1996

Using the J language for NN and GA experiments

Peter Anderson

Follow this and additional works at: http://scholarworks.rit.edu/article

Recommended Citation

This Article is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Articles by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Using the J Language for NN and GA Experiments

!!! DRAFT !!!

Peter G. Anderson*
Computer Science Department
Rochester Institute of Technology
Rochester, NY 14623-5608

Abstract

We introduce the programming language J and show its applicability for experimenting with neural networks and genetic algorithms.

We illustrate the use of J with complete programs for perceptron and back propagation learning.

1 Introduction

The J language provides an excellent programming environment for rapidly expressing complex, mathematically-based algorithms. It is an interpreted system, but it can execute large sections of code (e.g., searching, multiplying, or inverting large matrices) between the program scanning steps, so there may be minimal run-time penalty for interpretation.

J’s data types, primitive operations, and the glue to combine them give experimenters an expressive power like that of traditional mathematical notation.

J’s ability to work with aggregate data (strings, vectors, matrices, etc.) encourages J programmers to readily bring that same manner of thinking to algorithm development and program implementation. This effect on programmers is in stark contrast to languages like Fortran and C that require, for instance, array operations to be specified at the subscript level; these languages train programmers to focus on irrelevant details, even in their application area analyses.

In the present paper, we focus on J’s expressive abilities as applied to neural network training. (An extended version of this paper that also addresses genetic algorithms is available from the author.) Our approach is to present several annotated examples to illustrate J’s power and to leave the detailed presentation and training of J to other sources.

We have successfully exploited J in our recent research [1, 2].

2 What is J?

J, the most recent dialect of APL, was developed by APL’s creator, Kenneth Iverson [6]. Like its predecessor, J is built to deal with aggregates of data: arrays, vectors, lists, strings. We use it to manipulate arrays of neural cells, matrices of synaptic weights connecting them, chromosomes, and populations.

J has the appropriate primitives to create and manipulate these structures. Fig. 1 shows a simple interactive session. J’s prompt is three blanks—i.e., indentation; “NB.” introduces comments; and “J” is the identity function of one variable (“J” returns its right argument when it is called with two arguments).

2.1 Functions, binding, & composition

Iverson uses the terminology of English grammar to describe J: scalars, vectors, lists, and matrices are nouns; functions ([=., i., $, +, *, =, */+] are verbs; verb modifiers (/) are adverbs; and so on. There are also conjunctions and gerunds.

J is relatively unique among programming languages in its treatment of verbs as “first class objects” which programmers can assign to names and use to build more complex verbs. Like APL, J’s functions take one or two parameter (called monic or dyadic, respectively). We can define:

\[
\max =. >.
\]

\[
\min =. <.
\]
Figure 1: Sample J computations in an interactive session.

```j
NB. The first 10 natural numbers.
] x =: i. 10
0 1 2 3 4 5 6 7 8 9
NB. 2 other vectors of 10 numbers.
] y =: 10 $ 3 1 4
3 1 4 3 1 4 3 1 4
] z =: 3 3 4 # _6 1 5
_6 _6 _6 1 1 1 5 5 5 5
NB. Arithmetic on vectors.
>: y NB. 1 more than y.
4 2 5 4 2 5 4 2 5 4
|. y NB. Reverse of y.
3 4 1 3 4 1 3 4 1 3
x + 10 * y
30 11 42 33 14 45 36 17 48 39
NB. Right association of operators.
10 * y + x
30 20 60 60 50 90 90 80 120 120
NB. Where do vectors agree?
x = y
0 1 0 1 0 0 0 0 0 0
NB. How many components agree?
+/ x = y
2
NB. Multiply, sum, find greatest.
(*/ y), (+/ y), (>:/ y)
5184 27 1
```

Figure 2: Characters used in function names.

and use:

```
largest =: A max B
smallest =: A min B
```

where A and B are variables such as numbers or arrays of numbers. Unlike API, J uses the standard, ASCII keyboard. The built in functions' names are, generally, composed of one of the 29 characters shown in Fig. 2 used by itself or followed by a period or a colon, giving up to 87 function names. As we saw in Fig. 1: “=” is the equality test operator, and “=:” is the assignment operator; “=:” is the global assignment operator.

