**DISCERNN: The Recurrent Neural Network Library**

Version 1.0

12 January 2017

University of Utah

Salt Lake City, UT 84112

# Overview

The RNN Library stores recurrent neural networks and the information necessary to execute them, along with input files organized as a set of input neuron values. RNNs and inputs can be stored and then executed as desired.

# RNN Library Data Structure

The RNN Library (called RNN\_Library) is a struct vector where each element has the following fields:

* name (string): name of RNN or input
* W (nxn array): weight matrix for RNN
* f (1xn vector struct): each element has a field:
	+ f (string): name of neuron function (e.g., ident, threshold, sigmoid)
* u0 (nx1 vector): initial configuration vector (neuron values)
* I (1xp vector): indexes of input neurons
* O (1xq vector): indexes of output neurons
* num\_steps (int): number of steps to iterate network computation
* type (int): one of two:
	+ `program’: indicates RNN is a program
	+ `input’: indicates that RNN is simply an input vector

Operations available on the library include:

* add: add an RNN to the library
* get: get an RNN from the library
* dir: list all RNNs in the library

Next we give some usage examples. To add an RNN to the library, it must be created with all the appropriate fields. For example, the following Matlab function creates the W and f fields for an RNN to compute the absolute value on each element of a 2D array:

 abs13 = TA\_abs\_RNN(13,13)

abs13 =

 W: [338x338 double]

 f: [1x338 struct]

Then to add the RNN to the library, the remaining info must be filled in during the call:

 L = TA\_RNN\_Library\_add([],'abs13',abs13.W,abs13.f,zeros(338,1),[1:169],[170:338],1,'program')

L =

 name: 'abs13'

 W: [338x338 double]

 f: [1x338 struct]

 u0: [338x1 double]

 I: [1x169 double]

 O: [1x169 double]

 num\_steps: 1

 type: 'program'

There is also a function to execute an RNN:

* TA\_RNN\_exec

This function takes as arguments a library name, an RNN name and an input name. For example, to run the net on a 13x13 input image with rows 1:10 having value -1 and the rest with value 2, first create a 13x13 array with the desired values:

 im = zeros(13,13);

 im(1:10,:) = -1;

 im(11:13,:) = 2;

 im\_1D = im(:);

 L = TA\_RNN\_Library\_add(L,'im',[],[],im\_1D,[],[1:169],1,'input’)

L =

1x2 struct array with fields:

 name

 W

 f

 u0

 I

 O

 num\_steps

 type

To see all the RNNs in the library (L), use TA\_RNN\_Library\_dir:

 TA\_RNN\_Library\_dir(L)

abs13

im

To run abs13 on im:

 res = TA\_RNN\_exec(‘abs13’,’im’,L);

which results in res, a 169x1 vector; to display the results, we reshape res:

reshape(res,[13,13])

 0 0 0 0 0 0 0 0 0 0 0 0 0

 0 1 1 1 1 1 1 1 1 1 1 1 0

 0 1 1 1 1 1 1 1 1 1 1 1 0

 0 1 1 1 1 1 1 1 1 1 1 1 0

 0 1 1 1 1 1 1 1 1 1 1 1 0

 0 1 1 1 1 1 1 1 1 1 1 1 0

 0 1 1 1 1 1 1 1 1 1 1 1 0

 0 1 1 1 1 1 1 1 1 1 1 1 0

 0 1 1 1 1 1 1 1 1 1 1 1 0

 0 1 1 1 1 1 1 1 1 1 1 1 0

 0 2 2 2 2 2 2 2 2 2 2 2 0

 0 2 2 2 2 2 2 2 2 2 2 2 0

 0 0 0 0 0 0 0 0 0 0 0 0 0

As can be seen, the boundaries of the image are zeroed.

