Logical Computation on a Fractal Neural Substrate

Simon D. Levy and Jordan B. Pollack

Brandeis University
Computer Science Department
Waltham, MA 02454, USA
levy, pollack@cs.brandeis.edu

Abstract

Attempts to use neural networks to model recursive symbolic processes like logic have met with some success, but have faced serious hurdles caused by the limitations of standard connectionist coding schemes. As a contribution to this effort, this paper presents recent work in Infinite RAAM (IRAAM), a new connectionist unification model based on a fusion of recurrent neural networks with fractal geometry. Using a logical programming language as our modeling domain, we show how this approach solves many of the problems faced by earlier connectionist models, supporting arbitrarily large sets of logical expressions.

1 Introduction

Logical computation, as represented by programming languages such as Prolog [3], is a classic example of a recursive symbolic system. Atoms, or primitives (`fred`, `wilma`, `loves`) are combined to form propositions (`loves(fred, wilma)`), which can in turn combine with other propositions (`knows(barney, loves(fred,wilma))`), *ad infinitum*. Attempts to build connectionist models of such systems have generally followed one of three approaches.

The first of these, exemplified by [15], dispenses entirely with traditional representations (data structures) and rules (algorithms on those structures), in favor of letting the network "learn" the patterns in the data being modeled, via the well-known back-propagation algorithm [14] or a similar training method. This approach became the subject harsh criticism, based on the disparity between the strength of the claims made and the actual results reported [9], as well as the apparent inability of such systems to handle the systematic, compositional aspects of meaning in recursive symbol systems [5].

The second sort of connectionist approach goes beyond the rules-and-representations view and directly to the heart of what computing actually means, by showing how a recurrent neural network can perform all the operations of a Turing machine, or more [16]. Though such proofs may hold a good deal of theoretical interest, they do not address the degree to which a particular computational paradigm (connectionism) is suited to a particular real-world task (logic). They are therefore not of much use in arguing for or against the merits of connectionism as a model of any particular domain of interest, any more than knowing about Turing equivalence will help you in choosing between a Macintosh and a Pentium-based PC.

The third approach, which some of its proponents have described as "Representations without Rules" [7], is the one that we wish to take here. This approach acknowledges the need for systematic, compositional structure, but rejects traditional, exceptionless recursive rules in favor of the flexible computation afforded by connectionist representations. Proponents of such a view are of course responsible for showing how these representations can support the kinds of processes traditionally viewed as rules. In the remainder of this paper we show how the behavior of neural network called an Infinite RAAM corresponds directly to one such process, unification, thereby supporting a systematic, compositional model of logical computation, as well as other recursive symbol systems.

2 Unification

Unification, a pattern-matching algorithm popularized by Robinson [12] as a basis for automated theorem-proving, is at the core of logical programming languages like Prolog. The basic unification algorithm can be found in many introductory AI textbooks (e.g., [11] p. 152), and can be summarized recursively as follows: (1)

A variable can be unified with a literal. (2) Two literals can be unified if their initial predicate symbols are the same and their arguments can be unified.

If, for example, we have a Prolog database containing the assertion `male(albert)`, meaning "Albert is male", and we perform the query `male(Who)`, asking "Who is male?" the unification algorithm will first attempt to unify `male(albert)` with `male(Who)`, and will succeed in matching on the predicate symbol `male`, by rule (2). The algorithm will then recur, attempting to unify the variable `Who` with the atomic literal `albert`, and will succeed by rule (1) and terminate, with the result that `Who` will be bound to `albert`, answering the query.

Of course, real programming-language applications require unification algorithms more complicated than the one illustrated in this simple example, but the example suffices for our goals here.

3 RAAM

Before describing how the Infinite RAAM model is suited to performing unification, some historical background on this model is necessary.

