The Neural Network Query Language (NNQL) Reference

Version 1.00 (pre-release)

Note: This is a pre-release document. All of these specifications are subject to change.

Contents

1 Motivation

Neural networks are computational devices inspired by the structure and function of the brain. They have been used to perform a wide variety of tasks that are difficult with traditional computer software, like image and voice recognition, and control. Despite advances in computer processing power, progress in neural networks their adoption has slowed. Neural networks in use today still do not have complex structures. And most don't even have multiple kinds of neurons.

One reason for these shortcomings is that there does not exist software that enables the easy construction of networks with arbitrary components and topologies. It is true that software for neural networks is readily available – for example, MATLAB provides an easy-to-use toolbox equipped with number of common neural network learning algorithms. However, these packages provide only the simplest, most common network types. Building arbitrary neural networks has always required programming.

A second obstacle to the propogation of diverse and interesting neural networks is the lack of a way to communicate them. Typically, once developed, a model could only be communicated in a way that obscured its structure: either via low-level source code, or by mathematical equations. Thus, understanding someone's neural network models meant figuring out their programming style and reading between the lines of their optimizations, or understanding their idiosyncratic mathematical conventions. This latter problem makes it perhaps unsurprising that very few neural network researchers use **each others'** models.

To solve these problems, it would be ideal to have a high-level language specific to the development of neural networks. This would help neural networks research advance because both development and communication of neural network models would be improved. This is the motivation behind NNQL.

2 Introduction

The Neural Network Query Language (NNQL) solves the problems mentioned above by providing a clear language that is designed specifically for developing and manipulating arbitrary neural networks. NNQL is based on a simplified SQL-like syntax, treating neural networks as though they are large databases.

Most programming languages require you to think about neural networks in terms of matrices and arrays. Instead NNQL allows you to think at the level of neurons and collections of neurons, and connections among them. It requires less time spent programming and reduces the likelihood of errors. NNQL code is faster and easier to write, and is perfect for developing and exploring novel neural network architectures.

And because of its text-based nature, NNQL is very precise, efficient, and communicable. Models become easy to replicate and to communicate to others. In a sense, NNQL can serve as a common language for neural networks researchers.

Finally, NNQL is useful for educational purposes, allowing students to study networks without the overhead, idiosyncrasies, and esoterica of various programming languages.

NNQL and its implementation, NNQLImpl, are released under the MIT open source license. The full text of this license is available in the appendix and in the source code distribution (in a file called LICENSE.txt).

2.1 Language Features

NNQL provides the following functionality:

- Current-based model neurons
- Spiking neurons (e.g., McCulloch-Pitts, Izhikevich)
- Numerous connectivity patterns, and fan-in / fan-out connection probability
- Axonal conduction delays
- Backpropogation of error (via a special "harness" attached to layers)
- Network simulation from mono audio (.WAV) files
- Network simulation from graphics (.PNG) files [soon]

NNQL can also easily extended to include additional features. We welcome the submission of source code patches to merge back into the NNQL distribution.

2.2 NNQLImpl: An NNQL Implementation

NNQLImpl is an efficient, multicore implementation of the NNQL language in Scala. It is:

- Cross-platform (runs on any Java VM).
- Optimized for multicore (specify how many cores you want to use).
- Minimal dependencies, easy to build/compile.

Other implementations are currently under development.

2.3 Requirements

Requirements for running NNQL simulations using NNQLImpl (the following should be placed on your CLASSPATH):

- Java (version 1.6 or higher)
- >= Apache commons-io-2.0.1.jar (http://commons.apache.org/io/)
- >= Apache commons-math-2.2.jar (http://commons.apache.org/math/)

2.3.1 Graphing

NNQL supports graphical output via R, which should be installed on the system if plotting is required from `http://www.r-project.org/`. NNQLImpl interacts with R using the RServe library. To install and start Rserve, use the following commands in R:

- install.packages(Rserve);
- library(Rserve);
- Rserve() [or Rserve(args='–no-save'); if you are running via a GUI]

NNQL can also create graphs of network structure using GraphViz from `http://www.graphviz.org/`.

2.4 General Usage

In general usage of NNQL, the modeler will create groups of neurons (called NUCLEI) and connect them to each other according to various patterns of connectivity. The modeler will then apply stimuli to these populations according to a pattern or parameters read from text files, and output from these populations can also be recorded from into text files.

