UNIVERSITY OF WATERLOO
Department of Systems Design Engineering

Lambda Calculus, Enumeration, and Neurons

SYDE 556: Final Project

prepared by

Alexander Huras

Table of Contents

Abstract

Given that symbolic processing can be modelled in the activities of spiking neurons, this report explores a novel application that involves the use of a mathematical formalism that has been around for quite some time: the lambda calculus. By treating anonymous function application and abstraction as generic semantic binding, a novel model of Church Enumeration is proposed. This model integrates NEF semantic binding patterns with a Basal Ganglia/Thalamus control circuit, and is designed to compute (severely limited) generic recursive programs such as "Count TWO more than ONE". This works in theory, but ultimately performs poorly in simulation, as the model can easily get lost after a few internal 'iterations'. We hypothesize that this is likely a limitation in the model, but entertain the prospect that the class of errors present in the model's design is similar to the class of errors that would be present in a successful model.

Chapter 1

Background

At the highest level (literally and figuratively), intelligence is the capability to abstract and reason. One of the many thoughts on how this could be conducted (either computationally or otherwise) is through the use of *symbolic processing*, which is to say: the expression of computation through formalisms regarding abstract constructs and concepts. This project began as a sort of *vision quest* into the wild and wonderful world of programming language design, and particularly, the methods that have been devised in order to facilitate the *expression* of computation. Through trials and tribulations the idea of expressing computation as a graph of neuromorphic transformations, as symbols, and as lambda expressions were brought together into the bizarre but oddly satisfying model that ultimately can perform Church Enumeration, poorly, under most circumstances.

This chapter is broken up into two major components—the first of which can be considered a superficial look at the structure and application of lambda calculus, specifically in relation to Church encoding. The second consists of a high-level overview of the geometric architecture that enables general symbol processing within the constraints of the Neural Engineering Framework (NEF) algorithm.

1.0 Lambda Calculus

The lambda calculus arose from the study of 'functions as rules' [1], in particular the investigation of how functions could be used to exhibit/formalize theories in logic, and foundational mathematics. Putting aside some of the more formal discussion of its roots, the lambda calculus remained an obscure formalism until the

1960s when more 'useful' mathematical semantics were found, and it became synonymous with computer language theory, and as a means of expressing computation. Since then it has seen extensive application in linguistics, and computer science [2], and as demonstrated in [3]; is Turing complete.

Within the context of this report it is not necessary to fully understand the subtleties involved in expressing computation in a *language* of anonymous abstractions and applications. However it is helpful to have an understanding of the underlying syntax, as it is used to define the symbolic *geometry* used within our model.

Definition From [1]: The alphabet of the lambda calculus takes the left and right parentheses, the dot '.', the symbol 'λ', and an infinite set of variables. The class of 'λ-terms' (valid lambda calculus expressions) are defined inductively as follows:

1. Every variable is a λ-term.
2. If t and s are λ-terms, then so is (ts) (*application*).
3. If t is a λ-term and x is a variable, then $(\lambda x.t)$ is a λ-term (*abstraction*).

Ultimately, there are only two things we can do in lambda calculus, abstract, and apply, and that is enough, furthermore it can be shown that all functions can be expressed as compositions of unary functions (not shown here), and that lambda calculus is turing complete [3].

1.1 Church Numerals

In mathematics, Church encoding/Numerals (hereafter the 'Numerals') represent data and operators in the lambda calculus. Specifically within untyped lambda calculus, it has been shown that by mapping more primitive types into higher-order functions, all computable operators (and arguments) can be represented under Church encoding [1]. This is of particular significance within the realm of symbolic processing, since Church encoding provides a general form of representing the natural numbers (for which there exist a great many other representations within the lambda calculus). The canonical ostensive definition of the Numerals is contained within Table 1.1.

Number	Lambda Expression	Function Definition
0	$\lambda f.\lambda x.x$	0 f x = x
1	$\lambda f.\lambda x.f\ x$	1 f x = f x
2	$\lambda f.\lambda x.f\ (f\ x)$	2 f x = f (f x)
3	$\lambda f.\lambda x.f\ (f\ (f\ x))$	3 f x = f (f (f x))
$\vdots$	$\vdots$	$\vdots$
n	$\lambda f.\lambda x.f^n\ x$	n f x = $f^{(n)}x$

Table 1.1: Canonical Definition of the Church Numerals

Of note within the definition of the Numerals is that ultimately, the numeral represents the action of repeated function application (of the *ZERO* numeral, which represents *not* applying the function).

Given the ostensive definition of the Numerals as recursive *application* of an arbitrary function $fx \equiv \lambda f.\lambda x.x$ a formal definition of arithmetic functions can be designed. Complete tables and derivations of which can be found in [1] or [4]. Within the context of this report, we will focus on the relatively simple definition of succesorship (and hence, addition) under Church Numerals.

Intuitively, the concept of succession within the natural numbers is quite simple: we start from one number, and count *one-higher*. Algebraically, successorship within the natural numbers is simply adding 1. Within the realm of Church enumeration, we can see this relationship expressed as applying the function f to itself *one-more time*.

Similarly, the concept of *adding* two numerals m, n, is equivalent to the repeated *enumeration* of n, m-times. This relationship is defined more explicitly in Table 1.2.

Method	Algebra	Function Definition	Lambda Expression
Successor	$n+1$	$succ\ n\ f\ x = f\ (n\ f\ x)$	$\lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$
Addition	$n+m$	$plus\ m\ n\ f\ x = m\ f\ (n\ f\ x)$	$\lambda m.\lambda n.n\ succ\ m$

Table 1.2: Definitions of Succession and Addition under the Church Numerals

The models depicted in this project aim to provide an evaluation/execution mechanism for the processing of Numeral-like computation under the vector symbolic architecture.

2.0 Semantic Binding as Abstraction and Application

As discussed in Section 1.0 , arbitrary computation can be defined in terms of the construction of arbitrary lambda-expressions/terms. In Section 1.1, we outlined a simple encoding scheme that relies on enumeration to ostensively define the concept of natural numbers, specifically within the context of *iteration*. Within this section we'll look at how algebraic functions can be computed symbolically via the Church Numerals through the use of semantic *binding*, this idea is extended to the more general sense where purely functional data structures and algorithms can be defined (all in lambda calculus) symbolically, but this last point is out of scope for this project.