Recursive Auto-Associative Memory or RAAM [10] is a method for storing tree structures in fixed-width vectors by repeated compression. Its architecture consists of two separate networks: an encoder network, which can construct a fixed-dimensional code by compressively combining the nodes of a symbolic tree from the bottom up, and a decoder network which decompresses this code into its two or more components. The decoder is applied recursively until it terminates in symbols, reconstructing the tree. These two networks are simultaneously trained as an autoassociator [1] with time-varying inputs. If the training is successful, the result of bottom up encoding will coincide with top-down decoding. Figure 1 shows an example of a RAAM for storing binary trees using two bits of representation for each input and output:[1] `a` = 01, `b` = 10. Solid lines depict encoder weights, dashed lines decoder weights. Note the real-valued representation of the tree `(a b)` on the hidden layer, which would be fed back into the encoder to build a representation of the trees `(a(a b))`, `(b(a b))`, `((a b)a)`, etc.

[1] Restricting the network to only two bits per symbol allows straightforward visualization of its hidden-layer dynamics as an X/Y plot. RAAMs for real-world tasks would use many more bits per symbol.

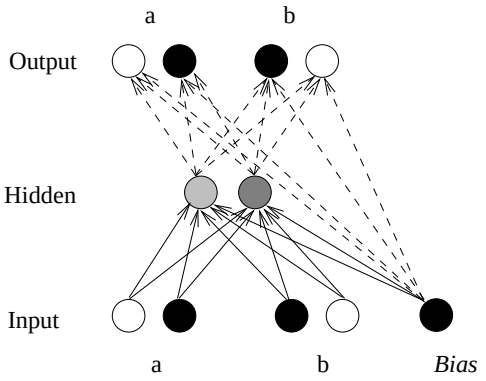

Figure 1: RAAM encoding and decoding the tree `(a b)`

4 RAAM as an Iterated Function System

Consider the RAAM decoder shown in Figure 2. It consists of four neurons that each receive the same (X, Y) input. The output portion of the network is divided into a right and a left pair of neurons. In the operation of the decoder the output from each pair of neurons is recursively reapplied to the network. The bar at the top of the figure is a "gate" that determines whether it is the the left or the right output that will be reapplied.

Using the RAAM interpretation, each such recursion implies a branching of a node of the binary tree represented by the decoder and initial starting point. However, this same network recurrence can also be evaluated in the context of dynamical systems. This network is a form of *iterated function system* (IFS) consisting of two *transforms*, which are iteratively applied to points in a two-dimensional space.

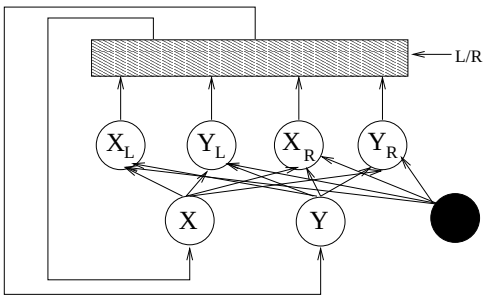

Figure 2: Detail of RAAM decoder

In a typical IFS [2], the transforms are linear equations of the form $T_i(x) = A_i x + b_i$, where x and b are vectors

and A is a matrix. The *Iterated* part of the term IFS comes from the fact that, starting with some initial x, each of the transforms is applied iteratively to its own output, or the output of one of the other transforms. The choice of which transform to apply is made either deterministically or by non-deterministic probabilities associated with each transform. If the transforms T_i are *contractive*, meaning that they always decrease the distance between any two input vectors x and y, then the limit of this process as the number of iterations N approaches infinity yields an *attractor* (stable fixed-point set) for the IFS. Most IFS research has focussed on systems whose attractor is a *fractal*, meaning that it exhibits self-similarity at all scales.

