


        # we need to mask off convolutions that won't be used
        for i in range(window_size-1):
            nengo.Connection(processing.conv_zip.output[i],
processing.conv_mask[i], function=lambda x: max(*x))
            if i < window_size - 2:
                nengo.Connection(processing.conv_mask[i],
processing.conv_zip.input[1][i+1])

        # Enable all positions from the location of phi onwards
        # if the window starts at phi (cursor is in position -1 or -2), all
subsequent values should be added
        nengo.Connection(processing.window_cursor.output[:-1],
processing.conv_zip.input[0][::-1])
        nengo.Connection(processing.window_cursor.output[-1],
processing.conv_zip.input[1][0])

        # Connect convolutions to sum zip
        for i, conv in enumerate(processing.convolutions):
            nengo.Connection(conv.output, processing.sum_zip.input[i*dimensions:
(i+1)*dimensions])
            nengo.Connection(processing.conv_mask[i],
processing.sum_zip.input[(window_size-1+i)*dimensions:(window_size+i)*dimensions],
transform=np.ones((dimensions, 1)))

            for j in range(dimensions):
                pass
                #nengo.Connection(processing.sum_zip.output[i][j],
processing.sum.input[j], function=prod)

    return processing

# this one's more complicated because you need to dynamically reroute data
depending on
# the cursor position
def processing_block_items(dimensions, phrase_length):
    processing = spa.SPA('item vectors processing block')

    with processing:
```

```
        ##### declare elements
        # inverts cursor for use in mask
        processing.inverter = inverter(phrase_length)
        # zips inverter+stopwrds to make a mask for the phrase
        processing.mask_zip = zipper(phrase_length, label='mask zipper')
        # zips input phrase with mask
        processing.phrase_zip = zipper_array(dimensions, phrase_length,
label='phrase zipper')
        # stores phrase representation before sum
        processing.phrase_state = state_array(dimensions, phrase_length,
label='phrase')
        # summed phrase elements
        processing.sum = spa.State(dimensions)

        ##### wire critical connections
        # setup mask
        nengo.Connection(processing.inverter.output, processing.mask_zip.input[1])

        for i in range(phrase_length):
            # connect mask to phrase zipper
            nengo.Connection(processing.mask_zip.output[i],
processing.phrase_zip.input[i*dimensions:(i+1)*dimensions], function=prod,
transform=np.ones((dimensions, 1)))
            # sum vectors in phrase
            nengo.Connection(processing.phrase_state.output[i*dimensions:
(i+1)*dimensions], processing.sum.input)

        for i in range(phrase_length*dimensions):
            # apply mask to phrase
            nengo.Connection(processing.phrase_zip.output[i],
processing.phrase_state.input[i], function=prod)

        ##### declare interface
        ### inputs
        processing.stops = processing.mask_zip.input[0]
        processing.cursor = processing.inverter.input
        processing.phrase = processing.phrase_zip.input[dimensions*phrase_length:]

        ### outputs
        processing.output = processing.sum.output

    return processing

# includes holo, memory normalizer, phrase normalizer
# this is small but important
# defines interface:
#   inputs
#       encoding.episode: inputs episode to be encoded
#       encoding.memory: inputs vector to be updated
#   outputs
#       encoding.hologram: outputs updated vector
#       encoding.control_vector: outputs normalized phrase vector
def encoding_block(dimensions):
    encoding = spa.SPA('encoding block')

    with encoding:
        ##### declare elements
```

```
        # normalize memory and episode so we don't get unbounded vector growth
        encoding.episode_normalize = normalizer(dimensions, label='episode
normalize')
        encoding.memory_normalize = normalizer(dimensions, label='memory
normalize')

        # a hologram representing the updated state of the vector
        encoding.holo = spa.State(dimensions)

        ##### wire critical connections
        # weight inputs respectively by 1-retention and retention, so we get a
magnitude of 1
        # retention describes the rate of forgetting old memory, high retention is
slower to forget
        nengo.Connection(encoding.episode_normalize.output, encoding.holo.input,
transform=1-retention)
        nengo.Connection(encoding.memory_normalize.output, encoding.holo.input,
transform=retention)

        ##### declare interface
        ### inputs
        encoding.episode = encoding.episode_normalize.input
        encoding.memory = encoding.memory_normalize.input

        ### outputs
        encoding.hologram = encoding.holo.output
        encoding.control_vector = encoding.memory_normalize.output

    return encoding

model = spa.SPA('BEAGLE')
with model:
    ##### Declare modules
    # takes in the chunk of ready file to be read
    # outputs concatenated list of vectors, binary list of stop words, chunk of
ready file for the phrase
    model.input = nengo.Node(size_in=1, size_out=((dimensions + 2) *
phrase_length), output=process_input)
    # responds to control signal from cursor+input by retrieving appropriate values
from memory
    # takes [phrase word ids] + [cursor] + [holo], yields [target word]
    model.memory_access = nengo.Node(size_in=phrase_length*2+dimensions,
size_out=dimensions+1, output=memory_access)

    model.encoding = encoding_block(dimensions)
    model.processing = processing_block_items(dimensions, phrase_length)
    model.control = control_block(dimensions, phrase_length)

    model.order = processing_block_order(dimensions, phrase_length, window_size,
phi, left_perm, right_perm)

    ##### Connect modules
    ### Connect inputs
    # connect visual vectors to processor input
    nengo.Connection(model.input[:dimensions*phrase_length],
model.processing.phrase)
    # connect stop words to processor input
```

```
    nengo.Connection(model.input[dimensions*phrase_length:
(dimensions+1)*phrase_length], model.processing.stops)
    # send the list of items in the phrase to memory access so it can get the
relevant item vectors
    nengo.Connection(model.input[(dimensions+1)*phrase_length:],
model.memory_access[:phrase_length])

    ### Connect memory access
    # current target word to be updated
    nengo.Connection(model.memory_access[:-1], model.encoding.memory)
    # next sentence signal
    nengo.Connection(model.memory_access[-1], model.control.next_phrase)
    nengo.Connection(model.memory_access[-1], model.input)

    ### Connect control block
    # tell memory access which word to select from phrase
    nengo.Connection(model.control.cursor,
model.memory_access[phrase_length:phrase_length*2])
    nengo.Connection(model.control.cursor, model.processing.cursor)

    ### Connect processing block
    nengo.Connection(model.processing.output, model.encoding.episode)

    ### Connect encoding block
    nengo.Connection(model.encoding.control_vector, model.control.control_vector)
    nengo.Connection(model.encoding.hologram,
model.memory_access[phrase_length*2:])
```