2.1 Semantic Binding

As stated in [5], binding using vector representations has been addressed through the use of a family of approaches collectively under the umbrella of "Vector Symbolic Architectures" (VSAs) [6]. VSAs introduce an operation that combines (binds) two vectors creating a third that is geometrically distinct from either of the pair, furthermore this operation is invertible, allowing recovery of the 'bound' vectors through an inverse operation.

Within the context of this paper, circular convolution $\otimes$ was used to bind vectors, and circular correlation $\oslash$ as an approximately inverse process. Symbolic binding and retrieval allows for the construction of idealized structures (akin to abstract data structures in computer science) for use in arbitrary computation. Furthermore it has been demonstrated that neural models involving such structures can (with a reasonable amount of accuracy) be modelled in the NEF for linguistic processing—a fairly sophisticated task. Within this paper, we focus on much simpler structures, and the space that they occupy.

2.2 Symbolic Church Numerals

Given that we've defined Church Numerals, it is now important to note the importance of the zeroth Numeral. Within Church Encoding, there is no special *value* associated with the zero-ith numeral (functionally: f, or canonically $\lambda f.\lambda x.x$), it is simply a lambda term representing a function that has not been applied to any arguments. In the vector-symbolic sense, it is a *symbol* that for all intents and purposes is *unbound*. Analogous to how each successive Numeral was defined through successive applications of the zero-ith numeral, each successive *symbol* is defined through successive *binding* of the zero-ith or *root* symbol.

Numeral	Symbol	Symbolic Definition
0	$ZERO$	$ZERO$
1	ONE	$ZERO \otimes ZERO$
2	TWO	$ZERO \otimes ZERO \otimes ZERO$
$\vdots$	$\vdots$	$\vdots$
n	N	$ZERO_0 \otimes \ldots \otimes ZERO_n$

Table 1.3: Symbolic Numerals via VSA Binding

As shown in Table 1.3, there is a relatively natural translation from pure lambda application to symbolic binding–note the use of N to describe the symbolic representation of the arbitrary natural number/numeral n. This representation also illuminates some interesting properties of the scheme (as well as on Church Numerals in general), in that the $ZERO$ symbol is interpreted as representing a *function*, rather than a value; in that by *binding* via $\otimes$ we are *applying* the function $ZERO$ to an argument. The lack of distinction between functions and values is one of the primary characteristics of lambda calculus, (and pure functional programming in general). What is somewhat interesting, is that it ultimately doesn't matter whether or not we treat a particular symbol as a function or data, so long as our schemes are internally consistent.

Similar to Table **??**, we can define a set of symbolic functions a-la the lambda calculus defined in terms of binding rather than composition. Through the existence of the inverse binding (or *extraction*) $\oslash$, we also get subtraction (predecessorship) for free, so long as it is not applied to $ZERO$. This is a serendipitous property of symbol binding which otherwise requires a more involved definition of predecessorship (and thus subtraction) via pure lambda calculus and our symbolic analog. Suffice to say that in the numerals $pred(0) = 0$, while algebraically $ZERO \oslash ZERO \neq ZERO$, this is ultimately an implementation detail that must be observed to reap the associated benefits of not performing arbitrary subtraction via full-blown lambda predecessorship.

Method	Algebra	Symbolic Interpretation
Successor	$n+1$	$ZERO \otimes N$
Addition	$n+m$	$ZERO_1 \otimes \ldots \otimes ZERO_m \otimes N$
Predecessor	$n-1$	$ZERO \oslash N \; \forall N \neq ZERO$
Subtraction	$n-m$	$ZERO_1 \oslash \ldots \oslash ZERO_m \oslash N \; \forall n > m$

Table 1.4: Arithmatic on Symbolic Numerals under VSA Binding

Chapter 2

System Design

As with any vaguely neuromorphic system, the model was architected to contain some of the canonical cortical structures found within the mammalian brain. Specifically, regions well suited to massively parallel concurrent computation are delegated to the pseudo region entitled ''Cortex''. The information contained within the Cortex is funnelled into the Basal Ganglia (another artifact from biology), which handles the prioritization of particular workloads (a.k.a. action selection). After which the cortical regions are updated with the processed work and the cycle is closed.

Within this particular model the inputs are not to be interpreted as sensory (they are still representing symbols in the same *space* as the working components), but could be considered to be outputs from similarly architected systems (such as an as-yet unbuilt generic neuromorphic lambda calculus engine), and similarly the system output should be given the same consideration.

1.0 Goals and Motivation

Implementing a general symbolic lambda calculus machine in neurons is a relatively challenging proposition, both in terms of simulation expense, but also architecturally, and to do so in a biologically plausible manner requires a degree of symbolic processing knowledge that is outside of the scope of this project. Instead, we've focused on the design of a so-called 'Numeral Processor', which is capable of performing arbitrary iterative computations (in a manner not unlike a von Neumann machine) on the types of symbols described in Table 1.3.

A simple model, that still retains significant expressive power is desired. The Numeral Processor designed and implemented thus performs addition congruent to Table 1.4, and implicitly performs subtraction in the same manner.

A general system map is shown in Figure 2.1, each module or *block* will be discussed to some depth in the following subsections.

