Tools for connectionist modeling: The dynamical systems methodology

PAUL W. MUNRO

University of Pittsburgh, Pittsburgh, Pennsylvania

and

JAMES A. ANDERSON

Brown University, Providence, Rhode Island

Some understanding of dynamical systems is essential to achieving competency in connectionist models. This mathematical background can be acquired either through a rigorous set of upper undergraduate and/or graduate formal courses or via disciplined self-teaching. As part of developing a course in connectionism, we feel that although certain very basic mathematical tools are most appropriately learned in their "pure" form (i.e., from mathematics textbooks and courses), more advanced exposure to dynamical systems theory can be given in the context of an introduction to connectionism. Students thus learn to write connectionist simulations by first writing programs for simulating arbitrary dynamical systems, then using them to learn some aspects of dynamical systems in general by simulating some special cases, and finally applying this technique to connectionist models of increasing complexity.

The rapid growth of connectionist study and its inherent interdisciplinary nature make this field attractive to individuals from a wide range of disciplines, including psychology, computer science, neuroscience, linguistics, mathematics, and even physics. Eagerness to investigate properties of these networks has accelerated progress in connectionism enormously, but many investigators have been discouraged in their hurried attempts at understanding and contributing to this new approach to the mind/brain. Our opinion is that the problem is pedagogical. Some foundation in the underlying mathematics needs to be established before one jumps into the business of constructing connectionist models.

Continued expansion of the field may result in a change in the mathematics components of curricula in most psychology and neuroscience programs where competency in statistics is currently the principal goal. The time-evolution of complex systems, including connectionist systems, is described and analyzed using a mathematical framework known as *dynamical systems*. The field of dynamical systems overlaps with several other branches of mathematics, including linear algebra, differential equations, and numerical methods. Table 1 proposes a regimen of coursework (with sample texts to illustrate level) in preparation for a foray into the world of connectionist models. For the aspiring student of connectionism, these courses represent a considerable investment of effort. A class of programming techniques, which we call the dy-

namical systems programming methodology (DSPM), can be learned to simulate the behavior of dynamical systems, and thus provide access to the field of connectionism to those with limited mathematical background (the first column of Table 1).

DYNAMICAL SYSTEMS

Dynamical systems are subject to mathematical analysis of various kinds, ranging from simple techniques, including numerical simulation, to very advanced formal techniques. As shown in Table 2, the approach in DSPM is kept quite simple by approaching connectionist programs as the end stage in a sequence of skills, beginning with elementary vector concepts and continuing through an introduction to differential equations. The final stage is to teach simulation of arbitrary differential equations and to demonstrate that dynamical systems provide an excellent language for describing connectionist systems.

Mathematical Descriptions of a System: States and Their Dynamics

At a given instant, a system is described as a vector. This vector is the *state* of the system. The state varies from instant to instant as a function of the state that immediately preceded it (i.e., it is like a nonlinear generalization of a first-order Markov process). The function that governs the state-to-state transition is the *dynamical system*. These functions are typically expressed as sets of differential equations in time, such as

$$\frac{dx_i}{dt} = x_1^2 - x_i.$$

Please address correspondence to Paul W. Munro, Interdisciplinary Department of Information Science, LIS Building, University of Pittsburgh, Pittsburgh, PA 15260.

Table 1
Suggested Math Courses (and Sample Texts) Preparatory
for Connectionist Study

Essential	Nonessential, but Useful
Vector algebra	Linear algebra (e.g., Noble, 1969)
Calculus (e.g., Thomas, 1975)	Differential equations (e.g., Boyce & DiPrima, 1969)
Computer language (e.g., Pascal)	Numerical methods

Table 2
The DSPM Approach

- Concept of a vector state and trajectories of vector states
- Detailed study of first-order linear equations
- Conversion of second-order ODEs to first-order systems
- Some aspects of nonlinear ODE systems
- Computer simulation of general dynamical systems
- Casting neural networks as dynamical systems

Note—DSPM = Dynamic systems programming methodology. ODE = ordinary differential equation.

