Simulating Recurrent Neural Networks in Forth

Sergey Baranov
St. Petersburg Institute for Informatics and Automation of the Russian Academy of Sciences (SPIIRAS), ITMO University[1]
SNBaranov@gmail.com

Introduction. Recurrent neural networks (RNN) [1] regained attention of researchers as an instrument for data recognition with a great variety of strategies for transmitting information (usually binary images, video, and audio frames) among their network layers and ways of information transformation and processing. After a certain boom in instrument creation a number of software tools appeared [2, 3] which helped researchers to study various aspects of RNN-based solutions, Matlab [4] probably being among mostly widespread ones. However, most of these tools look like "dinosaurs" – they are huge and inflexible for running sophisticated experiments with carefully carved parameters and features.

An alternative approach based on the "small is beautiful" paradigm [5] was successfully used to overcome some of these hurdles, tools based on the Python language [6, 7] being quite successful and thus encouraging to try other options.

This paper describes a program, called the Associative Intellectual Machine (AIM) [8], for simulating the behavior of a multi-layer RNN under a particular scenario of input signals incoming and the given structure of their further propagation among RNN layers. An input signal considered as a binary image is converted into a matrix of pulses that propagate through the RNN and may produce a series of output signals in the same form. AIM is a prototyping tool for studying the associative memory mechanism modeled through such an RNN and establishing its characteristics (e.g., the precision of recognition of presented binary images) in comparison with conventional memory under various space-time structures for pulse propagation [9]. The ultimate goal of this preliminary research is to design an AIM as an analog associative memory device of the RNN architecture on highly parallel memristors [10] as its base elements.

In the classical McCalloch-Pitts discrete neural network model with N neurons ν_i (i=1..N) (often referred to as the perceptron model), the N binary inputs $x_j(t)$ of each neuron ν_i received at the time moment t are converted into its single binary output $y_i(t+1)$ at the next time moment t+1 according to the formula:

$$y_i(t+1) = max(0,signum(\Sigma_{j=1..N}(w_{ij} \times x_j(t) + w_{i0}))).$$

Here w_{ij} are the weights of the input synaptic links to the neuron ν_i and w_{i0} is the so called threshold value for this neuron.

In a more general Hebb model, the weights of synaptic links are updated at each step of the network functioning, thus performing *training* of the network:

$$w_{ij}(t+1) = w_{ij}(t) + \eta \times y_i(t) \times y_j(t).$$

Here η is the so called "training factor" usually selected from the interval [0.7..0.9]. A number of different training strategies were proposed; however, they all comply with the known Widrow-Hoff rule:

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t+1), \quad \Delta w_{ij}(t+1) = x_j(t) \times (d_i(t+1) - y_i(t+1)),$$

$d_i(t+1)$ being the expected output value at the next time moment t+1. After a series of trainings on the provided set of M inputs $X=\{X_1,X_2,...X_M\}$ (where each X_j is a vector of N binary values $X_j=\{x_{1j},x_{2j},...,x_{Nj}\}$) and expected outputs $D=\{d_1,d_2,...d_M\}$ the weights are not changed anymore and the network continues its performance with thus obtained weights of the synaptic links.

In an RNN of the classical Hopfield model the output $y_i(t+1)$ also serves as an input to all other neurons at the next time moment t+1.This *feedback* link extends the network capabilities

[1] This work was partially financially supported by Government of the Russian Federation, Grant 074-U01.

for training and self-learning of the RNN. In this model pulse propagation is relatively straight forward (even for recurrent networks) and homogeneous for all constituting neurons.

Existing Forth implementations of neural networks, like [11], [12], and [13], follow the above mentioned classical models of one or more layer networks with discrete timing cycles of 1 unit. In contrast to these, AIM uses various timing delays in pulse (or excitation) propagation and elaborates further the *routes* or *paths* of such propagation among the network layers. The neuron layers are split into equal fields and a path is specified through enumerating them as they occur in this path [14]. Neurons in two adjacent fields of a pulse propagation path are called neighboring or twin neurons if they are located in the same places in these adjacent fields. While in general case each RNN neuron is linked to all other neurons located in the adjacent layers and thus may receive/send pulses from/to them, the neighboring neurons have priority in pulse propagation over other neuron pairs: the effect of a pulse between them is much stronger than that of a pulse between non-twin neurons.