### 2.1.1 Function binding with &

As usual, “-” denotes subtraction, a function of two arguments. Subtraction can be bound with 1 two ways to form functions of one variable:

```
f =: - & 1
```

```
g =: 1 & -
```

With this binding, we have the meanings

```
f(x) = x - 1
```

```
g(x) = 1 - x
```

### 2.1.2 Adverbs / and ~

J uses adverbs to modify verbs and form new verbs. The (prefix) adverb “/” can be applied to dyadic verbs like “*” to form monadic verbs that operate on arrays. This was illustrated in Fig. 1: */ y is the product of the elements in the vector y.

The (postfix) adverb “~” commutes the operands of a dyadic operator: A ~ B means B ~ A. The adverb “~” also creates a monadic function from a dyadic function: ~ X is X * X.

### 2.1.3 Function composition with @

J provides a good selection of composition rules for monic and dyadic functions.
If $h$ and $k$ are functions of one variables and $w$ is a function of two variables, then the function, $F$, of one variable given by the composition

$$F(x) = w(h(x), k(x))$$

is known as a “fork,” and is expressed in J as

$$F =. h \ w \ k$$

This often reads well; consider the following example:

- minus =. -
- range =. max minus min

The components’ names describe their meanings. Use the defined function as:

```julia
range SomeArray
```

Another example is the function

$$G(x) = (x - 1)(x + 1)$$

which can be expressed as

```julia
f =. - \& 1
G =. f \* g
```

Several operators can be written in a “train,” as shown above. Explicit function composition is expressed by the `@` operator: `@`. For example, the square-of-the-range function can be defined as

```julia
square =. *-
sRange =. square @ range
```

### 2.1.4 Function selection with ‘ and @.

J’s analog for a `case` construction is a list of functions (built using the grave accent, ‘) and selected via another function’s value. If `f0, f1, f2, f3, f4, and g`, are already defined (we think of an array of functions: `f0, f1, ..`) then, then

$$F =. f0 \ ' f1 \ ' f2 \ ' f3 \ ' f4 \ @. g$$

defines a function so that

$$F y$$ evaluates $f_g(y) y$

and

$$x F y$$ evaluates $f_g(x, y) x, y$

### 2.1.5 Nonlinear functions for neural nets

Neural networks typically use nonlinear functions, so we illustrate how several of these can be tacitly defined:

- `signum` =. *
- `step` =. > & 0
- `hardlim` =. >. & 0
- `tanh` =. 7 & 0.
- `pwl` =. <<- 1 ((1 @ + &1) - (1 @ - &1))
- `pwr` =. ] % >: @ |  
- `sigmoid` =. % @ >: @ - @ -

In more conventional notation:

```julia

```julia
csignum x is signum(x)
cstep x is 0.5 - (1 + signum(x))
chardlim x is max(0, x)
canh x is tanh(x)
cpwl x is 0.5([x + 1] - [x - 1])
cpwr x is x / (1 + |x|)
csigmoid x is 1 / (1 + e^-x)
```

### 2.1.6 A program structuring device

Our neural network training programs use the function definitions:

```julia
xpos =. |:
out =. 1:2 & 2
fmt =. ":
ip =. +/ . *
sigma =. tanh =. 7 & o.
sign =. *
signum =. *
tau =. (1&-)*(1&+)
sigma =. */
sum =. 2!:55
```

in a file `defs.j`. These are imported at the beginning of the programs via