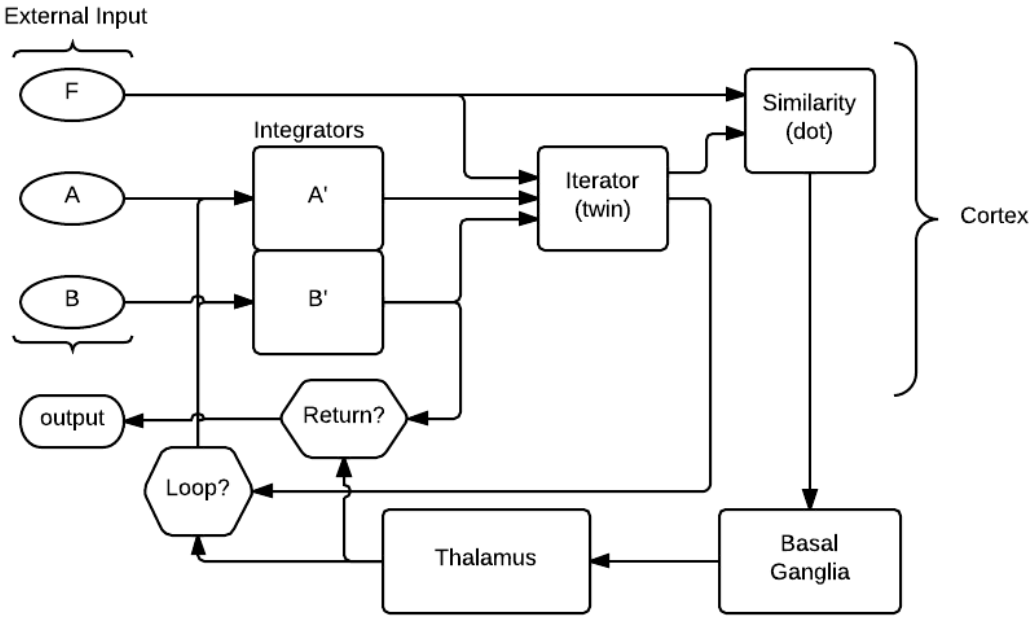

Figure 2.1: Functional overview of the prototype Numeral Processor, as implemented in the NEF. The system performs the Numeral computation $A + B$, by successively applying $B' = F \otimes B$, and $A' = F \oslash A$ until $A' \cdot F$ is sufficiently small. This is analogous to decrementing A via $\oslash$ until it is zero, after which returning B'

Ultimately, the system *iterates* the computation performed in the 'Iterator' component in Figure 2.1 until the value of A' becomes indistinguishable from $ZERO$ (which in the general sense can be attributed to any arbitrary symbol F, but in the case of Numeral operations: $ZERO$).

The system receives its sensory input directly as Numeral symbols which are congruent under $\otimes$ (i.e. the computation will not converge if $ZERO$ is not in the set of Predecessors of A), after which the system iterates until it perceives through the 'Similarity' block that the computation has converged. At this point the utility associated with opening the output gate exceeds that of keeping it closed, or continuing the computation, and a result symbol is returned to the 'output' channel.

1.1 Model Algorithm

Through successive function application, the model performs Numeral addition of A and B, this process is described below:

Data: A, B, F, s.t. $A \in succ_n F$
while $A \neq F$ **do**
  $A := pred\ A$;
  $B := succ\ B$
end
Result: B

Algorithm 1: Addition Under Numerals

Where *pred* and *succ* are defined under symbolic Numerals in Table 1.4.

2.0 Cortex

Nontrivial cortical components are outlined in the following section. It should be noted that the model relies heavily on the neural integrator (detailed in [5, 7]), which is used as working memory for the model, storing the current values of A' and B'.

2.1 Iterator

This component performs the actual computation associated with a given Numeral iteration. The iterator transforms the input symbols A, B into their next state A', B' by way of Eqs (2.1) and (2.2).

$$A' = F \oslash A \tag{2.1}$$

$$B' = F \otimes A \tag{2.2}$$

By using VSA, the model is able to make use of the NEF Circular Convolution Ensemble, which is featured in Fig. 2.2.

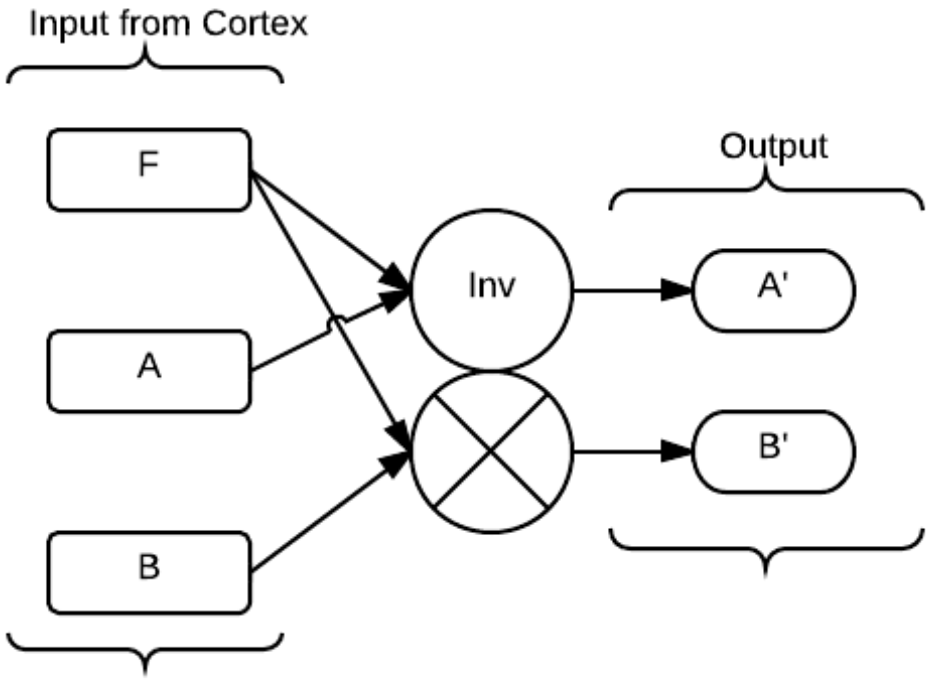

Figure 2.2: The Iterator simultaneously computes $A - F \equiv F \oslash A$ and $B + F \equiv F \otimes B$ through the use of Circular Convolution Ensembles

2.2 Are We There Yet?—Assessing Similarity

The control pathways of this model rely heavily on the similarity between A', and the goal state: F (colloquially $ZERO$ in the addition case). Symbolically, such similarity can be expressed as the dot-product between the two vectors representing A' and F respectively, and so the 'Similarity' block in Fig. 2.1 does just that. Explicitly, it implements $A \cdot B = \sum_i^d A_i B_i$.

This is necessary to determine (in a very general sense), whether the model can consider itself finished iterating. While at the core, this component controls whether or not to continue iterating, its value is transformed into twin utility scores which are used as input to the Basal Ganglia. Furthermore, given the large amount of noise in the system (due to simulation constraints) it is unlikely that similarity scores (dot products) will be fairly high. To combat this, the utility functions Q_{loop} and Q_{return} were designed to place a threshold on what the model considers 'close enough'. This was largely done experimentally to control when, and for how long a given loop-cycle can take.

The NEF transformation function was designed to exploit the ensembles saturation, while also setting the decision boundary for what the model considers 'similar'. Specifically, given the similarity score $s = A \cdot F$, the utility associated with continuing to iterate $Q_{loop} = 10s - 2.5$ and the complement utility (associated with returning) is simply $Q_{return} = -Q_{loop}$.