The transforms of the RAAM decoder have the form $T_i(x) = f(A_i x + b_i)$, where f is the familiar logistic-sigmoid "squashing" function $f(x) = 1/(1+e^{-x})$. Typical of connectionist models, the matrix A ranges over the entire set of real numbers, so it is not necessarily contractive. Nevertheless, the squashing function provides a "pseudo-contractive" property that yields an attractor for the decoder. In the context of RAAMs, however, the main interesting property of (pseudo-) contractive IFSes lies in the trajectories of points in the space. For such IFSes the space is divided into two sets of points. The first set consists of points located on the underlying fractal attractor of the IFS. The second set is the complement of the first, points that are not on the attractor. The trajectories of points in this second set are characterized by a gravitation towards the attractor, as follows: Each iteration produces a set of left and right copies of the points from the previous iteration. Finite, multiple iterations of the transforms have the effect of bringing the set of copies arbitrarily close to the attractor.

Dividing the space in this way allowed us to solve a vexing problem in the behavior of the decoder: Unlike the *en*coder, whose feedback terminates once it has exhausted the set of trees that make up its input, the *de*coder has no way of "knowing" when it is has decoded a terminal representation – that is, when its output is arbitrarily close enough to zero or one. Using a standard threshold ($< 0.2 = 0; > 0.8 = 1$) solved this problem to some extent, but led to several other problems, such as infinite loops and premature termination, that limited the scalability of the RAAM model. Using membership in the attractor as the "terminal test" of the decoder completely solves these problems, and allows the model to represent extremely large sets of trees in small fixed-dimensional neural codes. The attractor, being a fractal, can be generated at arbitrary pixel resolution. In this interpretation, each possible tree, instead of being described by a single point, is now an *equivalence class* of initial points sharing the same tree-shaped trajectories to the fractal attractor.

Using the attractor as a terminal test also allows a natural formulation of assigning labels to terminals. Barnsley [2] noted that each point on the attractor is associated with an address which is simply the sequence of indices of the transforms used to arrive on that point from other points on the attractor. The address is essentially an infinite sequence of digits. Therefore to achieve a labeling for a specific alphabet we need only consider a sufficient number of significant digits from this address.

These ideas are encapsulated in Figure 3, which shows a "Galaxy" attractor obtained by iterative Blind Watchmaker selection [4] to a visually appealing shape, along with sample derivations of the trees `(a b)` and `(a (a b))`. In this figure, attractor points with addresss `a`, reachable from the attractor on the left transform, are colored dark gray; points with address `b`, reachable on the right transform, are light gray. The left transients to the attractor are shown as dashed lines, and the right transients as solid lines.

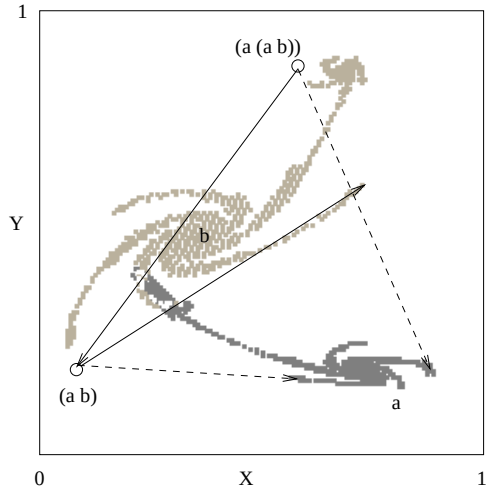

Figure 3: The Galaxy attractor and two of its trees

5 Infinite RAAM

With the "on-the-attractor" terminal test, we were able to use hill-climbing to train a RAAM decoder to generate all and only the strings in the set $a^n b^n \cup a^n b^{n+1}, n \leq 5$. As the simplest example of a non-regular, context-free formal language, $a^n b^n$ has been used as a target set by a number of recurrent-network research projects ([13, 18]), so it serves as a benchmark for the formal

power of a model such as RAAM. Analysis of the decoder weights of this $a^n b^n$ RAAM revealed a pattern that we were able to generalize into a formal constructive proof for deriving a set of weights to generate this language for arbitrarily large values of n, as a function of the pixel resolution ϵ [8].