```julia
0!:010 <'defs.j'
```

which serves a function similar to C’s `#include`.

### 3 Feed forward neural nets

#### 3.1 Perceptron training

The perceptron is the simplest of all the neural network models; it occurs as a component of many other, richer
models. Our perceptron consists of a single neuron with 
\(n + 1\) inputs, \(x_0, x_1, x_2, \ldots, x_n\), with \(x_0 = 1\) serving as a bias (so that we can dispense with thresholds as a separate issue). The perceptron’s transfer function is
\[
y = \text{sign} \left( \sum_{i=0}^{n} w_i x_i \right) = \text{sign} \ w \cdot x
\]
The synaptic weight vector, \(w = (w_0, w_1, w_2, \ldots, w_n)\), is determined using Rosenblatt’s training algorithm [7], as follows.

The training input consists of \(P\) training exemplars
\[(x^k, t^k), \text{ for } k = 1, \ldots, P\]
where the \(x^k\) are real \(n + 1\) vectors, and the \(t^k = \pm 1\) are the associated targets. The initial value of the synaptic vector \(w\) is arbitrary; we choose \(w = (0, 0, \ldots, 0)\).

The network’s output is repeatedly evaluated for the input exemplars, \(x^k\), in no particular order. Whenever the output, \(y\), fails to equal the target output, \(t^k\), the synaptic weights are adjusted by the rule:
\[
w := w + t^k x^k
\]
This process continues until every input produces the desired output. Rosenblatt’s theorem assures us that this process will eventually converge (produce the desired behavior) if and only if there is some value, \(w = w^*\), that produces the target output; i.e., if and only if the set of exemplar \(n + 1\) vectors associated with positive targets is linearly separable from the set of those associated with negative targets.

In the discussion that follows, we treat \(X\) as the \((n + 1) \times P\) matrix whose \(k\)-th column is \(x^k\), and \(T\) as the row \(P\)-vector whose \(k\)-th element is \(t^k\).

Our \(J\) implementation of this training algorithm, shown in Fig. 4, trains using all the training set at once (“batch training”), using whole array operations. \(J\) encourages such an approach. The training algorithm, \(\text{p}\), is specified by the code between the header
\[
\text{p} = 3 : 0
\]
and a line with a single right parenthesis. The weights vector \(W\) is initialized to \(N_i\) zeros; the epoch counter \(\text{count}\) is initialized to zero; a \texttt{while}-loop performs the training iterations; and the final value of \(\text{count}\) is returned as a function value. The single parameter, represented by \(y\), in the code is the halting criterion—the number of allowable errors (we use 0 in the present examples). The condition in the \texttt{while} statement involves computing the perceptron’s response to the entire training set and evaluating the error vector \(E\):
\[
E = \text{sign} \ T - \text{sign} \ W \text{ ip } X
\]
This expresses:
\[
E = \text{sign}(T - \text{sign}(w \times X))
\]
The expression \(+/|E\) is the sum of the absolute values of the errors; since each error is \(+1\) or \(-1\), this is the error count, which we assign to \(\text{eCnt}\) and compare with \(y\) for the loop termination condition.

The step p2,
\[
W =: W + E \text{ ip } \text{xpos X}
\]
expresses:
\[
w = w + E \times X^T
\]
This updates the synaptic weights using as an increment the row vector \(E\) matrix multiplied with the transpose of the input exemplar matrix \(X\).

The step,
\[
\text{count} =: >: \text{count}
\]
maintains the epoch count (\(>\) is the monadic operator “add one”).

To test this system, we generated several points in the \((x_1, x_2)\) plane using \(J\)’s random number generator. We classified these points according to whether or not the following linear constraint was satisfied:
\[
0 < 1 - 2x_1 + 0.5x_2
\]
That is, a perceptron with weights given by the vector \(w\text{Goal} = 1.0 \_2.0 \ 0.5\)
would correctly classify the exemplar set.
\[
\text{Ni} = 3 \ NB. \text{ Number of input nodes.}
\text{exm} = 12 \ NB. \text{ Number of exemplars.}
\]

\(\text{NB. Inputs are exm points on the unit square,}
\text{with bias: (1,x1,x2).}
\text{mm} = 1000000\)
\[
X = 1, \ mm \% \ ? ((\text{Ni} - 1), \text{exm}) \$ mm
\text{6.3 fmt xpos X}
\]
\[
w\text{Goal} = 1.0 \_2.0 \ 0.5
\]
\text{'targets'}
\[
3 "": T =. \text{ sign } w\text{Goal} \text{ ip } X
\]
The expression

\[ ((\text{Hi} - 1), \text{exm}) \times \text{mm} \]

yields a 2 \times 12 matrix consisting of the value \( \text{mm} = 1000000 \); the “roll” operator, “?,” converts each of these values to an integer between 0 and 999999; and “\( \text{mm } \% \)” divides each value by \( \text{mm} \). The operation “1,” adds a top row of all 1’s to the matrix, making it a matrix of size 3 \times 12, each column of which is a suitable input exemplar for our perceptron. The expression

6.3 fmt xpos X

formats the transpose of the matrix, \( X \), using (C’s or Fortran’s) format 6.3, which is then printed.

The vector of targets, \( T \), is computed and printed as though it were computed by a perceptron with weights \( \text{wGoal} \).

The output of the system, shown in Fig. 3, consists of the 12 training exemplars and associated target values, 16 lines showing the progression of the \( E \) vector, the goal and the system-developed weights vectors, and the count of the number of steps the algorithms took to achieve zero errors.

Figure 4 shows the entire perceptron program written in J, including the settings of the various functions, the creation of the training data, and the perceptron procedure itself. The program shown in Fig. 4 may be the contents of a file named \( \text{perceptron} \), and, in Unix, executed by the command:

\%
\text{j < perceptron}

3.2 Multilayer perceptrons & backprop

The example problem we will work with in this section is “three-bit parity” in the form of three-bit multiplication: the input vector is a 4-tuple, \((x_0, x_1, x_2, x_3)\) with \(x_0 = 1\), as before, and \(x_i = \pm 1\) for \(i > 0\). The corresponding target is \(y = x_1x_2x_3 = \pm 1\). There are eight exemplars.

The multilayer perceptron model we will discuss here consists of three layers of neurons, denoted by vectors; \( X \) is the input layer, \( Y \) is the hidden layer, and \( Z \) is the output layer. The matrix of connection weights \( W \) connects the input layer to the hidden layer, and \( V \) connects the hidden layer to the output layer. \( \sigma_1 = \sigma_2 = \text{tanh} \) is our squashing function. The transfer

Inputs:
1.000 0.132 0.035
1.000 0.756 0.053
1.000 0.459 0.530
1.000 0.533 0.671
1.000 0.219 0.008
1.000 0.047 0.383
1.000 0.679 0.067
1.000 0.679 0.417
1.000 0.935 0.687
1.000 0.384 0.589
1.000 0.519 0.930
1.000 0.831 0.846

Targets:
1 _1 1 _1 _1 _1 _1 _1 _1 _1 _1

Successive error vectors:
1 _1 1 1 1 1 _1 _1 _1 _1 _1 _1
0 _1 0 0 0 0 _1 _1 _1 _1 _1 _1
1 0 1 1 1 0 0 0 1 1 0 _1
0 _1 0 0 0 0 _1 _1 _1 _1 _1 _1
0 1 1 1 0 0 0 0 1 _1 1 0 _1
0 _1 0 0 0 0 _1 _1 _1 _1 _1 _1
0 _1 1 1 1 0 0 0 1 1 1 0 _1
0 _1 0 0 0 0 _1 _1 _1 _1 _1 _1
0 0 _1 1 1 0 0 0 0 1 1 0 _1
0 0 0 0 0 0 _1 _1 _1 _1 _1 _1
0 0 _1 1 1 0 0 0 0 0 0 0 _1
0 0 0 0 0 0 0 0 0 0 0 0 0 _1

Goal weights and trained weights:
1.000 _2.000 0.500
6.000 _12.613 5.185

Number of epochs: 16

Figure 3: Perceptron program output.
0!0:10 < 'defs': NB. fetch usual definitions

Ni = . 3
exn = . 12
mm = . 1000000