Dynamical systems can be approximated by finite-difference equations that approximate instantaneous time steps with finite (but small) time differences. Careful selection of an appropriate time-step size (dt) allows some control over the trade-off between precision of the approximation and duration of a simulation. During each iteration, the state variables (i.e., the components of the state vector) are updated such that they take on slightly different values; dt controls just how "slight" this change is. The above system would be simulated by iterating through a loop containing a statement that resembles

$$x[i] = x[i] + (x[1] * x[1] - x[i]) * dt.$$

Computer Simulation of General Dynamical Systems

The heart of any dynamical system simulation is a loop that iteratively represents jumps from one time step to the next. Although these steps ideally represent infinitely brief intervals, some short but finite grain size must be chosen for the duration of each interval. The differential equations that make up the dynamical system are coded here:

$$\begin{aligned}\frac{dx}{dt} &= -x + 3y \\ \frac{dy}{dt} &= -3x - y.\end{aligned}\quad (1)$$

Listing 1 gives an outline program for simulating the arbitrary dynamical system in Equation 1 (together with the graphics used for the example in Listing 2, this shows interesting convergence properties).

The program consists of an initialization procedure, an increment procedure, and an analysis procedure for evaluating and/or displaying the results of the simulation. Surprisingly, the input/output and initialization routines usually make up the bulk of such programs, whereas the

increment procedure, which implements the dynamical system equations, occupies a relatively small fraction of the code.

COMPUTER SIMULATION OF NEURAL NETWORKS

The first step in simulating the dynamics of any system is to define those variables relevant to the system. The next step is to construct the differential equations relating them. Two kinds of functions must generally be considered: process functions and weight-modification functions. Process functions involve computing the activities of the units in the network, which are determined by the interaction of the units with each other and/or the external environment. These functions generally include "weight" parameters that reflect the influence of one unit's activity on another. By allowing these weights to be dynamic, the response properties of the network can adapt. Adaptation of this kind is modeled by weight-modification functions.

There are many types of process functions in use: for example, the input (I) to a unit may instantaneously determine that unit's firing rate (r) independent of its previous activity (Equation 2a), or input may simply cause the firing rate to increase or decrease (Equation 2b):

$$r = f(I) \quad (2a)$$

$$\frac{dr}{dt} = g(I). \quad (2b)$$

In most networks, the response r_j of unit j contributes to the inputs I_i of a set of other units via connection strengths denoted w_{ij} ; the simplest and most common example is the linear case: $I_i = \sum_j w_{ij} r_j$.

Weight-modification functions exhibit considerable variety as well. There is a general preference for functions that are dependent on *local* variables; that is, the change Δw_{ij} should be expressible as a function of variables linked to unit i and unit j (Equation 3):

$$\Delta w_{ij} = m(a_i, b_i, c_i, \dots, x_j, y_j, z_j, \dots, A, B, C). \text{ (local)} \quad (3)$$

Global variables ($A, B, C, \dots$ in Equation 3) can be included as well, but specific influences of "third-party" units on a particular weight are very unusual. The notion of local modification functions is a direct descendent of Hebb's (1949, p. 62) statement that "when an axon of cell A is near enough to excite a cell B and persistently or repeatedly takes part in firing it, some growth process or metabolic change takes place in one or both cells, such that A 's efficiency, as one of the cells firing B , is increased," which has been converted to the following literal mathematical interpretation (Equation 4) by several modelers (e.g., Anderson, Silverstein, Ritz, & Jones, 1977):

$$\Delta w_{ij} = r_i r_j. \quad \text{(Hebbian)} \quad (4)$$

More general "Hebb-like" rules are members of the class of functions that can be factored into a "postsynap-