# L Language

The L language [Davis1994] has 4 instructions:

 V 🡨 V [**NOOP**]

 V 🡨 V+1 [**INC**]

 V 🡨 max(0,V-1) [**DEC**]

 IF V$\ne $0 GOTO L [**JMPNZ**]

The operations are represented in Matlab as:

NOOP: 1

INC: 2

DEC: 3

JMPNZ: 4

And variables are all kept in one vector, vars; if there are m input variables, then vars(1:m) represents those, the output variables are next (assume n of those) vars(m+1:m+n), and the local variables follow that, vars(m+n+1:end).

Each Instruction is a 3-tuples: [I,V,L], where I is the instruction code, V is the variable index, and L is 0 for operations NOOP, INC and DEC, and the index number of the instruction to branch to. Programs are then a kx3 array, where each row describes an instruction. For example, the L program, L\_not, to compute logical not is as follows. L\_not has one input variable, and one output variable:

JMPNZ X A

Y 🡨 Y + 1

 [A] Y 🡨 Y

The encoded version of this is:

 4,1,3

 2,2,0

 1,2,0

As another example, consider the logical and function L program and encoding:

 Z 🡨 Z+1 2,4,0

 JMPNZ X1 A 4,1,4

 JMPNZ Z E 4,4,7

 [A] JMPNZ X2 B 4,2,6

 JMPNZ Z E 4,4,7

 [B] Y 🡨 Y+1 2,3,0

 [E] Y 🡨 Y 1,2,0

Hyotyniemi [Hyottyniemi1996] provides a method to convert L programs to RNNs. The target RNN execution model runs to steady state using a unique activation function at each node:

 $y\_{q}\left(k\right)=f\left(\sum\_{p=1}^{n}w\_{qp}x\_{p}\right)$

where

 $f\left(x\right)= \left\{\begin{array}{c}x x>0\\0 else\end{array}\right.$

Conversion from an L program to an RNN is accomplished using the following rules:

1. For each V, create a node $N\_{V}$
2. For each instruction i, create a node $N\_{i}$
3. For each conditional branch instruction i, create a nodes $N\_{i^{'}}$ and $N\_{i^{''}}$
4. For each $N\_{V}$, set the weight matrix, W($N\_{V}$,$ N\_{V}$) to 1
5. If instruction i is NOOP on V:
	1. set W($N\_{i}$,$ N\_{i+1}$) to 1
6. If instruction i is INC on V:
	1. set W($N\_{i}$,$ N\_{i+1}$) to 1
	2. set W($N\_{i}$,$ N\_{V}$) to 1
7. If instruction i is DEC on V:
	1. set W($N\_{i}$,$ N\_{i+1}$) to 1
	2. set W($N\_{i}$,$ N\_{V}$) to -1
8. If instruction i is JMPNZ on V to instruction j:
	1. set W($N\_{i}$,$ N\_{i'}$) to 1
	2. set W($N\_{i}$,$ N\_{i''}$) to 1
	3. set W($N\_{V}$,$ N\_{i''}$) to -1
	4. set W($N\_{i''}$,$ N\_{i+1}$) to 1
	5. set W($N\_{i'}$,$ N\_{j}$) to 1
	6. set W($N\_{i''}$,$ N\_{j}$) to -1

# Current RNN Library

* A library of RNNs
	+ dx
	+ dy
	+ mag
	+ ori
	+ ori hist
	+ shift image
	+ translational symmetry
	+ point symmetry (corners)
	+ FEP
	+ Rotation symmetry
	+ Perceptron classifier
* Composition mechanism
* Execution capability
* Set of text images and ground truth
* Classification experiments

Execution Model

Input:

* Weight matrix, W
* Function at each node, fi
* Input: m values
* Input indexes
* Initial configuration
* Output neurons
* Number of step

Execution:

 $u\_{t+1}$= $f(wu\_{t})$ -- for fixed number of steps or until converges

Output: values of output neurons

# References

Davis, M.D., R. Sigal and E.J. Weyuker, Computability, Complexity, and Languages, Academic Press, Boston, 1994.