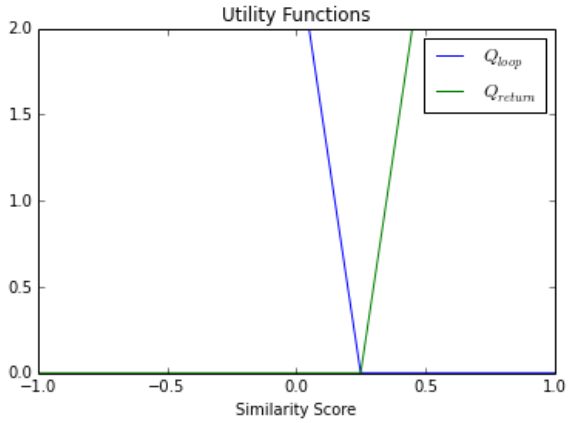

Figure 2.3: Sharp linear utility functions. Everything with a similarity score higher than 2.5 triggers a function return

3.0 Action Selection and Execution

The line between action selection and execution is somewhat blurred in this model, specifically because cortical regions continue to perform computation, and the Thalamus/Basal Ganglia (BG) loop simply controls whether the system output is *visible*.

3.1 Basal Ganglia

Given that the resulting behavior (either keep looping, or expose the output) is a function of the similarity between the current, and goal states, it was necessary to introduce a sort-of binary control circuit that operates in a similar fashion to discrete logic components. Central to this idea of logical control flow is the 'selection' of a particular action to take, the canonical application for the Basal Ganglia (both biologically, and in the NEF).

The Basal Ganglia receives a transformed version of the output of the Similarity block, which is essentially bifurcated into "similarity", and "dissimilarity", which are interpreted as the utility associated with continued iteration, and the utility associated with returning a value to the output.

The BG circuit used was the de-facto standard BG model in Nengo 2.0 [7], which is an adaptation of the model proposed by Gurney, Prescott and Redgrave in [8].

3.2 Thalamus

The Thalamus as used in the NEF is typically associated with action execution, and control flow. Within the design outlined in Fig. 2.1, the Thalamus as a component is simply a unit which translates the inhibitory signals from the Basal Ganglia into a 'one-hot' representation.

This each dimension of the 'one-hot' vector output from the Thalamus is fed into the associated *gate*, in a manner similar to scalar multiplication. This in essence turns the gate components into a continuous version of a logical 'AND' gate, by using a population of 'positive' neurons whose output can be heavily suppressed/inhibited by the control signal emanating from the thalamus.

The "Loop?" gate in Fig 2.1 controls whether the enumerated values A' and B' are fed back into their respective integrators. Similarly, the "Return?" gate is maintained in the opposite state as the "Loop?" gate, and controls whether the current B' value is projected to the output. Ideally, once A' is similar to F (as defined in the Similarity block), the integrators will maintain their respective values (subject to drift). Thus once the enumeration has concluded, we expect the output value to slowly degrade as well.

Chapter 3

System Specification

Given the use of Leaky Integrate and Fire (LIF) neurons, the individual neuron populations in the model were generally left at their default values. The post-synaptic time constants throughout the system are loosely based on the expected spatial layout of the system, which loosely maps to the physical layout of the brain. Specifically, this means that the post-synaptic time constants τ between the basal and cortical regions are relatively high (on the order of 8 ms), while the intercortical τ values hover around 5 ms.

Similarly, the neurons in the Basal Ganglia circuit were left with their default values which are accessible in [7], and described in more detail in Table 3.1. In general, the BG circuit involves neurons that are sensitive to high-frequency input (smaller post-synaptic time constants), but slower to inhibit, with peak firing rates in the ballpark of 100 Hz.

1.0 Language Processing as Symbolic Processing

In order to design a biologically *realistic* system it is necessary to draw critical biological parameters from neuroscience. However, our goal was to put forward a biologically *plausible* mechanism for abstract enumeration, and thus in some sense, we found useful neural properties by projecting the model back on the brain. This was done by abstractly modelling the form of information processing the model is performing, and draw analogies between that and cognition.

Lambda calculus is Turing complete, it can thus represent *all* effective computation, however it is not

sufficient to simply illustrate that the brain computes, and move on. In particular, the Numeral Processing model we have described performs iterative symbolic manipulation, a form of computation closely language to language processing. Suffice to say that the computation is being performed on a very high/abstract level, and it is likely that similar computation is performed in regions connected to equally abstract thought—language processing being a prime example.

2.0 Symbolic Constraints

Given that vector-symbolic representations require suitably high-dimensional vectors for representation, it is necessary to balance the desire for a sufficiently high-dimensional vector representation, with the representational error associated with a fixed neural population. In [10] NEF ensembles in a circular convolution network were shown to exhibit significant noise (and thus error), and that 50-100 neurons/per dimension were required for successful symbol parsing. However, due to significant hardware limitations, the Numeral Processing model had to operate in the ballpark of roughly 32-neurons per dimension (for the Cortex units).

2.1 Wernicke's Area

A brain area commonly associated to language processing is Wenicke's Area (thought to be located in the posterior part of Brodmann Area 22) [9]. However, within the scope of this project (particularly given the fidelity of the model), it is unlikely that matching the neuron properties of the cortical areas of the system to Type I, and II neurons in Wernicke's area will offer any outstanding functional benefit—a problem that is addressed by referencing existing models in similar domains, but particularly those using the NEF.

2.2 Component Parameters

Given the the simultaneously low availability of LIF parameters (most literature on Wernicke's area models using Hodgekin-Huxley), and the relative lack of direct applicability (after all, we're modelling general abstract computation, not necessarily language—although the similarity is evident) to Wernicke's area, model parameters were sourced from existing prototypes operating in a similar domain. Thus the neuron properties were matched on a per-component basis using reference values from [10] which deals with a similar symbolic domain (linguistic parsing), and [5] which indicates that 'state' values (symbols representing a given goal, or process) can be effectively represented with 128-dimensions, with approximately 20-neurons per dimension.