LISTING 1
An Example of a Dynamical System Simulation

```

program dynamo (input, output) ;

var    i      : int ;           {simulation iteration counter}
       iter   : int ;           {total iterations in simulation}
       dfreq  : int ;           {display frequency}
       dt     : real ;          {size of time step}
       x, y, dx, dy : real ;

procedure initialize ;
begin
    write("Input size of time step: ") ;
    readln(dt) ;
    write("Number of iterations: ") ;
    readln(iter) ;
    write("Display frequency: ") ;
    readln(dfreq) ;
    write("Initial values of x and y: ") ;
    readln(x, y) ;
    i := 0 ;
end ;

procedure increment ;
begin
    i := i+1 ;
    dx := (-x + 2*y) * dt ;
    dy := (-2*x - y) * dt ;
    x := x + dx ;
    y := y + dy ;
end ;

procedure showstate ;
begin
    {this routine will give periodic reports every dfreq iterations.
     the reports can be numeric and/or graphic}
end ;

procedure analysis ;
begin
    {this routine gives a final report on any final state or
     statistical values that are of interest; it may be omitted if
     showstate has provided all relevant results}
end ;

begin
    initialize ;
    repeat
        increment ;           {incremental simulation}
        if ((i mod dfreq) = 0) showstate ; {periodically display state}
    until (i=iter) ;
    analysis                   {can be inside loop}
end.

```

Note—This illustrates a partial Pascal program for simulating the system in Equation 1. Addition of the Turbo Pascal graphics routines in the program in Listing 2 yields interesting results. A spiral attractor point is seen at the origin.

LISTING 2
The Delta Rule in Two Dimensions

```

program delta (input,output) ;

const  XSIZE = 639 ;
       YSIZE = 199 ;

var a,w,r,phi,mag : array[1..2] of real ;
    s : array[1..2,1..3] of real ;
    offset : array[1..2,1..2] of real ;
    i,j,k,ctr,xaxis,yaxis : integer ;
    q,dt,xmin,xmax,ymin,ymax : real ;

procedure initialize ;
var ii,jj,xx,yy,x1,x2,y1,y2 : integer ;
begin
    dt := 0.01 ;
    write('Input pattern vector 1: '); readln(s[1][1],s[1][2],s[1][3]) ;
    write('Input pattern vector 2: '); readln(s[2][1],s[2][2],s[2][3]) ;
    a[1] := (s[1][1] + s[2][1]) / 2.0 ;
    a[2] := (s[1][2] + s[2][2]) / 2.0 ;
    mag[1] := (s[1][1]*s[1][1]) + (s[1][2]*s[1][2]) ;
    mag[2] := (s[2][1]*s[2][1]) + (s[2][2]*s[2][2]) ;
    for ii := 1 to 2 do for jj := 1 to 2 do
        offset[ii][jj] := s[ii][3] * s[ii][jj] / mag[ii] ;
    write('Input xmin,xmax,ymin,ymax: ');
    readln(xmin,xmax,ymin,ymax) ;
    xaxis := trunc(YSIZE * (ymax/(ymax-ymin))) ;
    yaxis := trunc(XSIZE * (-1.0*xmin/(xmax-xmin))) ;
    HiRes ;
    TextColor(1) ;
    draw(0,xaxis,XSIZE,xaxis,1) ;
    draw(yaxis,0,yaxis,YSIZE,1) ;
    for ii := 1 to 2 do begin
        xx := trunc(XSIZE * (s[ii][1] - xmin) / (xmax - xmin)) ;
        yy := trunc(YSIZE * (ymax - s[ii][2]) / (ymax - ymin)) ;
        draw(yaxis,xaxis,xx,yy,1) ;
        draw(yaxis+1,xaxis,xx+1,yy,1) ;
        draw(yaxis,xaxis+1,xx,yy+1,1) ;
        draw(yaxis,xaxis+1,xx,yy+1,1) ;
    end ;
    for ii := 1 to 2 do begin
        x1 := trunc(XSIZE * (-10.0*s[ii][2] + offset[ii][1] - xmin) / (xmax - xmin)) ;
        y1 := trunc(YSIZE * (ymax - 10.0*s[ii][1] - offset[ii][2]) / (ymax - ymin)) ;
        x2 := trunc(XSIZE * (10.0*s[ii][2] + offset[ii][1] - xmin) / (xmax - xmin)) ;
        y2 := trunc(YSIZE * (ymax + 10.0*s[ii][1] - offset[ii][2]) / (ymax - ymin)) ;
        draw(x1,y1,x2,y2,1) ;
        draw(x1+1,y1,x2+1,y2,1) ;
        draw(x1,y1+1,x2,y2+1,1) ;
        draw(x1+1,y1+1,x2+1,y2+1,1) ;
    end ;
end ;