With this proof in hand, we felt justified in using the term *Infinite RAAM* (IRAAM) to refer to our decoder networks. Against a traditional approach to problems like logic, in which recursive symbol systems are held to be the only sufficiently generative models and neural networks are seen as mere finite-capacity implementations [5], the formally proven existence of a set of "pure" $a^n b^n$ weights provided evidence that a neural network can serve as an infinitely generative model, under a dynamical-systems interpretation of the network's behavior.

6 Unification-based IRAAM

Nevertheless, a fundamental problem exists in the general case when investigating the capacity of a given IRAAM decoder via discrete sampling of the space of tree equivalence classes. Transients to the attractor can potentially meander around the entire unit space before coming to rest on the attractor, so the potential depth of the trees encoded using even a low-resolution sampling is quite large. Because the number of possible trees grows factorially with the depth of the trees, the discrete sampling method is therefore doomed to find only an infinitesimal portion of the trees that a given IRAAM could be encoding. Solving this problem requires knowing precisely how many trees to search for, and where to find them.

To limit the number of trees, it is sufficient to limit the number of IFS iterations. Like sampling, limiting the iterations produces only an approximation to the actual, infinite attractor. For zero iterations, the entire space is the attractor approximate, and the only tree encoded is a terminal, which we may refer to generically as `X`. For one iteration, each point not on the attractor goes to the attractor on one iteration, and the only tree encoded is `(X X)`. For two iterations, the trees encoded are `(X (X X))`, `((X X) X)`, and `((X X) (X X))`, and so on for more iterations. This solves the first part of the problem.

Solving the second part of the problem – locating the trees in space – requires switching from a "top-down" approach to a "bottom up" approach. We no longer start at a point off the attractor and decode the tree as this point's path to the attractor. Instead, we start at a point (or set of points) on the attractor, and ask what other point(s) that point can be unified with, using the *en*coder: hence the term *unification-based* IRAAM.

To perform this unification, we first compute the attractor, then take its image under the left and right *inverses* of the transforms. Unifications (trees) are located precisely within the intersections of these inverses. Under this interpretation, asking whether two representations can be unified means asking whether their inverses have a non-empty intersection.

For example, to determine the locations of the binary trees of depth two or less, we must first iterate the IFS twice, producing the attractor approximate A_2, which encodes the abstract terminal tree `X`. This process is depicted in Figure 4. As the figure shows, the IFS can be thought of as a kind of broken copy machine, which produces two distorted copies (left and right transforms) of its input. The union of these two copies then becomes the input to the machine on the next iteration. The first iteration makes two copies of the unit square (the attractor approximate A_0), as shown in (i). The union of these copies (ii), the attractor approximate A_1, is fed back into the machine, which makes two copies of it (iii). The union of these two copies is the attractor approximate A_2, shown in (iv).

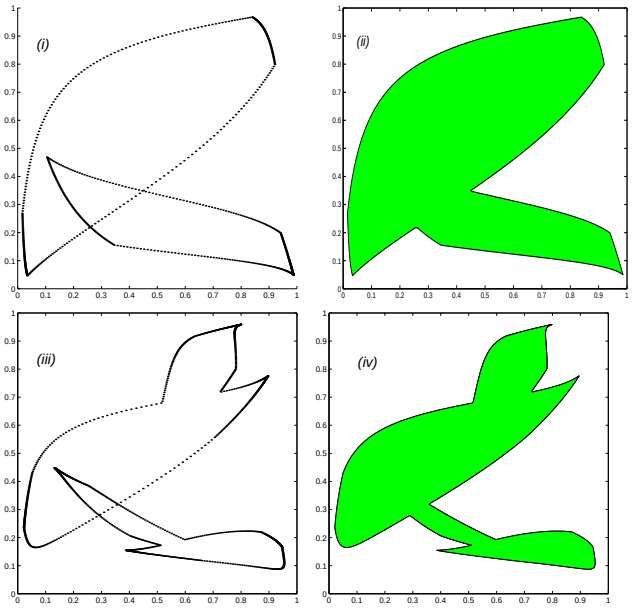

Figure 4: Two iterations of the Galaxy IFS

The attractor from (*iv*) is the region encoding the "terminal tree" `X`. Intersecting the left and right inverses of this region gives us the region encoding the tree `(X X)`. To encode the tree `((X X) X)`, we take the left inverse of this `(X X)` region and intersect it with the right inverse of the attractor. Swapping "left" for "right", the same operations can be done to obtain the tree `(X (X X))`. Finally, the tree `((X X) (X X))` is encoded by the region not encoding any of the other trees.