X = [1, mm % ]atitis ((Ni-1), exn) $ mm
wGoal = . 1.0 .2.0 .5
T =. sign wGoal ip X

'Exemplars and targets:'
6.2 fmt X, T

p = . 3 : 0
W =: Ni # 0
count = . 0
while. y. < +/|E =. sign T - sign W ip X
do.
   out 3 fmt E
   W =: W + E ip xpos X
   count = . >: count
end.
count

'Epochs to converge: ', fmt p 0
'Original net: ', 10.6 fmt wGoal
'Trained net: ', 10.6 fmt W % 0 { W

Figure 4: The perceptron program

function is

\[ z_k = \sigma_2(\sum_j v_{kj} \sigma_1(\sum_i w_{ji} x_i)) \]

where the hidden layer is

\[ y_j = \sigma_1(\sum_i w_{ji} x_i) \]

Working with J encourages us to think in terms of matrices and vectors, so the above equations are more compactly described as

\[ Z = \sigma_2 \left( V \times \sigma_1 \left( W \times X \right) \right) \]

Including the error, D as the difference between the output, Z, and the target, T, we have

\[ D = T - \sigma_2 \left( V \times \sigma_1 \left( W \times X \right) \right) \]

This is nearly as clearly expressed in J:

\[ D = . T - Z \cdot \sigma V \ ip Y = . \sigma W \ ip X \]

The error for the whole training set is

\[ E = \frac{1}{2} \text{tr} (D \times D^T) \]

As with the perceptron model discussed above, we are encouraged to think in terms of arrays, and we shall henceforth regard X as the array whose columns are the input training exemplars, and Y, Z, T, E, as the arrays whose columns are the hidden node activations, output activations, targets, and errors corresponding to the training exemplars.

3.2.1 Backprop

Backprop is a gradient descent search for network parameters (weights) associated with as small as possible an error measure E as a function of W and V. That is, we decrement W and V by small multiples of \( \frac{\partial E}{\partial W} \) and \( \frac{\partial E}{\partial V} \), respectively. Using the formulae above, we have

\[
\frac{\partial E}{\partial V} = -(D \cdot \sigma_2'(V \times Y)) \times Y^T
\]

\[
\frac{\partial E}{\partial W} = -\sigma'_1(W \times X) \times (V^T \times (D \cdot \sigma_2'(V \times Y))) \times X^T
\]

(Dot denotes component-by-component multiplication; \( \times \) denotes matrix multiplication.) A well known side benefit from using tanh as the squashing functions is the simplicity of its derivative calculation:

\[ \frac{d}{dx} \tanh x = (1 - \tanh x)(1 + \tanh x) = \tau(\tanh x) \]

This reduces gradient calculations to:

\[
\frac{\partial E}{\partial V} = -(D \cdot \tau(Z)) \times Y^T
\]

\[
\frac{\partial E}{\partial W} = -\tau(Y) \times (V^T \times (D \cdot \tau(Z))) \times X^T
\]

We identify \( \delta_z \) and \( \delta_y \) (Dz and Dy in the program):

\[ \delta_z = D \cdot \tau(Z) \]

\[ \delta_y = \tau(Y) \cdot (V^T \times \delta_z) \]

and we describe the partial derivatives in the well-known “delta rule” form:

\[ \frac{\partial E}{\partial V} = -\delta_z \times Y^T \]

\[ \frac{\partial E}{\partial W} = -\delta_y \times X^T \]
The overall program structure of backprop is the same as perceptron, however we subdivide the program into a “main” program, \texttt{train.j}, shown in Fig. 5, and backprop training, shown in Fig. 6, or backprop with momentum, shown in Fig. 7.

3.2.2 Program results and experiments

The backprop program runs successfully and terminates after 580 epochs. The structure of this program is sufficiently simple that an experimenter is encouraged to play with the program, to modify it in a wide variety of ways—fiddling that is definitely discouraged when a program is several hundred lines long. Playing with long programs requires long planning; such programs need to be parameterized with flexibility definitely planned in; that is, they become longer yet.