```

(continued)

LISTING 2 (Continued)

```

procedure plotw(xval,yval: real) ;
var ii,jj,x1,y1: integer ;
begin
  x1 := trunc(XSIZE * (xval - xmin) / (xmax - xmin)) ;
  y1 := trunc(YSIZE * (ymax - yval) / (ymax - ymin)) ;
  for ii := -1 to 1 do for jj := -1 to 1 do plot(x1+ii,y1+jj,1) ;
end ;

procedure iterate ;
begin
  r[1] := s[1][1]*w[1] + s[1][2]*w[2] ;
  r[2] := s[2][1]*w[1] + s[2][2]*w[2] ;
  q := a[1]*w[1] + a[2]*w[2] ;
  phi[1] := s[1][3] - r[1] ;
  phi[2] := s[2][3] - r[2] ;
  w[1] := w[1] + dt*(phi[1]*s[1][1] + phi[2]*s[2][1]) ;
  w[2] := w[2] + dt*(phi[1]*s[1][2] + phi[2]*s[2][2]) ;
  plotw(w[1],w[2]) ;
end ;

begin
  initialize ;
  write('Input weight vector: ') ;
  repeat
    gotoxy(23,1) ; write(' ') ;
    gotoxy(23,1) ; readln(w[1],w[2]) ;
    for ctr := 1 to 200 do iterate ;
    writeln('Weights = ', w[1]:7:3, w[2]:7:3) ;
    writeln('Responses = ', r[1]:7:3, r[2]:7:3) ;
    until ( (w[1]=0) and (w[2]=0) ) ;
  readln ;
  textmode ;
end.

```

Note—This program runs a simulation of a single unit learning to give particular output values in response to two arbitrary stimulus two-dimensional vectors. Each pattern-response pair gives a constraint on allowed weight values that is represented by a line in the plane. The constraints can be simultaneously satisfied only at the intersection of these lines. The program first prompts for the two pattern vectors, each of which consists of three numbers: the two vector components and a training value. Limits for the graphics window are requested next. The program then draws the pattern vectors and the associated constraints, which are perpendicular to the vectors. Initial weight values are then requested. After every 100 iterations, the weights and the corresponding response values are printed. Entering a <RET> at this point allows the current trajectory to continue: a new starting point can also be entered. Each learning trajectory should converge to the intersection of the two constraint lines. Normally, the approach to the constraint will occur along some characteristic direction, but if the stimulus vectors have equal magnitudes and are perpendicular, the approach trajectories will come from all directions.

tic" (unit i) component Φ_i and a "presynaptic" (unit j) component Ψ_j (Equation 5):

$$\Delta w_{ij} = \Phi(a_i, b_i, c_i, \dots) \Psi(x_j, y_j, z_j, \dots). \quad (\text{Hebb-like}) \quad (5)$$