7 Labeling the Terminals

The discussion of the hill-climbing $a^n b^n$ decoder described a scheme for labeling the points of the attractor terminal set by means of their fractal addresses. The method involved approximating the attractor at some pixel resolution, then labeling each attractor point by the transform(s) on which that point was reachable from any points on the attractor. This scheme cannot be implemented in a model in which the attractor is approximated by iteration, because the only points reachable from the current attractor approximate A_N lie on the approximates $A_{N+k}, k > 1$. Since these points themselves are not on the part of A_N reachable from outside A_N , this scheme cannot be used to label the terminals of trees, which by definition are transients to the attractor from points outside it.

As Figure 4 illustrates, the approximated attractor is a connected set of points lying in a bounded region in space. Therefore, it is possible to treat the labeling task as a partitioning problem, in which each label corresponds to a distinct sub-region of the attractor. Under this interpretation, deriving a RAAM for a set of logical propositions requires coming up with (1) a set of decoder weights and (2) a set of partitions on the attractor induced by treating those weights as IFS transforms, such that the unifications (trees) over the partitions yield exactly those propositions, and no others.

As a first step toward solving this problem, we took a fixed set of decoder weights and used hill-climbing to obtain a set of attractor partitions yielding the following small set of propositions, borrowed with modification[2] from the introductory tutorial to a standard Prolog textbook [3]:

```
(male albert).
(male edward).
(female alice).
(female victoria).
(parent victoria).
(parent albert).
```

Each of the seven terminals (`male, female, parent, albert, edward, victoria, alice`) was represented as a distinct circular region on the attractor.[3] The initial locations (centers) of the circles were chosen so as to fall in the portion of the attractor reachable on the appropriate transform: `male`, `female`, and `parent` were on the part of the attractor reachable from the unit square on the left transform, and `albert`, `alice`, `edward`, and `victoria` were on the part of the attractor reachable on the right transform. On each hill-climbing iteration the center and radius of each circle were mutated by the addition of a small amount normally-distributed random noise. This noise was scaled by the "error" for the center point, defined as the average distance between the point's inverse and the inverses of the center points of the labels with which that point needed to unify. If the added noise moved the inverses closer together, the mutated center and radius became the new center and radius for the label.

Using a number of different initial conditions, hill-climbing was a able to discover a set of circular labeled regions for the small set of propositions after several hundred iterations. One such solution is shown in Figure 5, which displays the part of the attractor where the labels ended up. The predicates are the left side of the figure and the arguments in the upper-right part of the figure (`a=albert`, `c=alice`, `e=edward`, `v=victoria`, `m=male`, `f=female`, `p=parent`). The trees (unifications / intersections of inverses) appear as elliptical regions at the top of the figure.

Having obtained these labels, we can see how they work in conjunction with the weights to answer logical queries about the database. Consider, for example, the Prolog query `female(Who)`, corresponding to the English question "Who is female?" If we load our six-proposition database into a Prolog interpreter and pose this query, the interpreter will answer `Who = alice; Who = victoria`. To answer the question using our IRAAM model, we locate the region (circle) corresponding to the predicate `female`, take the left inverse transform of this region, and then take the right forward transform of that inverse. The regions intersected by the resulting shape contain the answer to

[2] The propositions were put into prefix form to correspond to the equivalent binary trees, so that, *e.g.*, `female(alice)` became `(female alice)`. The `parent` predicate was changed from a two-argument to a one-argument predicate, encoding only parenthood, and not who was the parent of whom.

[3] Using a circle or other convex shape makes it easy to determine intersections parametrically.