Please visit the NNQL website or correspond with the author for more information on planned features in upcoming versions.

2.5 Credits

Patryk Laurent gratefully acknowledges contributions from:

- Joe Monaco (The Johns Hopkins University) contributions to much of the core code, including file processing and simulation code.
- Mark Huckvale (University College London) for audio spectrogram FFT code (audio input).

3 Installing NNQL

3.1 Binary Bundle

The easiest way to begin developing with NNQL to download a binary bundle. The binary bundle allows you to run all NNQL simulations and includes all of the dependencies for NNQL, including R plotting functions.

To install the NNQL Binary Bundle, follow these steps:

1. Make sure that Java[1] and Scala[2] are installed.
2. Obtain the NNQL Bundle, NNQLBundle.jar (`http://nnql.org/NNQLBundle.jar`)
3. Open a terminal and place NNQLBundle.jar on your CLASSPATH, e.g.,:
   `export CLASSPATH=$CLASSPATH:~/Desktop/NNQLBundle.jar:.`

You're done. Simply compile your NNQL scripts by typing `scalac MyScript.scala` and run them by typing `scala MyScript`.

3.2 Source Code

If you would like to make change to the NNQL language itself, you should download the source code.

Instructions for this will be forthcoming.

[1]To install Java on a Mac, just type "java" in a terminal. If it's not installed Mac OS X will guide you in the steps to installing it.
[2]Obtain Scala from `http://www.scala-lang.org/downloads` and make sure the bin/ directory of Scala is on your PATH.

4 NNQL Examples

4.1 Example 1 (Two-layer recurrent spiking network)

The following simulates a two-layer recurrent network composed of phasic spiking neurons[3].

```
    CREATE NUCLEI ("V1", "V2")
    INSERT NEURON IZ("phasic_spiking") INTO "V1" QUANTITY 30
    INSERT NEURON IZ("phasic_spiking") INTO "V2" QUANTITY 400
    PROJECT FULLY FROM NUCLEUS "V1" TO "V2" WEIGHTS U(0.0,1.0) DELAY 35
    PROJECT FULLY FROM NUCLEUS "V2" TO "V1" WEIGHTS N(1.0,0.5) DELAYS I(15,20)
    COMPLETE MODEL

    RECORD FROM "V2" INTO "V2Traces"
    RECORD FROM "V1" INTO "V1Traces"
```

Once the network model has been created, it can be stimulated with input current. The current applied to each neuron could be specified in an external text file created before running the simulation (this is shown in later examples). Alternatively, NNQL provides a number of commands to generate input patterns. The following stimulates the network with a sequence of input currents:

```
SEQ_GENERATOR N(100) SIZE(20) STUTTER(5) SHIFT(20)
    SEQLEN(5) CURRENT(60) FILENAME("seq.txt") RUN
STIMULATE NUCLEUS "V1" FROM "seq.txt"
```

This creates a file named `seq.txt`, containing space-separated rows of numbers. Each row is a time-step of stimulation, and each number (column) represents the current to inject into each ordered neuron in the nucleus. To run and graph the network, and then display population raster histograms (like the one on page 41):

```
EXECUTE MODEL 1000;
LIST DOTDIAGRAM(threshold=6.5,scale=0.5) SHOW;   // Uses GraphViz to visualize network
EXPORT TO "./output"
ALL_RECORDERS makePopRaster;
```

Note that `LIST DOTDIAGRAM` requires that GraphViz be installed.

[3]Note: This example uses the abbreviated or "shorthand" form of weight distribution specifications. The use of `WEIGHTS U(x,y)` and `WEIGHTS N(x,y)` replaces full-length lines like `WEIGHTS DISTRIBUTION UNIFORM FROM x TO y`, and `WEIGHTS DISTRIBUTION NORMAL FROM x TO y` that could have been written above each `PROJECT` statement.