The main distinguishing feature of the AIM is how the neuron output and the weight of the respective synaptic link are recalculated when a pulse comes through this link, this may change the neuron potential which is its output and the weight of the link, both considered as integers. The new potential U_ν $(t+\Delta t)$ of a neuron ν at the next time moment $t+\Delta t$ equals to the sum of all its inputs, not exceeding some U_{max}:

$$U_\nu(t+\Delta t) = min(U_{max}\,,\, max(0,U_\nu(t) + \Sigma_{\eta\in\{\nu\leftrightarrow\eta\}}(U_\nu(t) - U_\eta(t)) \times w_{\nu\leftrightarrow\eta}))),$$

where $\{\nu\leftrightarrow\eta\}$ is a set of all synaptic links connecting neurons ν and η, and $w_{\nu\leftrightarrow\eta}$ is the weight of this link $\nu\leftrightarrow\eta$ which is recalculated accordingly:

$$w_{\nu\leftrightarrow\eta}(t+\Delta t) = w_{\nu\leftrightarrow\eta}(t) + f(\nu,\eta)$$

where f is a function of these two neurons which tends to zero very fast as the distance between these neurons increases. It's noteworthy that this function may be programmed with fixed point arithmetic only (using tables for expressions like $y=e^{-x}$) thus avoiding floating point arithmetic with related issues and trade-offs. This approach assumes that weights and potentials are treated as integers multiplied by some scaling factor (e.g., 10000 for precision of up to 4 digits).

This AIM program was developed in compliance with the Forth 200x standard [15] on the VFX Forth for Windows IA32 [16] platform to be portable and run on any 32-bit Forth system compliant with Forth 200x, including freeware platforms like gForth [17]. Its core is an event-driven engine. AIM allows the user to specify various cases and combinations of “experiment parameters”, to specify pulse propagation paths, timing delays, etc., and to visualize the obtained results of RNN behavior simulation as well as the very process of their development in order to objectively estimate and compare them. Fig. 1 presents a typical AIM consisting of a two-layer RNN with one propagation path (its projection on the upper layer is marked with a dotted line).

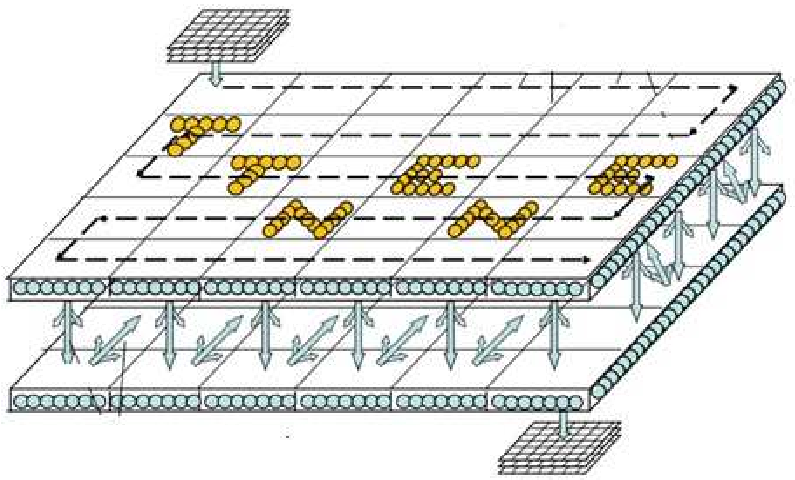

Fig. 1. A two-layer RNN of 5×6=30 fields with 6×7=42 neurons each

The simulator size is around 2 KLOC in Forth and employs a simple model of a multi-layer RNN with a user-defined interface. It relies on other advanced tools for further analysis and visualization of simulation results. The AIM source code is planned to be uploaded to an appropriate open-source repository and is currently available from the author on request.

To minimize the number of various code patterns in the code this Forth implementation intensively uses iterators, which perform the same parameterized action on each of homogeneous elements of a data structure composed by them in form of an array or a vector. They remind the standard word `TRAVERSE-WORDLIST` in their effect but rely on user-defined data structures, other than the implementation dependent list of words in a Forth vocabulary.

Representing RNN in Forth. The AIM program represents its subject RNN with special data structures and reuses an approach developed earlier [18] with a discrete counter of system time and a list of all simulated events `EventList` attached to the time axis with respective time-stamps.

Each neuron ν of the network may be in one of two states: *excited* or *unexcited*, and is characterized by the value of its current potential U_ν: $0 \le U_\nu \le U_{\max}$ which changes with time. An unexcited neuron becomes excited when its potential reaches some value U_{min}: $0 < U_{min} \le U_{\max}$.

$F_0[0,0]$	$F_0[0,1]$	$F_0[0,2]$	$F_0[0,3]$	$F_0[0,4]$	$F_0[0,5]$
$F_0[1,0]$	$F_0[1,1]$	$F_0[1,2]$	$F_0[1,3]$	$F_0[1,4]$	$F_0[1,5]$
$F_0[2,0]$	$F_0[2,1]$	$F_0[2,2]$	$F_0[2,3]$	$F_0[2,4]$	$F_0[2,5]$
$F_0[3,0]$	$F_0[3,1]$	$F_0[3,2]$	$F_0[3,3]$	$F_0[3,4]$	$F_0[3,5]$
$F_0[4,0]$	$F_0[4,1]$	$F_0[4,2]$	$F_0[4,3]$	$F_0[4,4]$	$F_0[4,5]$