Component	Dimensions	Number of Neurons	Encoder	Rates	τ
Integrators (x2)	128	4096	random	40-200 Hz	10 ms
Iterator	128 x 2	8192	random	40-200 Hz	5 ms
Similarity (Map Phase)	128	2560	random	40-200 Hz	2 ms
Similarity (Reduce Phase)	1	64	[1]	40-200 Hz	8 ms
Basal Ganglia (excite, inhibit)	2	600	[1]	0-100 Hz	2, 8 ms
Thalamus Actuator (x3)	1	64	[1]	0-100 Hz	2 ms
Thalamus Gate (x3)	128	4096	random	40-200 Hz	8 ms

Table 3.1: Specification of Neuron Parameters by Component

Fig. 3.1 outlines the various model parameters. In general, the model uses 128-dimensional vectors to represent the Numerals (A, B, and F in Fig 2.1), and all associated specialized components (Similarity, and Iterator) use 32-neurons per dimension.

Chapter 4

Implementation

The model was implemented in Nengo 2.0 [7], and as such utilized many template components. Among those were the high-dimensional integrators, the circular convolution (and associated inverse) block, and the Basal Ganglia circuit. Within this chapter, the design and implementation of custom NEF components are detailed and explained. This includes the Thalamus-like 'gates', which are currently unavailable within Nengo 2.0, and the Similarity module which at its core computes the dot-product between two vectors without exploding the dimensionality of a particular ensemble.

1.0 Determining Similarity

The Similarity component is designed to ascertain the relative alignment of two arbitrarily high-dimensional inputs. Given two vectors/symbols A, and B, the Similarity component returns the vector dot-product $A \cdot B$.

When computing functions that are the result of multiple input values, a common pattern within the NEF is to construct an intermediary group of neurons, that is twice the dimensionality of each input vector, and then decode out the desired result, a rather expensive proposition on slow hardware. For particularly high-dimensional values this can also cause sparsity problems (i.e. curse of dimensionality). Admittedly, on the scale of this simulation (128-dimensional), simply 'exploding' the dimensionality and taking the traditional path is entirely possible and reasonable. However, in the interest of generality, we implemented an abstract element-wise operator pattern, which consists of generating a small ensemble of neurons (an Ensemble Array) for each common input subspace/dimension of the two inputs, and symmetrically computing an arbitrary

function within each sub-ensemble—in the case of a dot product this equates to a simple multiplication. The scalar output of each sub-ensemble is then connected directly to a single small 'reducer' ensemble, which simply implicitly evaluates the sum.

In our scenario involving the computation of a dot product, the fidelity of the element-wise operators (low dimensional) can be increased dramatically through the use of evenly spaced encoders, a luxury that can in some situations enable us to use fewer neurons-per-dimension, decreasing the computational burden of the operation.

2.0 Thalamus Gates

In a similar style to [5], the Thalamus system is comprised of an ensemble of neurons with strictly positive encoders. Thus, when receiving inhibitory input from the BG circuit, they essentially turn off, however when the inhibitory signal is lifted, they chatter away happily (one can imagine). The end goal of this roundabout method is transforming the inhibitory BG signal into a 'one-hot' representation.

Looking at a single dimension of the 'one-hot' signal, the signal is *broadcast*, or *mapped* to a BinaryElementwiseOperation array as defined in Appendix B, which acts as a scalar multiplication circuit, where the control signal (one-hot) is multiplied with each dimension of the controlled signal. In the case of the Thalamus gates, these control when the integrators are refreshed with new state. A probe of a gated integrator input from the model simulation illustrates this effect in Fig. 4.1.

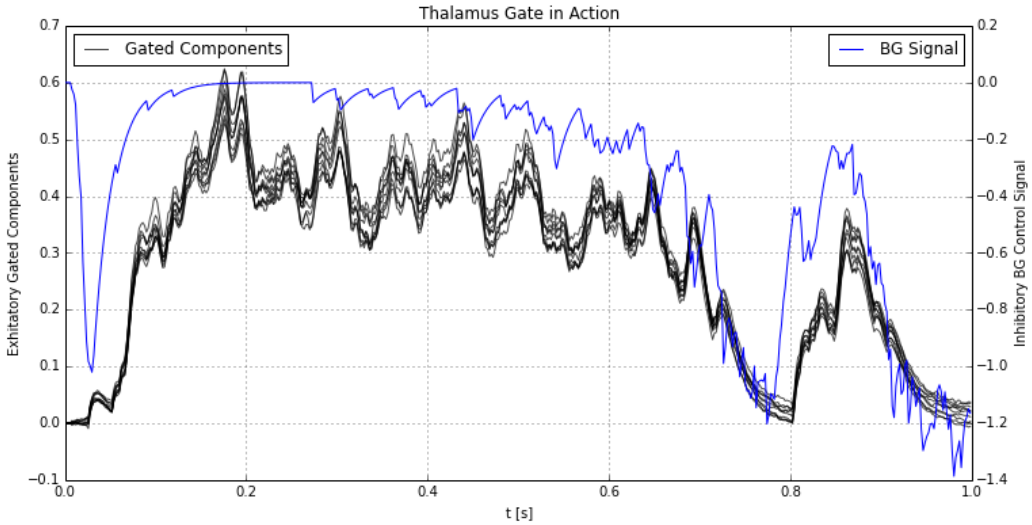

Figure 4.1: Thalamus Gate: The Inhibitory control input is transformed to scalar multiplication.

Chapter 5

Simulation

While it is all well and good to propose a high-dimensional symbolic processing network, it is another entirely to run it. Through the usage of Nengo 2.0—which is at this moment still quite young, we were unable to perform even moderate scale simulations—specifically building simulators with many ensembles. In order to perform simulations with a shred of interactivity (without hour-long 'compiling breaks'), the number of dimensions had to be reduced to 10 (for the symbolic components) it was found that this was due to a severe limitation with the graph reduction operations performed by the underlying model builder (which was compounded through our extensive use—perhaps abuse, of EnsembleArrays), however the number of neurons in the model was not a limiting factor, and so they were maintained at 52-neurons per dimension. However, even under these conditions the Similarity, Basal Ganglia, and Thalamus Gate neural circuits performed well.

Throughout the simulation, the model had a hard time differentiating between symbols generated from circular convolution. In particular, the similarity between (for example), the true $ZERO$ symbol, and the identity equivalent $ZERO \oslash ONE$ was marginal at best.

1.0 Runaway Error

Given that the 'stopping criterion' is rarely met in simulation (we never truly 'find' $ZERO$), in fact if left long enough, the integrators will drift until they become dissimilar to $ZERO$, at which point the model starts looping until it (through essentially unguided random exploration) transforms A into something similar to

$ZERO$ and returns.

