CTRNN Class Documentation

Version 1.1 – 2/15/08

Randall D. Beer
Cognitive Science Program
1910 E. 10th St. – 840 Eigenmann
Indiana University
Bloomington, IN 47406
rdbeer@indiana.edu
http://mypage.iu.edu/~rdbeer/

1. Introduction

An N-neuron continuous-time recurrent neural network (CTRNN) is defined by a set of N ordinary differential equations of the following form:

$$\tau_i \dot{y}_i = -y_i + \sum_{j=1}^{N} w_{ji}\sigma\left(g_i\left(y_j + \theta_j\right)\right) + I_i \qquad i = 1,\ldots,N$$

where y_i is the state of the ith neuron , τ_i is the neuron's time constant, w_{ji} is the weight of the connection from the jth to the ith neuron, θ_i is a bias term, g_i is a gain term, I_i is an external input, and $\sigma(x) = 1/(1+e^{-x})$ is the standard logistic output function. CTRNNs are a specific instance of the general class of additive neural network models (Grossberg, 1988). They are equivalent to the continuous model that was popularized by Hopfield (1984), except that Hopfield placed restrictions on his networks such that only equilibrium points were possible. Without these restrictions, CTRNNs are known to be universal approximators of smooth dynamics (Funahashi & Nakamura, 1993; Kimura & Nakano, 1998; Chow & Li, 2000). CTRNNs were first introduced into adaptive behavior research by Beer and Gallagher (1992) and are now widely used within evolutionary robotics. Both the general dynamical properties of CTRNNs (Beer, 1995; Mathayomchan & Beer, 2003; Beer, 2006) and the dynamics of specific CTRNNs evolved to perform particular tasks (Beer, Chiel & Gallagher, 1999; Chiel, Beer & Gallagher, 1999; Beer, 2003) have been studied extensively.

2. Class Overview

The purpose of the `CTRNN` class is to provide a simple, efficient C++ implementation of CTRNNs for use in evolutionary agent simulations. It depends on the files `VectorMatrix.h`, `random.h` and `random.cpp`.

A CTRNN can be created by simply calling the class constructor with an integer specifying the number of neurons in the circuit. If no number is specified, the circuit size defaults to 0. The size of a CTRNN can be changed at any time by calling the member function `SetCircuitSize()`. Note that neuron indices run from 1 to `CircuitSize()`.

The current state y and output $\sigma\left(g\left(y+\theta\right)\right)$ of any neuron in a CTRNN can be accessed or set by calling the `(Set)NeuronState()` or `(Set)NeuronOutput()` member functions, respectively. Note that setting the state automatically updates the output and vice versa. It is also possible to obtain references to the state or output of a neuron using the `NeuronOutputReference()` and `NeuronStateReference()` member functions. Such references can be useful for plotting routines to keep a pointer to the value they are plotting. The member functions `RandomizeCircuitState()` and `RandomizeCircuitOutput()` can be used to generate random initial states or outputs, respectively, uniformly distributed over a given range.

The time constant τ, the bias θ, the gain g and the external input I parameters of a neuron can be accessed or set by calling the `(Set)NeuronTimeConstant()`, `(Set)NeuronBias()`, `(Set)NeuronGain()`, and `(Set)NeuronExternalInput()` member functions, respectively. Likewise, the connection strength w_{ji} between two neurons can be accessed or set by calling the `(Set)ConnectionWeight()` member functions. Time constants and gains default to 1, whereas all other parameters default to 0. In addition, the member function `LesionNeuron()` can be used to completely remove a neuron from a circuit and the member function `SetCenterCrossing()` can be used to set the CTRNN biases to their center-crossing values based on the connection weights (Beer, 1995; Mathayomchan & Beer, 2003).

Updating the state of a CTRNN is accomplished by numerically integrating the defining differential equations. Currently, two integration methods are provided. A forward Euler update is initiated by calling the member function `EulerStep()` and a 4th-order Runge-Kutta update is initiated by calling the member function `RK4Step()`. As a general rule of thumb, the integration stepsize should be a factor of 10 smaller than the smallest time constant in the network. Note that it is always a good idea to try different integration methods and different integration step sizes in order to ensure that the observed dynamics of an evolved agent is robust and not just a numerical artifact of the particular integration choices.

A typical CTRNN simulation proceeds as follows. First, a CTRNN is created and initialized, either by manually setting the circuit parameters, reading them in from a file (see below), or decoding them from a search vector (see the documentation for the `TSearch` class). Then a loop is entered in which the appropriate CTRNN state update function is called repeatedly. See Section 3 for a simple example of this process. If the CTRNN is embedded within an agent model, then each update step should be preceded by reading the agent's sensors and using them to set the external inputs of the corresponding sensory neurons and followed by using the outputs of the motor neurons to set the states of the corresponding effectors.

It is sometimes convenient to be able to read and write files containing CTRNN parameters. For this reason, the C++ stream insertion and extraction operators have been overloaded to work with CTRNNs. The format of a CTRNN file is as follows:

```
<Number of Neurons>
<Time Constant 1> . . . <Time Constant N>
<Bias 1> . . . <Bias N>
<Gain 1> . . . <Gain N>
<Weight from 1 to 1> . . . <Weight from 1 to N>
          .                          .
          .                          .
          .                          .
<Weight from N to 1> . . . <Weight from N to N>
```

3. Fast Sigmoid Evaluation

Every evaluation of the sigmoidal function is expensive because it involves a call to the `exp()` math library function and a floating-point divide. For this reason, the `CTRNN` class provides a fast sigmoid option that considerably speeds up the numerical integration of CTRNNs. This option uses a table-based method with linear interpolation between table entries. The fast sigmoid option is enabled by uncommenting the `#define FAST_SIGMOID` line in the file `ctrnn.h`.