Upper neuron layer #0

$F_1[0,0]$	$F_1[0,1]$	$F_1[0,2]$	$F_1[0,3]$	$F_1[0,4]$	$F_1[0,5]$
$F_1[1,0]$	$F_1[1,1]$	$F_1[1,2]$	$F_1[1,3]$	$F_1[1,4]$	$F_1[1,5]$
$F_1[2,0]$	$F_1[2,1]$	$F_1[2,2]$	$F_1[2,3]$	$F_1[2,4]$	$F_1[2,5]$
$F_1[3,0]$	$F_1[3,1]$	$F_1[3,2]$	$F_1[3,3]$	$F_1[3,4]$	$F_1[3,5]$
$F_1[4,0]$	$F_1[4,1]$	$F_1[4,2]$	$F_1[4,3]$	$F_1[4,4]$	$F_1[4,5]$

Lower neuron layer #1

Fig. 2. Representation of a two-layer RNN of 30 fields as two matrices of 5 × 6 elements

Neurons form *K layers* of the same size, represented as a matrix of *M* × *N fields*, each field being itself a matrix of *m* × *n* neurons, where $1 \le K,M,N,m,n \le 32$. Thus, the total number of RNN neurons equals to the product $K \times M \times N \times m \times n$. This representation of a sample RNN in Fig. 1 is rendered in Fig. 2. The above mentioned RNN parameters *M*, *N*, *m*, and *n* are represented by Forth variables `m'`, `n'`, `m`, and `n`: respectively:

```
VARIABLE m' 5 m' ! \ The number of rows in the field layer matrix
VARIABLE n' 6 n' ! \ The number of columns in the field layer matrix
VARIABLE m  6 m !  \ The number of rows in a neuron field matrix
VARIABLE n  7 n !  \ The number of columns in a neuron field matrix
```

Fields within a layer with the number *l* ($0 \le l \le K-1$) are numbered as elements of the matrix $F_l[i,j]$, $0 \le i \le M-1$, $0 \le j \le N-1$, all numbering starting from zero as is common in Forth; neurons inside each field, in their turn, are numbered as elements of the matrix $Neu[i,j]$, $0 \le i \le m-1$, $0 \le j \le n-1$. Thus, the full address of a neuron consists of 5 items (5 bits each) packed in one cell value: its layer number, 2 indices of the neuron field in this layer, and 2 indices of the neuron in this field. Fields reside in their layer matrix elements. Due to the above constraints on the index values, the described neuron address may be packed in one integer of the cell size which requires 5 × 5 = 25 bits; that's why a 32-bit Forth platform is anticipated for this AIM implementation.

Neurons are represented with data structures of `NeuronSize` cells each, fields are just vectors of neurons, and layers are vectors of fields. Upper case words are standard Forth words, while words with low shift characters are the user-defined words of this implementation.

```
5 CELLS CONSTANT NeuronSize \ The size of the data structure for a neuron
BEGIN-STRUCTURE Neuron
       CELL +FIELD Neuron.ID \ Unique ID of the neuron
       CELL +FIELD Neuron.State \ Current status of the neuron: 1 excited, 0 unexcited
       CELL +FIELD Neuron.Potential \ Current potential multiplied by scaling factor
       CELL +FIELD Neuron.SynDn \ Address of a vector of references to the upward layer
       CELL +FIELD Neuron.SynUp \ Address of a vector of references to the downward layer
END-STRUCTURE
```

Iterators simplify performing a particular action upon each neuron in a field:

```
: IterNeuField ( fieldaddr,cfa--) \ Iterator on all neurons of this field
       {: Action :}            \ Action to be executed for each neuron
       ( fieldaddr) DUP n @ m @ * NeuronSize * + ( NeuFirstAddr,NeuLastAddr) SWAP
       DO
              I ( NeuAddr) Action EXECUTE  NeuronSize
       +LOOP ;
```

E.g., in order to unexcite all neurons of a particular field (i.e., to assign zero to the neuron status stored in the second cell of the neuron structure) with the word `Unexcite` one may write:

```
: Unexcite ( NeuAddr--) Neuron.State 0! ; \ Unexcite this neuron
    ... ( ...,fieldaddr) ['] Unexcite  IterNeuField ( ...) ...
```

The address of the given field is placed on the stack prior to this code and the iterator enumerates all neurons of this field executing the `Unexcite` word for each such neuron. Iterator on each field within a layer is defined in a similar way with the only difference that the field size is represented as a variable rather than a constant because it depends on the actual number of neurons in a field.

```
VARIABLE FieldSize   m @ n @ * NeuronSize * CELL 2* + FieldSize !
: IterFieldLayer ( l#,cfa--) \ Iterator on all fields of this layer
       {: Action :}          \ Action to be executed for each field
       CELL * Layers + @ CELL+ CELL+ ( field00addr)
       DUP ( field00addr,field00addr)
       m' @ n' @ *  FieldSize @ * + SWAP
       DO
              I ( FieldJIaddr) Action EXECUTE  FieldSize @
       +LOOP ;
```

As the field structure starts with two auxiliary cells; that's why `CELL+ CELL+` is needed to obtain the address of the first field $F_l[0,0]$ in this vector. E.g., one may print out the current status of the RNN under simulation with the word `.RNN` defined as follows:

```
: .RNN ( --) \ Print-out the RNN status
       #Layers @ 0
       DO
              CR ." Layer "  I .
              I ( l#) ['] .Field IterFieldLayer
       LOOP ;
```

Here `#Layers` is a variable which stores the number of layers in the RNN and `.Field` is a word which prints out a field which address is provided to it on the stack.