With a program as short as Fig. 6, many “software engineering” dicta apply in novel ways: constant numbers can appear throughout the code, multiple statements can appear on a line (using the “\texttt{\textbar}”), and the program can be toyed with at will. Many software engineering principles are put forward to encourage understandable programs; the approach suggested herein is to write programs that fit comfortably on less than one page.

One modification that we tried with the present backprop program is to add \textit{momentum}, so that when the changes to some weights are successively in the same direction, they can accumulate some speed. To do this, we introduce momentum, $\texttt{mu = 0.3}$, and save each epoch’s weight modifications in the variables $\texttt{DeltaW}$ and $\texttt{DeltaV}$. We modify the steps $\texttt{bp2}$ and $\texttt{bp3}$ as shown in Fig. 6. Using momentum allowed us to terminate training after 392 epochs. (This used the same random initialization as without momentun. Different but comparable results obtained with different random seed.)

Other experiments that are quite easy to perform are:

- Separate parameters $(\sigma_1, \sigma_2, \epsilon, \mu)$ for each layer.
- Adaptively determine the learning rates, $\epsilon$.
- Add a weight decay term to the error term as well as several other ideas described in [5].

3.3 Hopfield networks

See Fig. 8.

4 Genetic algorithms

4.1 A J GA program

A genetic algorithm (GA) is an indirect approach toward discovering good solutions to hard problems, problems for which there may be no reasonable direct approach. Examples of such problems are the \textit{NP complete problems} which includes a wide variety of problems in management science, such as scheduling and other instances of resource allocation.

To illustrate the meaning of \textit{good solutions}, consider a scheduling problem. It is not generally necessary to discover the optimal schedule, especially if the work to discover it consumes enormous computing resources; it may be possible to discover a nearly optimal schedule using modest computation.

A GA searches the set of all possible solutions to the problem using a strategy inspired by natural selection and “survival of the fittest.” Its principal data structure is a \textit{population} consisting of (tens or hundreds or thousands of) \textit{individuals} representing solutions to the problem. Each solution has a \textit{fitness}, a magnitude to be optimized, representing how well each solution meets the goals of the particular problem (cost minimization, conflict avoidance, payoff, etc.). Good solutions are \textit{selected} to serve as \textit{parents}, whose representations are disassembled and reassembled, in an operation called \textit{crossover}, to form child solutions, which are \textit{mutated}. Some form of \textit{population control} maintains the set of individuals, until a \textit{stopping criterion} is met.

Each of the italicized concepts in the preceding paragraph has to be determined for a particular GA implementation. A generic GA can supply answers for almost all of them, often representing individual solutions by bit strings and leaving it up to the particular user to supply a fitness function for the intended application.

In the program listed below, our individuals are represented by an array of length $L+1$: the first entry holds its fitness, a non-negative integer, and the remaining $L$ entries are 0’s or 1’s, denoted in our discussion by $(b_1, b_2, \ldots, b_L)$. Our program solves three toy problems. Problem 1 uses the fitness function $\sum_1^L b_i$; i.e., it seeks the bit string consisting of all 1’s. Problems 2 and 3 are somewhat richer: they reward a bit string for every time a bit in one half matches a bit in the other half. Problem 2 compares the left half to the right half; its fitness function is