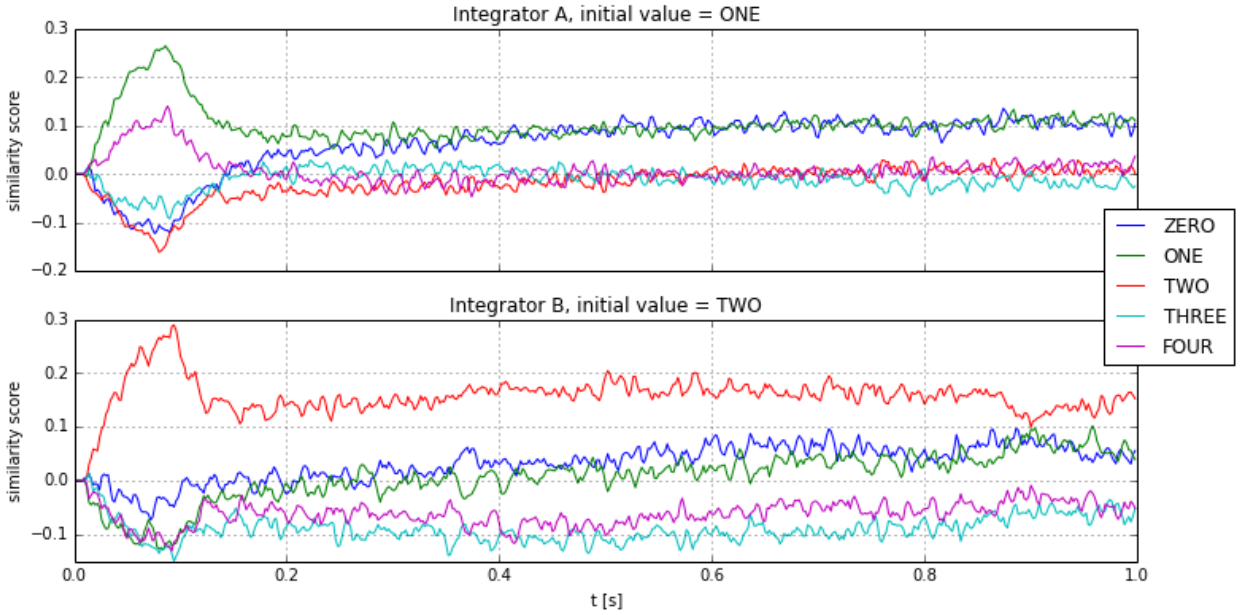

Figure 5.1: The internal state of the model's integrators, as measured by similarity with preprocessed symbols.

The interpretation of the symbolic state of the integrators in the model is conducted by computing the dot product between a set of prepared semantic pointers. The similarity score is then plotted over time to differentiate the exposure of a particular symbol. This is shown in Fig. 5.1. Note the path of the 'most-similar' symbol in the plot for the A integrator: starting from an initial value of ONE the integrator began representing a state half-way between ONE and $ZERO$ rather than simply shifting to $ZERO$. Oddly enough, this behavior was replicated to a greater extent with the integrator for B, which was expected to transition from TWO to $THREE$, however possibly due to representational error associated with non-orthogonal symbols, it picked up components of $ZERO$ along with the desired $THREE$, as well as not reducing its grasp of TWO. It is likely the recurrent time constants associated with the integrators could be tuned to reduce the likelihood of being caught in a *transient* state.

The relationship between the utility scores and associated Basal Ganglia action suppression is shown in Figs. 5.2, and 5.3. Of particular interest is the indecision related to similar Q-values. During the simulation, the first peak of Q_{return} occurs near 0.7-seconds. At this time, the integrators are no longer being refreshed with the data that yielded a highly similarity between $ZERO$ and A, this leads to drift, followed by a response cycle into the 'loop' state.

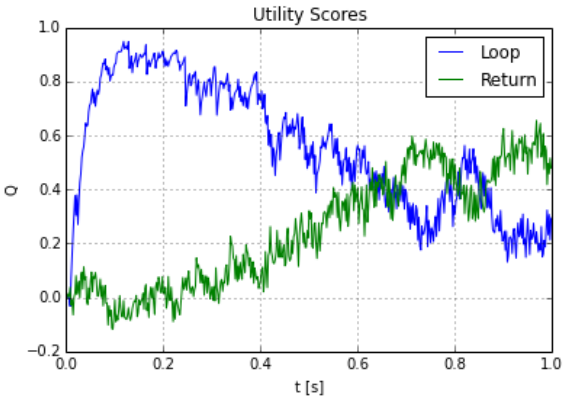

Figure 5.2: The utility values driving the Basal Ganglia.

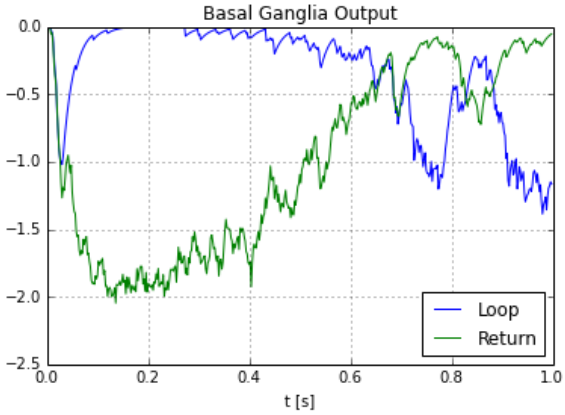

Figure 5.3: The Inhibitory output of the Basal Ganglia. Note the continued emphasis on Looping

Chapter 6

Conclusions and Future Work

Ultimately, the model as simulated does not possess the capability for arbitrarily deep Church enumeration. This is due both in part to the accumulation of error within the function application process (semantic binding), as well as the severely reduced fidelity that was required to simulate the model on aged hardware. The largest model that could be run pseudo-interactively was on the order of 3-thousand neurons—with a symbolic network this is not even close to enough, especially in a network designed to repeatedly apply what ends up being a very noisy operation (semantic binding/retrieval). As a result, the similarity between the numeral vectors could bleed into the calculation quite easily, skewing the accuracy. Interestingly, this manifested itself as the network deciding to 'return' a value, rather than an 'infinite loop'.