Data flow (in form of pulse or excitation propagations) in an RNN occurs between its layers, as each neuron in a layer is connected to all other neurons in adjacent layers via special channels called synapses. Thus, the total number of synapses: $(K-1)\times(M\times N\times m\times n)^2$ is rather large. However, it may be reduced with the notion of *synapse length* defined as the distance between its two neurons considered as points in a 3D space. A two-way synapse $\nu\leftrightarrow\eta$ is established between neurons ν and η from adjacent layers (considered as points in a 3D space), if and only if the distance $d_{\nu\eta}$ between them (i.e., the length of the synapse $\nu\leftrightarrow\eta$) does not exceed some predefined constant D_{max}. This distance is calculated as:

$$d^2_{\nu\eta}=D^2_z+D^2_{xy}\times((|x_\nu-x_\eta|+|x_F-x_{F^*}|\times n)^2+(|y_\nu-y_\eta|+|y_F-y_{F^*}|\times m)^2),$$

where D_z, and D_{xy} are scaling factors specified while configuring the AIM program and sub-indices F and F^* refer to coordinates of the respective field as an element of a layer matrix. However, for twin neurons the distance is equal to D_z and thus is the shortest possible. In order to minimize run-time computations, all these distances are stored as their squared values.

Each synapse is rendered with a 4 cell data structure, which stores references to its both neurons in two adjacent layers (the upward and downward ones), the synapse length and weight. The last serves as the RNN memory and may change in the process of RNN functioning if the respective flag `Training` is turned `TRUE` (such updates of synapse weights realize the so called "unsupervised training" of the RNN). Positive weight means storing data and negative weight means erasing (forgetting) it. Further research is needed to better control this "forgetting" feature of the AIM [19]. Each neuron refers to 2 sets of synapses connecting it to other neurons in the upward and downward adjacent layers (the upmost RNN layer has no upward neighbor, as well as the RNN bottom layer has no downward one), implemented as vectors of references to respective synapse structures.

```
4 CELLS CONSTANT SynapseSize \ The size of the data structure for a synapse
BEGIN-STRUCTURE Synapse
       CELL +FIELD Synapse.Weight \ The current weight multiplied by scaling factor
       CELL +FIELD Synapse.Length \ Synapse length squared
       CELL +FIELD Synapse.NeuUp  \ Reference to one neuron of the two
       CELL +FIELD Synapse.NeuDn  \ Reference to the other neuron
END-STRUCTURE
```

Each vector starts with a cell containing the number of its elements. A respective iterator is used to perform a particular action on each element referred to by such vector:

```
: IterRefVect ( VectRefAddr/NULL,cfa--) \ Iterator on all elements referred to
       {: Action :}  \ Action to be executed for each vector element
       ( VectRefAddr/NULL) ?DUP \ Check for the NULL parameter
       IF ( VectRefAddr)
              DUP @ CELL * ( VectRefAddr,VectLen) OVER + SWAP
              ?DO                \ The vector may have zero elements
                     I CELL+ @ ( RefAddr) Action EXECUTE  CELL
              +LOOP
       THEN ;
```

As was mentioned before, thus defined iterators are similar to the Forth 200x word `TRAVERSE-WORDLIST` in its semantics, but rely on different user-defined data structures in the AIM.

To reduce the number of colon definitions used only once in the respective iterator, one may use mechanism similar to quotations [20] or the word `:NONAME` of the Forth 200x standard.

With iterators, basic operations on neurons look quite straight-forward and transparent demonstrating the flexibility and power of Forth. E.g., passing excitation from an excited neuron to all unexcited neurons connected to it via synapses looks as:

```
: PassNeuExcit ( neu-addr--) \ Excite the neuron if its potential is high
\ and recalculate potentials of all neurons connected to it by synapses
       DUP Neuron.State 2@ ( neu-addr,potential,state) 0= SWAP Umin @ >= AND
       IF ( NeuAddr) \ The neuron is unexcited and its potential is high
              DUP Neuron.State 1+! \ Excite this neuron!
              DUP Neuron.SynDn @ ['] PulseDn IterRefVect
              DUP Neuron.SynUp @ ['] PulseUp IterRefVect
              ( neu-addr)
       THEN ( neu-addr)
       Neuron.State @ IF #ExcitedNeurons 1+! THEN ;
```