\[
\frac{L}{2} + H D((b_1, \ldots, b_{L/2}), (b_{L/2+1}, \ldots, b_L))
\]

where $H D$ is the Hamming distance. Problem 3 com-
0!:010 < 'defs.j' NB. fetch the usual definitions

NB. This problem's parameters
Ni = .4 NB. Number of input nodes.
Nh = .2 NB. Number of hidden nodes.
No = .1 NB. Number of output nodes.
exm = .8 NB. Number of exemplars.
epsilon = 0.15 NB. Learning rate.

NB. The XOR problem -- bipolar form is multiplication.
NB. Inputs w/ bias
X = (Ni, exm)$(exm $ 1), (exm $ 1 _1), (exm $ 1 1 _1), (1 1 1 1 1 _1 _1 _1)
T = (No, exm) $ * X NB. Targets

NB. Un-comment one of the following lines to train
NB. 0!:010 < 'backprop.j' NB. without momentum.
NB. 0!:010 < 'momentum.j' NB. with momentum.

'Number of epochs to train: ', "": bp 0.3
'Target desired: ', "": T
9.5 "": Z =. sigma V ip Y =. sigma W ip X
'Output achieved: ', "": *Z

Figure 5: Multilayer perceptron, train.j, training in J.

bp =. 3 : 0
  scale =. 5
  NB. Randomly initialize the weights matrices.
  W =: scale %" mm %" (mm%2) "? (Nh, Ni) $ mm =. 1000000
  V =: scale %" mm %" (mm%2) "? (No, Nh) $ mm

  count =. 0
  whilst. y. < >./ | , D
do.
    out 9.5 "": D =. T - Z =. sigma V ip Y =. sigma W ip X NB. Feed forward

    Dy =. (tau Y) * (xpos V) ip DZ =. D * tau Z NB. Feed backward

    V =: V + epsilon * DZ ip xpos Y
    W =: W + epsilon * Dy ip xpos X
  count =. >: count
end.
  count
)

Figure 6: Backprop in J. This file, backprop.j, is included by train.j.
bp =. 3 : 0
cap =. 5
NB. Randomly initialize the weights matrices.
W =: scale %- mm %- (mm%2) -' (Nh, Ni) $ mm =. 1000000
V =: scale %- mm %- (mm%2) -' (No, Nh) $ mm

mu =. 0.3
DeltaW =. ($W) $ 0
DeltaV =. ($V) $ 0

count =. 0
whilst. y. < >./ | , D
do.
   out 9.5 "': D =. T - Z =. sigma V ip Y =. sigma W ip X
   Dy =. (tau Y) * (xpos V) ip DZ =. D * tau Z
   V =. V + DeltaV =. (mu * DeltaV) + epsilon * DZ ip xpos Y
   W =. W + DeltaW =. (mu * DeltaW) + epsilon * Dy ip xpos X
   count =. >: count
   end.
count
)

Figure 7: Backprop with momentum in J. This file, momentum.j, is included by train.j.

0!:010 <‘defs.j’ NB. fetch usual definitions

N =. 50

m =. 1000000
6.2 fmt W =. (-./ =/ i. N) * temp + xpos temp =. 0.5 -' m %- (N,N) $ m
   ,
fmt X =. 1 ^/ 2 + ? N # 2

iterate =. 3 : 0
   count =. 0
   while. count < y.
   do.
      out Energy =. - X ip W ip X
      out 3 fmt X =. signum W ip X
      count =. >: count
      end.
      out Energy =. - X ip W ip X
   )

iterate 10

Figure 8: Hopfield network in J.
pares the half with odd subscripts to the half with even subscripts; its fitness function is

\[ L/2 - H D((b_1, b_3, b_5, ...), (b_2, b_4, b_6, ...)) \]

The reason behind choosing Problems 2 and 3 will be given below.

Our population is an array with \( P \) rows, each row consisting of an \( L+1 \)-vector individual. This approach allows us to easily sort the population by decreasing fitness. Whenever children are created, they, along with their fitnesses (the sort key), are appended to the population, and the population is re-sorted and truncated to the best (most fit) \( P \) individuals. This sorting and truncating is handled by the function `sortShrink`.

We select two individuals to serve as parents by choosing one parent from the most fit half of the population and the other parent arbitrarily. This occurs in the function `evolve`, which extracts two parents, then invokes `kid`, `mut`, and `fit` to convert the parent pair to a children pair, mutate the children, and supply them with a fitness.