4.2 Example 2 (A multilayer perceptron)

The following creates a three-layer network appropriate for using backpropagation of error:

```
    CREATE NUCLEI ("L1","L2","L3")
    INSERT NEURON mock    INTO "L1" QUANTITY 5
    INSERT NEURON sigmoid INTO "L2" QUANTITY 5
    INSERT NEURON sigmoid INTO "L3" QUANTITY 5
    PROJECT FULLY FROM NUCLEUS "L1" TO "L2" WEIGHTS N(0.1,0.1)
    PROJECT FULLY FROM NUCLEUS "L2" TO "L3" WEIGHTS N(0.1,0.1)
    COMPLETE MODEL;
```

NNQL provides a backpropagation of error training algorithm in the form of a virtual "harness" that is attached to the multilayer perceptron to train it. The "harness" also supports a context layer for training Simple Recurrent Networks (SRNs), see the Simple Recurrent Network example.

```
    HARNESS(1) ATTACH ("L1", "L2", "L3");
    LOOP (10000) {
        HARNESS(1) FEEDFORWARD(1,1,1,0,0) BACKPROP(0,0,1,1,1);
        HARNESS(1) FEEDFORWARD(0,1,1,0,1) BACKPROP(1,0,0,0,1);
    }
    print("Should output 00111: "); HARNESS(1) PROBE(1,1,1,0,0);
    print("Should output 10001: "); HARNESS(1) PROBE(0,1,1,0,1);
```

The "harness" also provides a convenience method for stimulating the network with test patterns. However, this is not necessary: instead, one could `STIMULATE NUCLEUS "L1"` with the appropriate text file and record the output from the `L3` nucleus.

Note: To add a bias unit, which is often important in these kinds of networks, one can use:

```
INSERT NEURON mock INTO "L1" QUANTITY 1 LABELED "BIAS"
PROJECT FULLY FROM UNIT "L1 BIAS" TO "L3" WEIGHTS N(0.0,0.01)
```

4.3 Example 3 (XOR Backprop)

```
import nnql.NNQL._;

object TestXOR extends App {
    CREATE NUCLEI ("L1","L2","L3")
    INSERT NEURON mock    INTO "L1" QUANTITY 1 LABELED "BIAS" // Bias unit is helpful.
    INSERT NEURON mock    INTO "L1" QUANTITY 2
    INSERT NEURON sigmoid INTO "L2" QUANTITY 2
    INSERT NEURON sigmoid INTO "L3" QUANTITY 1
    PROJECT FULLY FROM NUCLEUS "L1" TO "L2" WEIGHTS N(0.0,0.01)
    PROJECT FULLY FROM NUCLEUS "L2" TO "L3" WEIGHTS N(0.0,0.01)
    PROJECT FULLY FROM NUCLEUS "L1 BIAS" TO "L3" WEIGHTS N(0.0,0.01) // Bias unit connect.
    COMPLETE MODEL;

    HARNESS(1) ATTACH ("L1", "L2", "L3");
    HARNESS(1) ADDTRIAL( List(-1,  1,1), List(0) ); // XOR 11 -> FALSE
    HARNESS(1) ADDTRIAL( List(-1,  1,0), List(1) ); // XOR 10 -> TRUE
    HARNESS(1) ADDTRIAL( List(-1,  0,1), List(1) ); // XOR 01 -> TRUE
    HARNESS(1) ADDTRIAL( List(-1,  0,0), List(0) ); // XOR 00 -> FALSE

    HARNESS(1) LEARNINGRATE 0.05
    LOOP (400000) {
        HARNESS(1) SHUFFLETRIALS EXECUTE
        EVERY (10000) { HARNESS(1) PRINTSSE }
    }

    HARNESS(1) PROBE(-1,  1,1);  // Should -> 0
    HARNESS(1) PROBE(-1,  1,0);  // Should -> 1
    HARNESS(1) PROBE(-1,  0,1);  // Should -> 1
    HARNESS(1) PROBE(-1,  0,0);  // Should -> 0

    LIST DOTDIAGRAM(0,0.1) SHOW;   // Uses GraphViz to visualize network
    EXPORT TO "./output"
}
```