Words `PulseDn` and `PulseUp` propagate excitation through the synapse, passed to them as an address on stack, downward or upward respectively, using a common subroutine `PulseDn/Up` :

```
: PulseDn ( SynAddr--) \ Propagate pulse downward through the synapse
      DUP Synapse.NeuUp 2@ ( SynAddr,NeuDnAddr,NeuUpAddr) PulseDn/Up ;
: PulseUp ( SynAddr--) \ Propagate pulse upward through the synapse
      DUP Synapse.NeuUp 2@ ( SynAddr,NeuDnAddr,NeuUpAddr) SWAP PulseDn/Up ;
```

An RNN pulse propagation path is specified through non-recursive enumeration of RNN fields, each two adjacent elements in this list belonging to adjacent layers (in case of a two-layer

RNN this means alternating). The first element of this list is its entry – it may accept signals from the RNN environment in form of an input unitary image (IUI) which is a matrix of $m \times n$ binary values; each bit being mapped to a neuron in the entry field with the same matrix indices. If this neuron is unexcited, then it accepts the respective binary signal which may result in a change of the neuron potential. If the neuron potential accumulated from all synapses incoming to it reaches the value U_{min}, it becomes excited for a period of time equal to some Δt_{excite} (during that period the neuron accepts no other signals) and all excited neurons of the input field pass their excitation to other neurons and the process reiterates.

A number of paths should be specified in a configuration file to determine the routes for excitation to propagate among the RNN layers. This constitutes a distinguishing feature of the AIM. As An XML-like technique is used for that which employs a pair of words `<Path>` and `</Path>` to frame a particular path, while words `<F>` and `</F>` frame coordinates of each particular field in this path the respective order. Field coordinates consist of three integers: the layer number (0 or more), the row number (0..M – 1) in and the column number (0..N – 1) this layer; e.g.:

```
0 <Path> \ Specify a path number 0
<F> 0 0 0 </F> <F> 1 0 0 </F> <F> 0 0 1 </F> <F> 1 0 1 </F> <F> 0 0 2 </F> <F> 1 0 2 </F>
<F> 0 0 3 </F> <F> 1 0 3 </F> <F> 0 0 4 </F> <F> 1 0 4 </F> <F> 0 0 5 </F> <F> 1 0 5 </F>
<F> 0 1 5 </F> <F> 1 1 5 </F> <F> 0 1 4 </F> <F> 1 1 4 </F> <F> 0 1 3 </F> <F> 1 1 3 </F>
<F> 0 0 2 </F> <F> 1 1 2 </F> <F> 0 1 1 </F> <F> 1 1 1 </F> <F> 0 1 0 </F> <F> 1 1 0 </F>
<F> 0 2 0 </F> <F> 1 2 0 </F> <F> 0 2 1 </F> <F> 1 2 1 </F> <F> 0 2 2 </F> <F> 1 2 2 </F>
<F> 0 2 3 </F> <F> 1 2 3 </F> <F> 0 2 4 </F> <F> 1 2 4 </F> <F> 0 2 5 </F> <F> 1 2 5 </F>
<F> 0 3 5 </F> <F> 1 3 5 </F> <F> 0 3 4 </F> <F> 1 3 4 </F> <F> 0 3 3 </F> <F> 1 3 3 </F>
<F> 0 3 2 </F> <F> 1 3 2 </F> <F> 0 3 1 </F> <F> 1 3 1 </F> <F> 0 3 0 </F> <F> 1 3 0 </F>
<F> 0 4 0 </F> <F> 1 4 0 </F> <F> 0 4 1 </F> <F> 1 4 1 </F> <F> 0 4 2 </F> <F> 1 4 2 </F>
<F> 0 4 3 </F> <F> 1 4 3 </F> <F> 0 4 4 </F> <F> 1 4 4 </F> <F> 0 4 5 </F> <F> 1 4 5 </F>
</Path>
```

This path alternatively enumerates all fields of the RNN in Fig. 2, starting from the field $F_0[0,0]$ (the input field) and terminating with the field $F_1[4,5]$ (the output field). When excitation reaches the output field, the ultimate potentials of its neurons are converted in OUI (output unitary images), similar to IUI, and transmitted to the RNN environment as the result of RNN functioning. As mentioned before, adjacent fields in a path are the closest: the distance between their two neurons with the same coordinates is minimal irrespectively of the actual distance between them in a 3D space. Thus, different paths impact the RNN behavior differently.