The accuracy of the approximation is controlled by the constants `SigTabRange` and `SigTabSize`. `SigTabRange` determines the range over which the approximation occurs. For example, the default value of `15.0` means that the range of `fastsigmoid(x)` is [-15.0, 15.0], with a value of `0.0` for $x < -15$ `x` and a value of `1.0` for $x > 15$. The constant `SigTabSize` determines the number of table entries used. Note that the table only stores values of $\sigma(x)$ for $0 \leq x \leq$ `SigTabRange`, since $\sigma(x) = 1 - \sigma(-x)$ for negative values of x, effectively doubling the resolution of the table.

4. A Simple Example

```
// ************************************************
// A very simple example program for the CTRNN Class
// ************************************************

#include "CTRNN.h"

// Global constants
const double RunDuration = 250;
const double StepSize = 0.01;

// The main program
int main(int argc, char* argv[])
{
    // Set up an oscillatory circuit
    CTRNN c(2);
    c.SetNeuronBias(1, -2.75);
    c.SetNeuronBias(2, -1.75);
    c.SetConnectionWeight(1, 1, 4.5);
    c.SetConnectionWeight(1, 2, -1);
    c.SetConnectionWeight(2, 1, 1);
    c.SetConnectionWeight(2, 2, 4.5);

    // Run the circuit
    c.RandomizeCircuit(-0.5,0.5);
    cout << c.NeuronOutput(1) << " " << c.NeuronOutput(2) << endl;
    for (double time = 0.0; time <= RunDuration; time += StepSize) {
        c.EulerStep(StepSize);
        cout << c.NeuronOutput(1) << " " << c.NeuronOutput(2) << endl;
    }

    // Finished
    return 0;
  }
```

5. Class Reference

```
// --- Utility functions
double sigmoid(double x);

double InverseSigmoid(double y);


// --- The constructor
CTRNN(int newSize = 0);


// --- State accessors
int CircuitSize(void);
void SetCircuitSize(int newSize);

double NeuronState(int neuronNumber);
double &NeuronStateReference(int neuronNumber);
void SetNeuronState(int neuronNumber, double value);

double NeuronOutput(int neuronNumber);
double &NeuronOutputReference(int neuronNumber);
void SetNeuronOutput(int neuronNumber, double value);

void RandomizeCircuitState(double lowerBound, double upperBound);
void RandomizeCircuitState(double lowerBound,
                           double upperBound,
                           RandomState &rs);
void RandomizeCircuitOutput(double lowerBound, double upperBound);
void RandomizeCircuitState(double lowerBound,
                           double upperBound,
                           RandomState &rs);


// --- Parameter accessors
double NeuronBias(int neuronNumber);
void SetNeuronBias(int neuronNumber, double value);

double NeuronGain(int neuronNumber);
void SetNeuronGain(int neuronNumber, double value);

double NeuronTimeConstant(int neuronNumber);
void SetNeuronTimeConstant(int neuronNumber, double value);

double NeuronExternalInput(int neuronNumber);
double &NeuronExternalInputReference(int neuronNumber);
void SetNeuronExternalInput(int neuronNumber, double value);

double ConnectionWeight(int fromNeuron, int toNeuron);
void SetConnectionWeight(int fromNeuron, int toNeuron, double value);
```

```
// --- Miscellaneous
void SetCenterCrossing(void);

void LesionNeuron(int neuronNumber);


// --- State Update
void EulerStep(double stepsize);

void RK4Step(double stepsize);
```

6. References

Beer, R.D. (2006). Parameter space structure of continuous-time recurrent neural networks. *Neural Computation* **18**:3009-3051.

Beer, R.D. (2003). The dynamics of categorical perception in an evolved model agent. *Adaptive Behavior* **11**(4):209-243.

Beer, R.D. (1995). On the dynamics of small continuous-time recurrent neural networks. *Adaptive Behavior* **3**:469-509.

Beer, R.D., Chiel, H.J. and Gallagher, J.C. (1999). Evolution and analysis of model CPGs for walking II. General principles and individual variability. *J. Computational Neuroscience* **7**:119-147.

Beer, R.D. and Gallagher, J.G. (1992). Evolving dynamical neural networks for adaptive behavior. *Adaptive Behavior* **1**:91-122.

Chiel, H.J., Beer, R.D. and Gallagher, J.C. (1999). Evolution and analysis of model CPGs for walking I. Dynamical modules. *J. Computational Neuroscience* **7**:99-118.

Chow, T.W.S. and Li, X.-D. (2000). Modeling of continuous time dynamical systems with input by recurrent neural networks. *IEEE Trans. On Circuits and Systems – I: Fundamental Theory and Applications* **47**:575-578.

Funahashi, K.I. and Nakamura, Y. (1993). Approximation of dynamical systems by continuous time recurrent neural networks. *Neural Networks* **6**:801-806.

Grossberg, S. (1988). Nonlinear neural networks: Principles, mechanisms, and architectures. *Neural Networks* **1**:17-61.

Hopfield, J.J. (1984). Neurons with graded response properties have collective computational properties like those of two-state neurons. *Proc. National Academy of Sciences* **81**:3088-3092.

Kimura, M. and Nakano, R. (1998). Learning dynamical systems by recurrent neural networks from orbits. *Neural Networks* **11**:1589-1599.

Mathayomchan, B. and Beer, R.D. (2002). Center-crossing recurrent neural networks for the evolution of rhythmic behavior. *Neural Computation* **14**:2043-2051.