The function `kid` sees the two parents as a \( 2 \times (L + 1) \) array, which it breaks at an arbitrary point (after subscript \( c \), where \( 0 \leq c < L \)) and reassembles the two portions with the left portion flipped top for bottom. This is called “one-point crossover.”

The mutation function `mut` does nothing (i.e., is the identity function unless a random number in the range 0-.99 is less than the `mutPerecent`, in which case it sets one of the \( 2L + 2 \) bits of the children to an arbitrary value.

We know what the optimum fitness is for our three problems, which we denote by `maxFit`, and run the procedure until we detect an individual that achieves that goal. At the conclusion of the program, we print the number of fitness evaluations that were performed (fitness evaluation is very often the GA’s real computing bottleneck).

Each of our problems was run with the parameters shown in the program below. With the three lines of problem 1 following the five lines of problems 2 and 3 (as shown), problem 1 is the chosen one. With the five lines of problems 2 and 3 later, we run problem 2 or 3 depending upon which assignment to `mask` was done last. There is no computing penalty for writing code this way—these statements are all assignments that are done only once each. Problem 1 took 7,970 fitness evaluations; problem 2 took 18,170; and problem 3 took 4,890. The reason that problems 2 and 3 took such different times—although this was but a single test—is explained by the GA theory of schema. Schemata are the building blocks the individuals, the “genes” that contribute to an individual’s fitness. The interesting schemata for problem 1 are just single bits; those for problems 2 and 3 are bit pairs, but problem 2’s useful bit pairs are separated by a gap of length \( L/2-1 \), and two-point crossover is as likely to destroy a good pair as to leave it in place. The interesting schemata for problem 3 are adjacent bit pairs, which two-point crossover is very likely to leave undamaged. For problem 3, parents that have high fitness have it by having many “00” or “11” schemata, which have a good chance of being passed on to their children. If two good parents achieve their high fitnesses in different ways, then it may happen that one of their children will inherit a large number of good schemata from both parents—this is how the GA finds its good solutions. Unfortunately, many problems are like problem 2 in which the symbiotic relationship between schemata and crossover mechanism lets us down.

But the lesson of the present discussion is the ease by which we were able to perform this experiment.

References


0!:010 < 'defs.j' NB. fetch usual definitions

P =. 40 [ L =. +: L2 =. 10 [ mutPercent =. 15

pop =. _1, "1 ? ( P, L ) $ 2 NB. Initial population

fit =. ((out @ sum) , ] ) @ (1& ) . NB. sum bits(Prob 1)
maxFit =. L NB. for problems 1

'The initial fitnesses are:'
0 { xos pop =. \: fit "1 pop

kid =. 3 : 0
  xos _1, ( c { . xx ), |. ( (c =. ?L ) . xx =. 1 ) . xos y . )
)

mut =. 3 : 0
  if. mutPercent >: ?100
    do.
      NB. Set a random position to a random value..
      ($ y . ) $ (?2) ((?2*L+1 ) ) , y .
    else.
      y .
    end.
  )

evolve =. 3 : 0
  fit "1 mut kid ((? <. P%2 ){y . ), , (: (P?y . )
)

sortShrink =. (P & {. ) @ (\: )

prog =. 3 : 0
  :
  count =: 0
  out blank ='. ',
  while. x . > 0 { 0 { y .
    do.
      y . =. sortShrink y . , evolve y .
      count =: >: count
    end.
    out blank
    y .
  )

  :

'The fitnesses of each newly created child are:'

0 { xos maxFit prog pop

(fmt 2 * count), ' fitness evaluations.'

Figure 9: GA in J