Similarly, a scenario of input signals is specified with the pair of words `<Images>` and `</Images>` which frame the scenario, while words `<I>` and `</I>` frame each separate input matrix within it. The word `<I>` is preceded by two integers: the moment of the system time when this image enters the RNN and the path number. Binary representations of the rows of the given image reside between `<I>` and `</I>` (leading zeros may be omitted). The number of such elements should be equal to m. E.g., the following scenario specifies that 10 images:

$$\begin{bmatrix}0001100\\0010110\\0100011\\1111111\\1000011\\1000011\end{bmatrix},\begin{bmatrix}1111000\\1100100\\1111100\\1100110\\1100011\\1111110\end{bmatrix},\begin{bmatrix}1111111\\1001100\\0001100\\0001100\\0001100\\0001100\end{bmatrix},\begin{bmatrix}0111110\\1100011\\1100011\\1100011\\1100011\\0111110\end{bmatrix},\begin{bmatrix}1100011\\1010111\\1001011\\1001011\\1000011\\1000011\end{bmatrix},\begin{bmatrix}0001100\\0010110\\0100011\\1111111\\1000011\\1000011\end{bmatrix},\begin{bmatrix}1111111\\1001100\\0001100\\0001100\\0001100\\0001100\end{bmatrix},\begin{bmatrix}1111110\\1100011\\1100011\\1111110\\1100000\\1100000\end{bmatrix},\begin{bmatrix}1100011\\1100011\\1100011\\1111111\\1100011\\1100011\end{bmatrix},\begin{bmatrix}0111110\\1100011\\1100000\\1100000\\1100011\\0111110\end{bmatrix}$$

enter the input field $F_0[0,0]$ of the path number 0 which starts in the upmost left corner of the layer L_0 at the time moments 1, 72, 148, 232, 321, 414, 515, 625, 745, and 868 in this order.

```
<Images>
1   0 <I> 0001100 0010110 0100011 1111111 1000011 1000011 </I>
72  0 <I> 1111000 1100100 1111100 1100110 1100011 1111110 </I>
148 0 <I> 1111111 1001100 0001100 0001100 0001100 0001100 </I>
232 0 <I> 0111110 1100011 1100011 1100011 1100011 0111110 </I>
321 0 <I> 1100011 0010110 1001011 1001011 1000011 1000011 </I>
414 0 <I> 0001100 0010110 0100011 1111111 1000011 1000011 </I>
```

```
515 0 <I> 1111111 1001100 0001100 0001100 0001100 0001100 </I>
625 0 <I> 1111110 1100011 1100011 1111110 1100000 1100000 </I>
745 0 <I> 1100011 1100011 1100011 1111111 1100011 1100011 </I>
868 0 <I> 0111110 1100011 1100000 1100000 1100011 0111110 </I>
</Images>
```

Running an experiment with the specified RNN parameters and scenario is initiated by the command `Simulate`.

Output Data. AIM produces three outputs: the log, an output file with OUIs, and an auxiliary file with additional data used for debugging and further analysis of the AIM behavior. However, other outputs may be easily added. All these outputs are plain texts and may be further processed by other tools; e.g., MS Excel or others. Fig. 3 visualizes how the number of excited neurons changes with time. The diagram was built in Excel directly from the log data.

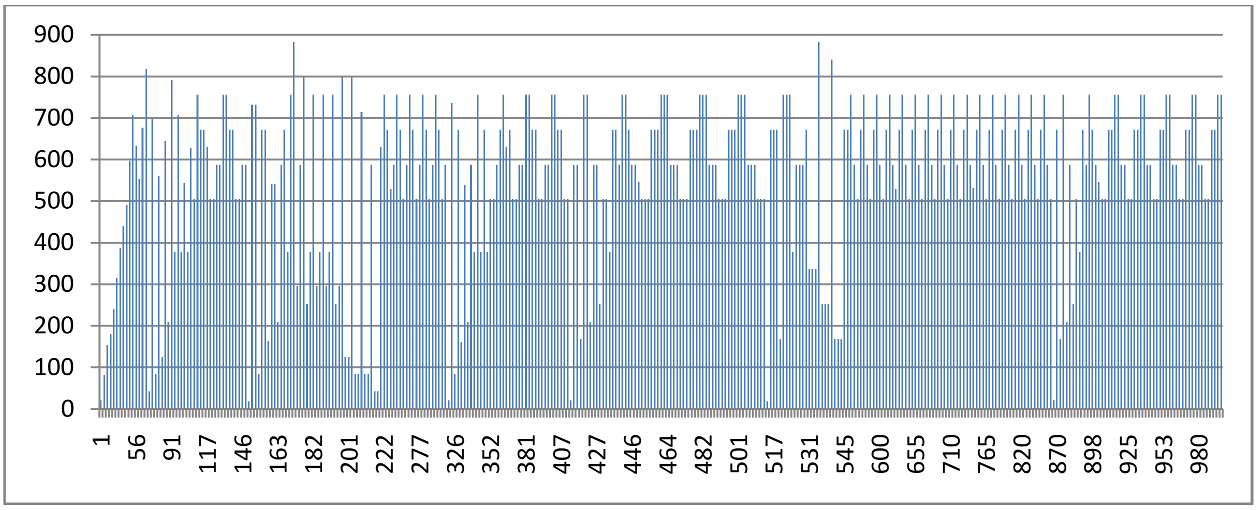

Fig. 3. The number of excited neurons vs. time in a sample RNN

Specialized tools and libraries allow for dynamic animation of the RNN functioning. A surface of a 3D manifold represents current potentials of neurons and the neuron is state rendered with the color – blue for unexcited and red for excited ones. Fig. 4 presents such an image for a frame #8 which corresponds to the *time*=41 of the above mentioned example. The left-hand chart represents the upper layer #0 and the right-hand one represents the bottom layer #1.