On a more philosophical note it is interesting to ask what a successful result would imply. If such a model were successful, it could open the door to posing arbitrary computational problems, and simulating in a biologically plausible way, a method by which they could be computed. Similarly, if the model proposed was successful, it could become a small symbolic unit for more complex algorithms. Instead of simple enumeration, data structures could be created and operated on, the only limit is the model's ability to comprehend them—which would be an interesting bit of information. Given that church enumeration is possible, and that successive semantic binding (function application) can occur without propagating too much error, it should be possible to perform arbitrarily complex computations.

The dream is to have a biologically plausible model implementing lambda calculus, which will demonstrate why understanding how such a model works is difficult.

Bibliography

[1] J. Alama, "The lambda calculus," in *The Stanford Encyclopedia of Philosophy*, summer 2014 ed., E. N. Zalta, Ed., 2014.

[2] I. Heim and A. Kratzer, *Semantics in Generative Grammar*, ser. Blackwell Textbooks in Linguistics. Wiley, 1998. [Online]. Available: http://books.google.ca/books?id=jAvR2DB3pPIC

[3] A. M. Turing, "Computability and lambda-definability." *The Journal of Symbolic Logic*, vol. 2, pp. 153–163, 1937.

[4] C. J. M. Kemp, "Theoretical Foundations for Practical 'Totally Functional Programming'," Ph.D. dissertation, University of Queensland, Nov. 2007. [Online]. Available: http://www.archive.org/details/TheoreticalFoundationsForPracticaltotallyFunctionalProgramming

[5] *Neural Cognitive Modelling: A Biologically Constrained Spiking Neuron Model of the Tower of Hanoi Task*. Austin, TX: Cognitive Science Society, 2011.

[6] R. W. Gayler, "Vector symbolic architectures answer jackendoff's challenges for cognitive neuroscience," *CoRR*, vol. abs/cs/0412059, 2004. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr0412.html#abs-cs-0412059

[7] T. Bekolay, J. Bergstra, E. Hunsberger, T. DeWolf, T. C. Stewart, D. Rasmussen, X. Choo, A. R. Voelker, and C. Eliasmith, "Nengo: A python tool for building large-scale functional brain models," *Frontiers in Neuroinformatics*, vol. 7, no. 48, 2014. [Online]. Available: http://www.frontiersin.org/neuroinformatics/10.3389/fninf.2013.00048/abstract

[8] K. Gurney, T. J. Prescott, and P. Redgrave, "A computational model of action selection in the basal ganglia. I. a new functional anatomy," *Biological Cybernetics*, vol. 84, pp. 401–410, 2001.

[9] T. C. Stewart, X. Choo, and C. Eliasmith, "Sentence processing in spiking neurons : A biologically plausible left-corner parser."

[10] J. E. Bogen and G. M. Bogen, "Wernicke's regionwhere is it?" *Annals of the New York Academy of Sciences*, vol. 280, no. 1, pp. 834–843, 1976. [Online]. Available: http://dx.doi.org/10.1111/j.1749-6632.1976.tb25546.x

Appendices

This section contains only original code.

For libraries used, refer to the complete repository at:

http://www.github.com/athuras/MentalMath.

Appendix A

model.py

The underlying model as implemented in Nengo 2.0.

```
import numpy as np
import nengo
import symbols
import constructs
import nengo.networks as networks
from nengo.utils.distributions import Uniform
import dot_product

# Local version of nengo.networks.integrator.Integrator
from integrator import Integrator

# Simulation Control, Nengo2 is slow on my machine ... really slow
D = 24
n_per_d = 32
N = D * n_per_d

# Base Symbols
ZERO = symbols.initial_zero(D)
model = nengo.Network()
```

```
numeral_table = dict(symbols.numerals(D, 4, zero=ZERO))
def hold_for_t(t, value):
    '''yields function that outputs value for time <= t'''
    default = np.zeros_like(value)
    def f(tau):
        return value if tau <= t else default
    return f

def symbolic_decode(numeral_table, x):
    similarities = {k: np.dot(v.v, x.T)
                    for k, v in numeral_table.iteritems()}
    return np.vstack([d for _, d in
                      sorted([(k, v)
                              for k, v in similarities.iteritems()])])

with model:
    A_in = nengo.Node(label='A',
            output=hold_for_t(0.05, 1.3 * numeral_table[1].v),
            size_out=D)
    B_in = nengo.Node(label='B',
            output=hold_for_t(0.05, 1.3 * numeral_table[2].v),
            size_out=D)
    # Held Constant
    Fun = nengo.Node(label='Fun', output=ZERO.v, size_out=D)

    # Integrators, in general, Fun, and Ref would be as well,
    # but it involves growing the model too much, KISS etc.
    A, B = [Integrator(recurrent_tau=0.1,
                       neurons=nengo.LIF(n_neurons=N),
                       dimensions=D,
                       radius=1.,
                       max_rates=Uniform(40, 200))
```

```
        for i in xrange(2)]


# Function Application
NG1 = networks.CircularConvolution(nengo.LIF(N),
                                   dimensions=D)
NG2 = networks.CircularConvolution(nengo.LIF(N),
                                   dimensions=D,
                                   invert_a=True)


# Connect the stuff
nengo.Connection(A.output, NG1.B, synapse=0.01)
nengo.Connection(B.output, NG2.B, synapse=0.01)
# simulate coming from integrator
nengo.Connection(Fun, NG1.A, synapse=0.01)
nengo.Connection(Fun, NG2.A, synapse=0.01)
nengo.Connection(A_in, A.input, synapse=None)
nengo.Connection(B_in, B.input, synapse=None)

Similarity = dot_product.MapReduceDotProduct(
                           leo_neurons=nengo.LIF(n_per_d),
                           leo_dimensions=D,
                           reducer_neurons=nengo.LIF(2 * n_per_d),
                           reducer_radius=1.2)
# Now we compare 'Ref' to A_prime out of NG1.
nengo.Connection(NG1.output, Similarity.A, synapse=0.002)

# The utilities for updating the integrators are based on 1 - similarity
# The return utility is simply the value of similarity.
# The idea is once we've become similar to the 'zero' symbol, we're done.
Q = nengo.Ensemble(neurons=nengo.LIF(n_per_d * 2),
                   dimensions=2,
```

```
                    radius=1)
def utility(sim):
    '''if similarity is high, utility associated
    with looping should be low'''
    x =  sim[0]
    z = np.abs(1 - x)
    return np.array([z, 1 - z])
_intermediary = nengo.Ensemble(neurons=nengo.LIF(n_per_d * 2),
                               dimensions=2, radius=1)
nengo.Connection(Similarity.output, _intermediary,
                 transform=np.ones((2, 1)), synapse=None)
# Necessary because Pre is a nengo.Node, and therefore can't
# be involved in function computing ...
nengo.Connection(_intermediary, Q, function=utility, synapse=0.002)


# Now for the Basal Ganglia, which will suppress the input to the integrators
# This only works because the integrators represent positive values (I think)
BG = networks.BasalGanglia(dimensions=2, output_weight=-3)
nengo.Connection(Q, BG.input, synapse=0.008)

# Generate the 'Gates', these are controlled by the 'iterate' action
A_gate, B_gate = [dot_product.ThalamusGate(signal, inhibit=BG.output,
                                           inhibit_transform=np.array([[1., 0]]),
                                           mapper_neurons=nengo.LIF(n_per_d),
                                           control_neurons=nengo.LIF(n_per_d))
                  for signal in (NG1.output, NG2.output)
                  ]

nengo.Connection(A_gate.output, A.input, synapse=0.008)
nengo.Connection(B_gate.output, B.input, synapse=0.008)
```

```
# Ignore the 'return state', too many neurons already ...

# System Output
Output = nengo.Node(label="Return", size_in=D, size_out=D)
nengo.Connection(B.output, Output)
```