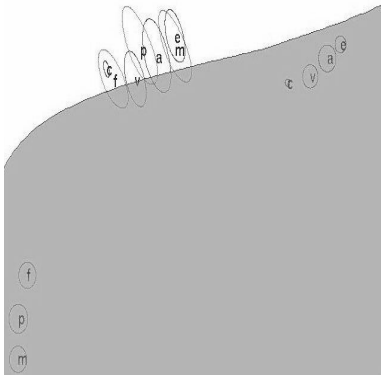

Figure 5: Attractor partitions obtained by hill-climbing

our query. For the attractor and labels shown in Figure 5, those regions are the circles labeled `alice` and `victoria`.

8 Conclusion and future work

The small hill-climbing experiment reported here provides a minimal proof-of-concept for applying unification-based RAAM to a problem in logical computation. This particular problem is not terribly interesting, as its solution is essentially a Venn diagram; it lacks the complexity and depth of real propositional tree structures. However, there is nothing inherent in the RAAM model or the partitioning algorithm that restricts us to such shallow examples, and we are currently working on extending the algorithm to trees of depth three and more. Future work will attempt to integrate the discovery of both the network weights and the partitions, using a co-evolutionary paradigm of the sort described in [6].

Fractal representation of structured information in neural networks is a relatively new field, and we have yet to test the model on substantial empirical data. We are however encouraged by the success of related work in fractal encoding of grammars [17], and see our work as contributing to this effort. We hope that such work will serve as a foundation for a principled "unification" of connectionist approaches with more traditional symbolic models, perhaps as an alternative to hybrid methods.

References

[1] D. H. Ackley, G.E. Hinton, and T.J Sejnowski. A learning algorithm for boltzmann machines. *Cognitive Science*, 9:147–169, 1985.

[2] M. F. Barnsley. *Fractals everywhere*. Academic Press, New York, 1993.

[3] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer Verlag, Berlin, 1994.

[4] R. Dawkins. *The Blind Watchmaker: Why the Evidence of Evolution Reveals a Universe Without Design*. W.W. Norton and Co., New York, 1986.

[5] J.A. Fodor and Z.W. Pylyshyn. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28:3–71, 1988.

[6] W.D. Hillis. Co-evolving parasites improves simulated evolution as an optimization procedure. In C. Langton, C. Taylor, and J.D. Farmer, editors, *Artificial Life II*, pages 313–324. Addison Wesley, 1992.

[7] T. Horgan and J. Tienson. Representations without rules. *Philosophical Topics*, XVII(1):147–175, 1989.

[8] O. Melnik, S. Levy, and J.B. Pollack. Raam for an infinite context-free language. In *IJCNN 2000*. International Joint Conference on Neural Networks, IEEE, 2000.

[9] S. Pinker and A. Prince. On language and connectionism: Analysis of a parallel distributed processing model of language acquisition. *Cognition*, 28:73–193, 1988.

[10] J.B. Pollack. Recursive distributed representations. *Artifical Intelligence*, 36:77–105, 1990.

[11] E. Rich and K. Knight. *Artificial Intelligence*. McGraw-Hill, New York, 1991.

[12] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[13] P. Rodriguez, J. Wiles, and J.L. Elman. A recurrent neural network that learns to count. *Connection Science*, 11:5–40, 1999.

[14] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representation by error propagation. In D.E. Rumelhart and J.L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 1. MIT, 1986.

[15] D.E. Rumelhart and J.L. McClelland. On learning the past tenses of english verbs. In D.E. Rumelhart and J.L. McClelland, editors, *op. cit.*, volume 2. 1986.

[16] H. Siegelmann. Computation beyond the turing limit. *Science*, 268:545–548, 1995.

[17] W. Tabor. Fractal encoding of context-free grammars in connectionist networks. *Expert Systems: The International Journal of Knowledge Engineering and Neural Networks*,, 17(1):41–56, 2000.

[18] R.J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1:270–280, 1989.