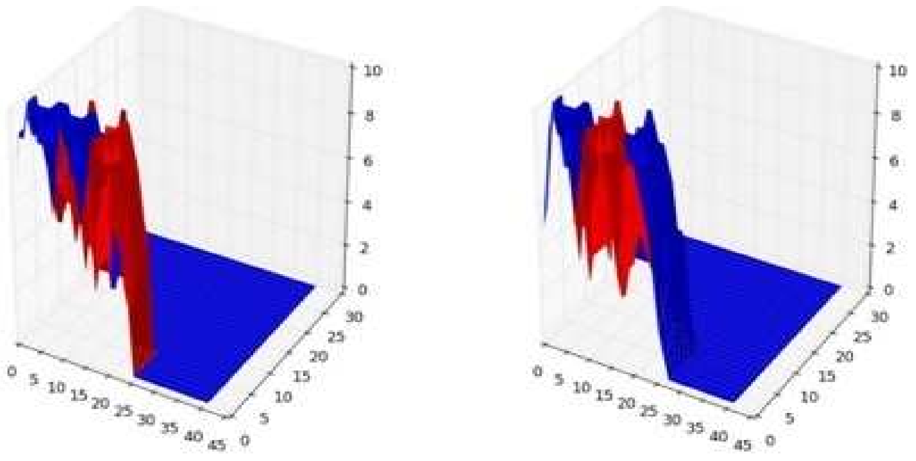
Fig. 4. A snapshot of the RNN in progress at a particular moment

The respective mapping of the output data was obtained from data generated by the AIM simulator in a plain text format with the NumPy package [21] and the Matplotlib library [22] (by the courtesy of my colleague Dr.Sergey Podkorytov). These packages allow to form a video file from a series of such images in the mp4 format as well.

The Simulator. The AIM simulator reuses the RTMT architecture [18] with a different set of events. Its overall workflow is presented in Fig. 5.

Four types of events are considered in this model: 1/2) Receive/Send an image from/to the external environment; 3) propagate excitement from excited neurons of the given field to all unexcited neurons connected with them through synapses, and excite these neurons if the accumulated potential is high enough; and 4) unexcite all excited neurons of the given field.

As already mentioned above, the simulation process is controlled by a list of events `EventList` ordered w.r.t. their time stamps: 0, t_1, t_2, t_3,... and assembled into same-time event groups. The main simulation loop consists in advancing the system time counter to the nearest time stamp of the events in this list and processing the events of this group one after another, which may produce new events with the same or later time stamp.

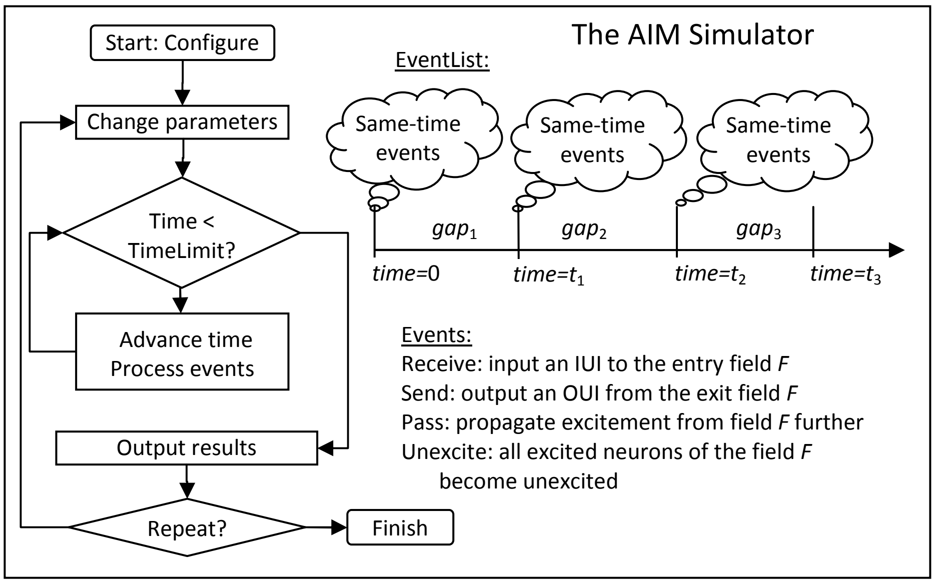

Fig. 5. The overall workflow of the AIM simulator

The main loop reiterates until the list of events becomes exhausted or the overall time limit for system time is reached, or a fatal error was encountered.

Conclusions. The described simulator is a relatively simple but powerful tool for studying various RNN structures and various combinations of pulse propagation paths and RNN parameters under various circumstances. It allows to easily specify the respective user interfaces and interoperate with other powerful tools for elaborated representing and visualization of experimental results.

