DISCRETE DYNAMICAL NEURAL ARCHITECTURE

DDNA

by

Joseph Norman

A Dissertation Submitted to the Faculty of

The Charles E. Schmidt College of Science

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

Florida Atlantic University

Boca Raton, FL

May 2012

DISCRETE DYNAMICAL NEURAL ARCHITECTURE

DDNA

by

Joseph Norman

This dissertation was prepared under the direction of the candidate's dissertation advisor, Dr. Christopher Beetle, Department of Physics, and has been approved by the members of his supervisory committee. It was submitted to the faculty of the Charles E. Schmidt College of Science and was accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

SUPERVISORY COMMITTEE:

Christopher Beetle, Ph.D.
Dissertation Advisor

Sukanya Chakrabarti, Ph.D.

Andy Lau, Ph.D.

Theodora Leventouri, Ph.D.

Charles E. Roberts, Ph.D.

Warner A. Miller, Ph.D.
Chair, Department of Physics

Gary W. Perry, Ph.D.
Dean, The Charles E. Schmidt College of Science

Barry T. Rosson, Ph.D.
Dean, Graduate College

Date

Warning: The table of contents of an FAU thesis *must* appear between the abstract and the list of tables. Please revise your LaTeX source accordingly.

DISCRETE DYNAMICAL NEURAL ARCHITECTURE

DDNA

Warning: The list of tables in an FAU thesis *must* appear between the table of contents and the list of figures. The command to omit the list of tables must appear in the same position. Please revise your LaTeX source accordingly.

Warning: The list of figures of an FAU thesis *must* appear between the list of tables and the beginning of the manuscript. Please revise your LaTeX source accordingly.

LIST OF FIGURES

Chapter 1

Dynamic Neural Array Architecture

The Dynamic Neural Array Architecture (DNAA) is an extensible object-oriented Matlab toolbox designed for rapid prototyping and testing of dynamical neural architectures. DNAA is designed with a focus on ease of implementation, flexibility, and robustness. Models of sufficient size or complexity can be constructed in a task-specific manner to cut down on computational overhead after prototyping with DNAA, if necessary.

DNAA is an open-source project. All of the code and documentation can be located and downloaded at `http://www.github.com/normonics/DNAA` where the project can be forked or one can request to become a contributor. This documentation covers the *alpha* release. Updated documentation will be made available with future releases.

1.1 THE NEURON CLASS FILE

The `Neuron` class file allows one to make both individual and arrays of neural elements whose properties and parameters can be specified. In general, neurons can be described by an equation of the form

$\tau \dot{u} = -u + h + input$

where u is the activation state of the neuron, h is the resting level, and *input* may represent synaptic inputs from other neurons, direct 'stimulus' inputs, or terms associated with other neural functions (e.g. adaptation).

Enabling certain properties of neurons (e.g., enabling change-detection) might lead to additional equations for each neural element. This will be detailed in the relevant sections of the documentation. By default, newly constructed neurons are of the 'simple' class described by the equation above.

1.2 PROPERTIES

1.2.1 id

Each neural element has an 'id' that can be assigned to it in order to label it. This is useful because the Neuron is a handle class, meaning that multiple handles can point to the same Neuron object. Giving a Neuron object a unique 'id' can minimize confusion stemming from this. *Note: there is nothing preventing multiple neurons from possessing the same id, so one must take care in ensuring an id is a unique identifier.*

1.2.2 t

The property 't' defines the current clock time of a given neural element. When a neuron is initialized (see init() method) 't' is set to 1. When a time step is taken in the simulation (i.e. when the step() method is called) the value of t is increased by 1 (i.e. t(n+1) = t(n) + 1). The value is always a single integer.

1.2.3 u

The activation state of each neural element is represented by the variable u. The property 'u' is a vector that represents the time-series of u for each neural element.

1.2.4 v

The antagonistic state of change-detection neural elements. To make a standard neural element into a change-detection neural element, set its `vCoefficient` property equal to 1. Each neural element with `vCoefficient = 1` is of the form (assuming no additional optional coefficients are not equal to zero):

$\tau \dot{u} = -u + h - v + input$

$\tau_v \dot{v} = -v + input$

1.2.5 a

1.2.6 aFr

1.2.7 fu

1.2.8 h

The property `h` sets the resting level of a neural element.

1.2.9 hV

1.2.10 hA

1.2.11 tau

The property 'tau' sets the time constant for the dynamical evolution of a neural element's state variable `u`.

1.2.12 tauV

The property 'tauV' sets the time constant for the dynamical evolution of a change-detection element's antagonistic state variable `v`. Typically, tauV ¿ tau. This parameter is only relevant if vCoefficient $\neq 0$.