A very common special case of Hebb-like modification is the class of functions for which $\Psi \equiv r_j$. This restriction states that the pattern of changes to the weights into unit i is a constant multiple of the activity pattern r_j ; in the language of vector algebra, the vector of weight

changes is *parallel* to the pattern vector. It is important to note that this class of "parallel" rules (Equation 6) implies that *the change to a unit's response characteristics induced by a particular input pattern is greater for that pattern than for any other pattern of the same magnitude:*

$$\Delta w_{ij} = \Phi(a_i, b_i, c_i, \dots) r_j. \quad (\text{parallel modification}) \quad (6)$$

The delta rule for supervised learning¹ can be used to provide a clear illustration of DSPM. In a supervised

learning paradigm, a network is given with a number of stimulus patterns together with the desired response pattern for each; the weight parameters are adjusted with the goal of minimizing the discrepancy between the generated and desired output patterns. The delta rule is derived from a gradient descent principle such that it drives parameters toward minima in an error function E defined as the summed square error between the actual response (r_i) of each output unit and its training response (T_i) over a set of patterns (denoted by α):

$$E \equiv \sum_{\alpha,i} (T_i^\alpha - r_i^\alpha)^2. \quad (7)$$

The delta rule for a network consisting of linear ($r_i \equiv I_i$) units is a rule of the parallel type, where the postsynaptic factor is given by the difference between the desired and actual response values. The response r_i^α of unit i to pattern α is compared to the training signal, or desired output T_i^α , to give a postsynaptic term (Equation 8):

$$\Delta w_{ij} = (T_i^\alpha - r_i^\alpha) s_j^\alpha. \quad (\text{delta rule}) \quad (8)$$

Given a particular probability distribution over the various patterns α , a network simulation program can use this probability distribution in a variety of ways. The principle methods are (1) the stochastic method (i.e., selecting random patterns according to the probability distribution and using Equation 8), and (2) the deterministic method (Equation 9) (i.e., the weights are modified according to the expected value of Equation 8):

$$\Delta w_{ij} = \sum_{\alpha} \text{Prob}(\alpha) (T_i^\alpha - r_i^\alpha) s_j^\alpha \quad (\text{deterministic delta rule}) \quad (9)$$

Listing 2 illustrates an application of DSPM to the deterministic delta rule. The initialization procedure initializes the counter and then prompts the user for the pattern vectors and their corresponding desired response values and for simulation parameters such as the number of iterations. Graphics can also be initialized here, if desired.

A graphical simulation with two two-dimensional patterns is a good illustration of the operation of the types of stability possible under the delta rule.

CONCLUSION

As has been the case in the development of other interdisciplinary ventures, connectionism is faced with profound gaps in the knowledge bases of all individuals involved. Mathematics provides all scientists with a common language for communicating their ideas. We have taken the point of view that (1) dynamical systems is the appropriate dialect of this language, and (2) its details can be learned in the context of learning connectionist theory. It is our belief that an introduction to dynamical systems by way of computer simulation is an effective preamble to learning the principles of connectionism.

REFERENCES

- ANDERSON, J. A., SILVERSTEIN, J. W., RITZ, S. A., & JONES, R. S. (1977). Distinctive features, categorical perception, and probability learning: Some applications of a neural model. *Psychological Review*, **84**, 413-451.
- BOYCE, W. E., & DIPRIMA, R. C. (1969). *Differential equations and boundary value problems*. New York: Wiley.
- HEBB, D. O. (1949). *The organization of behavior*. New York: Wiley.
- MCCLELLAND, J. L., & RUMELHART, D. E. (1986). *Parallel distributed processing: Explorations in the microstructure of cognition*. Cambridge, MA: MIT Press.
- NOBLE, B. (1969). *Applied linear algebra*. Englewood Cliffs, NJ: Prentice-Hall.
- ROSENBLATT, F. (1962). *Principles of neurodynamics*. Washington, DC: Spartan.
- THOMAS, G. B. (1975). *Calculus and analytic geometry*. Reading, MA: Addison-Wesley.

NOTE

1. The term *delta rule* was coined by McClelland and Rumelhart (1986), but was inspired by and is closely related to the perception learning rule of Rosenblatt (1962).