Appendix B

dot_product.py

Constructs for MapReduce, Thalamus Gate, and Linear Element-wise operations that utilize nengo.EnsembleArray.

```
from nengo.networks.ensemblearray import EnsembleArray
from nengo.utils.network import with_self
from nengo.utils.distributions import Uniform
import nengo
import numpy as np

def corners(n):
    c = np.array([[1., 1], [-1., 1], [1, -1], [-1, -1]])
    return np.tile(c, (n // 4, 1))

class BinaryElementwiseOperation(nengo.Network):
    '''Handles splitting high-dimensions stuff into ensemble arrays
    which each handle one 'dimension' or 'element'.
    the x.output object is of the same dimensions as the inputs'''
    def __init__(self, neurons, dimensions, kernel=None,
                 input_transform=None):
        self.kernel = kernel if kernel is not None else self.product
        self.dimensions = dimensions
        self.A = nengo.Node(size_in=dimensions, label='A')
```

```
        self.B = nengo.Node(size_in=dimensions, label='B')
        self.ensemble = EnsembleArray(neurons,
                n_ensembles=dimensions,
                dimensions=2,
                radius=1,
                label='elementwise_operation')
        self.output = nengo.Node(size_in=dimensions, label='output')

        if input_transform is None:
            self.input_transform = self.elementwise_comparator_transform
        else:
            self.input_transform = input_transform



        for ens in self.ensemble.ensembles:
            if not isinstance(neurons, nengo.Direct):
                ens.encoders = corners(ens.n_neurons)
        nengo.Connection(self.A, self.ensemble.input, synapse=None,
                transform=self.input_transform)
        nengo.Connection(self.B, self.ensemble.input, synapse=None,
                transform=self.input_transform)
        nengo.Connection(self.ensemble.add_output('result', self.kernel),
                self.output)

    @property
    def elementwise_comparator_transform(self):
        I = np.eye(self.dimensions)
        return np.vstack((I, I))

    @staticmethod
    def product(x):
        return x[0] * x[1]
```

```
class MapReduceDotProduct(nengo.Network):
    '''A combination of LinearElementwiseOperation (with product),
    and a reducer phase that simply adds everything up'''
    def __init__(self, leo_neurons, leo_dimensions, reducer_neurons, reducer_radius):
        self.map_phase = BinaryElementwiseOperation(neurons=leo_neurons,
                                                    dimensions=leo_dimensions)
        self.input_dimensions = leo_dimensions
        self.output_dimensions = 1
        self.reducer = nengo.Ensemble(neurons=reducer_neurons, dimensions=1,
                radius=reducer_radius)
                #encoders=np.ones((reducer_neurons.n_neurons, 1)))

        # Even though the high-dimensional result is available
        # from LinearElementwiseOperation.output, we'll go directly
        # to the ensembles for free addition!
        nengo.Connection(self.map_phase.output, self.reducer,
                synapse=0.002, transform=np.ones((1, leo_dimensions)))

        self.A = self.map_phase.A
        self.B = self.map_phase.B
        self.output = nengo.Node(size_in=1, label='output')
        nengo.Connection(self.reducer, self.output)

class ThalamusGate(nengo.Network):
    '''elementwise multiplication of high-dimensional signal A with scalar
    inhibitory signal :inhibit'''
    def __init__(self, A, inhibit, inhibit_transform, mapper_neurons, control_neurons):
        self.main_input = A
        self.map_phase = BinaryElementwiseOperation(neurons=mapper_neurons,
```

```
                                                dimensions=A.size_out)
        self.control = nengo.Ensemble(neurons=control_neurons, dimensions=1,
                                      radius=1.,
                                      encoders=np.ones((control_neurons.n_neurons, 1)),
                                      max_rates=Uniform(8, 128))
        nengo.Connection(inhibit, self.control, transform=inhibit_transform, synapse=0.006)
        nengo.Connection(self.main_input, self.map_phase.A, synapse=0.002)
        nengo.Connection(self.control, self.map_phase.B, synapse=0.002,
                transform=np.ones((A.size_in, 1)))

        self.output = self.map_phase.output
```

Appendix C

symbols.py

Used to generate comparison symbols.

```
from pointer import SemanticPointer
from nengo.utils.distributions import UniformHypersphere
import numpy as np

def rand_sphere(d):
    '''Draw one random vector from a unit hypershpere of dimension n'''
    sphere = UniformHypersphere(d)
    return np.ravel(sphere.sample(1))

def initial_zero(N):
    '''Randomly generate a symbol for ZERO'''
    zero = SemanticPointer(N=N, data=rand_sphere(N))
    return zero

def gen_numerals(zero):
    '''Successively convolve ZERO with itself'''
    i = 0
    state = zero
    while True:
```

```
        yield (i, state)
        state = state.convolve(zero)
        state.normalize()
        i += 1

def similarity(x, y):
    return x.compare(y)

def numerals(D, cap, zero=None):
    if zero is None: zero = initial_zero(D)
    _gen_numerals = gen_numerals(zero)
    return [next(_gen_numerals) for i in xrange(cap)]

def batch_similarity(symbol, nums):
    return [(i, similarity(x, symbol)) for i, x in nums]
```