The described programming solution based on the event list structure turned out to be both effective and efficient, so it's worth for reuse in other applications or subject domains. The described simple system log allows for relatively easy detecting violations and errors in the simulation process and helps in debugging of the simulator and its input data.

The simulator demonstrated acceptable performance on a regular laptop with relatively small RNNs of up to one million of synapses. Its performance can be even further improved with the assembler option offered by most Forth systems, which allows for direct programming of performance critical words in assembler, thus ensuring the most efficient realization of such critical data structures and respective processing means.

The application area of the AIM program is R&D of associative memory mechanisms for development of the respective hardware with improved characteristics and reliability. Future work will be focused on developing a variety of interfaces and typical solutions, as well as accumulating and analyzing the results of experiments with various RNN structures and data.

References.

1. Haykin, S.S., et al. Neural Networks and Learning Machines, vol. 3., Upper Saddle River: Pearson Education, (2009)
2. Brette, R., et al. Simulation of Networks of Spiking Neurons: a Review of Tools and Strategies. Journal of Computational Neuroscience, 23.3, pp. 349-398 (2007)
3. Zell, A., et al. SNNS (Stuttgart Neural Network Simulator). Neural Network Simulation Environments, Springer US, pp. 165-18 (1994)
4. Demuth, H., Beale, M. Neural Network Toolbox for Use with MATLAB, The MathWorks, Natick, MA (1998)
5. Elman J.L. Learning and Development in Neural Networks: the Importance of Starting Small. Cognition, 48, pp. 71-99, (1993)
6. Goodman D., Brette R. Brian: a simulator for spiking neural networks in Python. Frontiers in Neuroinformatic, vol. 2, article 5, web: http://www.frontiersin.org (2008)
7. Davison A. et al. PyNN: a common interface for neuronal network simulators. Frontiers in Neuroinformatic, vol. 2, art. 11, web: http://www.frontiersin.org (2009)
8. Osipov V.Yu. Associative Intellectual Machine with Three Signaling Systems. Informatsionno-upravliaiushchie sistemy (Information and Control Systems), vol. 5, pp.12-17, web: http://i-us.ru/en/article888 (in Russian) (2014)
9. Osipov V.Yu. Space-Time Structures of Recurrent Neural Networks with Controlled Synapses. Advances in Neural Networks – ISNN 2016. – Springer International Publishing, 2016, LNCS 9719, pp.177-184, web: http://link.springer.com/chapter/10.1007/978-3-319-40663-3_21 (2016)
10. Jo, Sung Hyun, et al. Nanoscale Memristor Device as Synapse in Neuromorphic Systems. Nano Letters 10.4 (2010): 1297-1301, web: http://web.eecs.umich.edu/~mazum/PAPERS-MAZUM/92_MemristorSynapse.pdf (2010)
11. Frenger P. A Forth-Based Hybrid Neuron for Neural Nets. Proc. of the Second and Third Annual Workshops on Forth. – ACM, 1991. – pp.99-102, web: http://dl.acm.org/citation.cfm?id=260009 (1991)
12. Dress W.B. Alternative Knowledge Acquisition: Developing A Pulse-Coded Neural Network. – Journal of Forth Application and Research, 1989, volume 5, number 3, pp.397-406, web: http://soton.mpeforth.com/flag/jfar/vol5/no3/article7.pdf (1989)
13. Hendrix M. Forth: Neural Net Programs, web: http://home.iae.nl/users/mhx/programs.html (1993)
14. Baranov S.N. A Practical Simulator of Associative Intellectual Machine. Advances in Neural Networks – ISNN 2016. – Springer International Publishing, 2016, LNCS 9719, pp.185-195, web: http://link.springer.com/chapter/10.1007/978-3-319-40663-3_22 (2016)
15. Forth 200x, web: http://www.forth200x.org/forth200x.html (2016)
16. VFX Forth for Windows. User manual. Manual revision 4.70, 19 August 2014. – Southampton: MPE Ltd, 2014. – 429 p., web: http://www.mpeforth.com/ (2014)
17. gForth. Free Software Foundation, Inc., web: https://www.gnu.org/software/gforth/ (2016)
18. Baranov S.N. A Forth-Simulator of Real-Time Multi-Task Applications. 31th EuroForth Conference, October 2-4, 2015, Pratts Hotel, Bath, England, pp.33-40, web: www.complang.tuwien.ac.at/anton/euroforth/ef15/papers/proceedings.pdf (2015)
19. Osipov V. Yu. Erase Outdated Information in Associative Intelligent Systems. Mekhatronika, avtomatizatsiya, upravleniye (Mechatronics, Automation, Control), vol. 3, pp.16-20, web: http://novtex.ru/mech/mech2012/annot03.html (in Russian) (2012)
20. Request for Discussion: Quotations, web: http://www.forth200x.org/quotations.txt (2016)
21. NumPy – Package for Scientific Computing with Python, web: http://www.numpy.org/ (2016)
22. Matplotib – Python 2D Plotting Library, web: http://matplotlib.org/ (2016)