4.4 Example 4 (Recurrent spiking McCulloch-Pitts STDP net)

```
import nnql.NNQL._;

object RecurrentLearningNet extends App {
    // (1) Build the recurrent network model.
    CREATE NUCLEUS "CA3" COMPETITIVE "10%"
    INSERT NEURON McCullochPitts INTO "CA3" QUANTITY 2048
    PROJECT FAN_IN FROM NUCLEUS "CA3" TO "CA3" PROBABILITY 0.08 WEIGHTS E(0.4) PLASTICITY "LEVY"
    UPDATE NUCLEUS "CA3" SET "ALPHA" TO "0.5"
    COMPLETE MODEL

    // (2) Train the model on a sequence.
    UPDATE NUCLEUS "CA3" SET "SYNMODRATE" TO "0.003"
    SEQ_GENERATOR N(1024) SIZE(20) STUTTER(3) SHIFT(20) SEQLEN(5) FILENAME("sequence.txt") RUN;
    STIMULATE NUCLEUS "CA3" FROM "sequence.txt" POFFNOISE 0.10
    LOOP (100) {
        RECENT_STIMULATOR rewind;
        EXECUTE MODEL 40;
        print(".")
    }

    // (3) Start recording.  Present the sequence so we can see how the network fires.
    RECORD FROM "CA3" INTO "CA3Traces";
    UPDATE NUCLEUS "CA3" SET "SYNMODRATE" TO "0.0"
    RECENT_STIMULATOR rewind;
    EXECUTE MODEL 50;
    RECENT_STIMULATOR deactivate;

    // (4) Now, present just the first pattern of the sequence to test recall.
    SEQ_GENERATOR N(1024) SIZE(20) STUTTER(3) SHIFT(20) SEQLEN(1) FILENAME("testprobe.txt") RUN;
    STIMULATE NUCLEUS "CA3" FROM "testprobe.txt" POFFNOISE 0.10
    EXECUTE MODEL 50;
    ALL_RECORDERS makePopRaster;
}
```

An example of visual output from the simulation is shown in Figure 1 on page 12

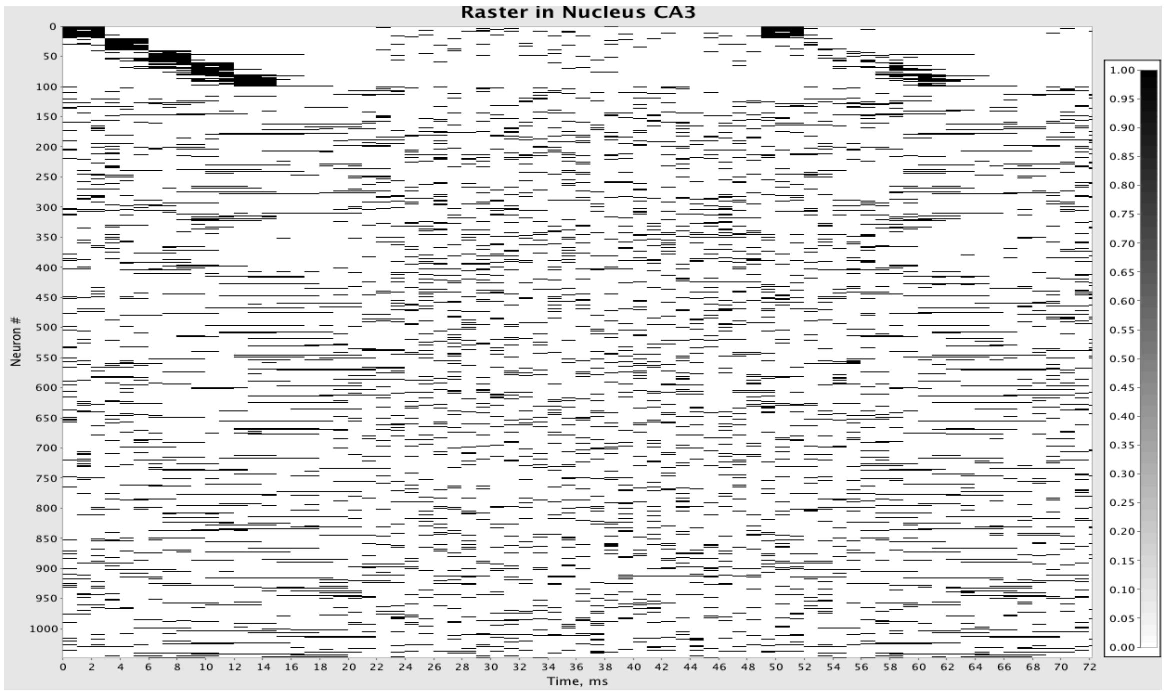

Figure 1: **Spiking Output of Example Recurrent Spiking McCulloch-Pitts Network.** On the left is an instance of the training sequence. On the right, only the first pattern of the sequence is presented to the network. The network then recalls the sequence of activated input patterns. Output was generated using the *makePopRaster* command. Each short horizontal bar is a spike.