1.2.13 tauA

1.2.14 tauAFr

1.2.15 vCoefficient

Setting vCoefficient $\neq 0$ turns on the antagonistic change-detection. When it is set to 1, the neural element can be described as

$\tau \dot{u} = -u + h - v + input$

$\tau_v \dot{v} = -v + input$

1.2.16 aCoefficient

1.2.17 aFrCoefficient

1.2.18 noiseCoefficient

The noiseCoefficient property sets the standard deviation of the normally-distributed mean-zero noise term associated with a noise element.

$\tau \dot{u} = -u + h + input + noise$

1.2.19 interaction

1.2.20 input

The input property represents all the external inputs that a neuron is receiving. The input array is defined as a $1 \times n$ cell array with n synaptic inputs. Each synaptic input is itself a cell array that contains one or more entries that are to be multiplied. Individual entries can be scalar values, time-series (i.e. 1-D vector), or the handles of neural elements.

A typical additive synapse is defined as {`synapticWeight, neuralElementHandle`}, so an input array with n additive neural inputs would be structured as

{{`synapticWeight_1, neuralElementHandle_1`},...

{`synapticWeight_2, neuralElementHandle_2`},...

...

{`synapticWeight_n, neuralElementHandle_n`}}.

Because the cell defining a given synapse can have any number of entries, a multiplicative synapse can be constructed as {`synapticWeight, neuralElementHandle_1, neuralElementHandle_2`} within the input cell array.

1.2.21 changeDetectionBypassInput

Treated in the same manner as the `input` property except that in change-detection neurons (vCoefficient $\neq 0$) only the equation determining the state `u` receives the input.

$\tau \dot{u} = -u + h + input + changeDetectionBypassInput$

$\tau_v \dot{v} = -v + input$

1.3 METHODS

1.3.1 Constructor

The constructor method is called as `Neuron([size])` where `[size]` can be empty (creates a single neural element), scalar (creates a row array of neural elements size long), or a vector of n elements that defines the size of each of the n dimensions in a multi-dimensional neural array.

Typically, newly constructed neural arrays should be assigned a handle in order to refer to the object(s). For example `x = Neuron()` assigns the handle `x` to a single neural element.

Example usage:

```
x = Neuron(4)
```

1.3.2 step

1.3.3 init

1.3.4 output

1.3.5 addInput

1.3.6 addBypassInput

1.3.7 sum

1.3.8 plot

1.3.9 map

1.3.10 Generative Methods

Generative methods are those that result in the construction of neural elements or arrays. This can be useful when one neuron space is defined in terms of one or more others. For instance, one can generate a two dimensional 'neural product space' through the combination of two one-dimensional neural arrays.

neuralProduct

1.3.11 diagProject

1.4 MODIFICATION METHODS

Modification methods set property values among extant neural arrays. This can be useful, for instance, when defining connectivity within an array or connecting arrays of neurons in a specific way.

1.4.1 coInhibit

1.4.2 wta

1.4.3 diagProjectBack

1.4.4 setDecreasingRestingLevels

1.5 TYPICAL WORKFLOW

1.5.1 Constructing neural elements and arrays

The first step in a typical workflow is to define a set of neural elements and/or arrays that will serve as the basis of the model. Some methods result in the construction of neural arrays, if these are used it is also good practice to call these near the top of a script.

1.5.2 Setting Properties

Setting properties can be done directly using dot notation or using specific `set` methods. Using `set` methods has the advantage that a single property can be modified for an entire neural array, while using dot notation must be done a single neural element at a time.

For example, if a handle points to an individual neural element, as would be the case if one called the constructor method with no arguments, the resting level of the neuron can be set with dot-notation that references only the handle (without index). The following commands demonstrates this:

```
x = Neuron()
x.h = -20
```

In this example, a single neural element is constructed and the handle `x` is assigned to it. The resting level is then changed from its default value (-10) to -20.

If a handle points to an array of neural elements, using dot notation in this way will result in an error:

```
x = Neuron(4)
x.h = -20
Incorrect number of right hand side elements in dot name assignment.
```

If one wants to set a property of a single neural element on an array, simply refer to its index when using dot notation. For example,

```
x(2).h = -20
```

sets the resting level of second neuron in the array to -20.

If one wishes to set a property for an entire array, a `set` method must be called. For instance, to create a neural array consisting of four neural elements and set all their resting levels to -30, one could write the command

```
x = Neuron(4)
setH(x, -30)
```

where the `setH` method takes two arguments: the neural array of interest and the value to be assigned. In this example, the four neural elements on the neural array have all of their resting levels set to -30. The list of `set` methods can be found in the methods section.

1.5.3 Defining connectivity

After all neural elements are constructed and their properties set, one can define the connectivity among the elements. This can be done on an individual element-to-element basis or through routines that apply some connectivity scheme to a set of elements.

Defining connectivity among individual elements can be done with the `addInput` method. Other connectivity routines included in the toolbox can be found in Section X on 'Modification methods'.

1.5.4 Running a simulation

1.5.5 Visualizing outputs