4.5 Example 5 (Simple Recurrent Network)

A simple recurrent network can use its recurrent connections as a sort of working memory, as demonstrated by the following example. First, we create two pairs of files, representing two desired input sequences and two desired output sequences, respectively.

srnseqinp1.txt

```
1 1 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

srnseqinp2.txt

```
0 0 0 1 1
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

srnseqout1.txt

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
1 1 0 0 0
```

srnseqout2.txt

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 1 1
```

Next, we create a Simple Recurrent Network (SRN). An SRN is a multilayer perceptron which has a "context layer" that copies the contents of the hidden layer on each time-step.

```
    CREATE NUCLEI ("L1","LC","L2","L3")
    INSERT NEURON mock    INTO "L1" QUANTITY 5
    INSERT NEURON mock    INTO "LC" QUANTITY 10
    INSERT NEURON sigmoid INTO "L2" QUANTITY 10
    INSERT NEURON sigmoid INTO "L3" QUANTITY 5
    PROJECT FULLY FROM NUCLEUS "L1" TO "L2" WEIGHTS N(0.01,0.1)
    PROJECT FULLY FROM NUCLEUS "LC" TO "L2" WEIGHTS N(0.01,0.1)
    PROJECT COPY  FROM NUCLEUS "L2" TO "LC" WEIGHTS E(1.0)
    PROJECT FULLY FROM NUCLEUS "L2" TO "L3" WEIGHTS N(0.01,0.1)
    COMPLETE MODEL;
```

Notice that a COPY projection must be made from Layer 2 (L2) to the Context Layer (LC)[4].

Using the same kind of backprop harness as was used for the multilayer perceptron, we train the SRN on the two pairs of sequences:

```
    HARNESS(1) ATTACH ("L1", "LC", "L2", "L3");
    HARNESS(1) LEARNINGRATE 0.01
    HARNESS(1) ADDTRIAL("srnseqinp1.txt", "srnseqout1.txt");
    HARNESS(1) ADDTRIAL("srnseqinp2.txt", "srnseqout2.txt");

    LOOP (500000) {
        HARNESS(1) SHUFFLETRIALS EXECUTE;
        EVERY (10000) { HARNESS(1) PRINTSSE }
    }
```

Test the network to see if it was able to maintain the information over time.

```
    println("Probe 11000 on input, expect to get 11000 on output in 3 ticks.");
    HARNESS(1) CLEAR; HARNESS(1) PROBEFILE ("srnseqinp1.txt");
    println("^^ should be ~ 11000");

    println("Probe 00011 on input, expect to get 00011 on output in 3 ticks.");
    HARNESS(1) CLEAR; HARNESS(1) PROBEFILE ("srnseqinp2.txt");
    println("^^ should be ~ 00011");
```

The structure of the network can be graphed using `LIST DOTDIAGRAM`, provided that GraphViz is installed.

[4]The harness needs to be made aware of the context layer so that it can clear it between sequence presentations. However, the harness does not automatically copy the `hidden` unit activations to the `context` layer. Therefore, for correct functioning, the SRN's context layer in the SRN must receive a projection from the hidden units (see the use of `PROJECT COPY` in the simple recurrent network example) of the hidden layer activations.

4.6 Example 6 (Simple Recurrent Network, no files)

Here is a version of the simple recurrent net from before, but without dependence on external files.

```
import nnql.NNQL._;

object NNQL_TestBpHarnessSRN0 extends App {
    CREATE NUCLEUS "L1"   // input   layer
    CREATE NUCLEUS "LC"   // context layer
    CREATE NUCLEUS "L2"   // hidden  layer
    CREATE NUCLEUS "L3"   // output  layer
    INSERT NEURON mock    INTO "L1" QUANTITY 5
    INSERT NEURON mock    INTO "LC" QUANTITY 6
    INSERT NEURON sigmoid INTO "L2" QUANTITY 6
    INSERT NEURON sigmoid INTO "L3" QUANTITY 5
    PROJECT FULLY FROM NUCLEUS "L1" TO "L2" WEIGHTS N(0.1,0.4)
    PROJECT FULLY FROM NUCLEUS "LC" TO "L2" WEIGHTS N(0.1,0.4)
    PROJECT COPY FROM NUCLEUS "L2" TO "LC" WEIGHTS E(1.0)  // these are "constant" weights.
    PROJECT FULLY FROM NUCLEUS "L2" TO "L3" WEIGHTS N(0.1,0.1)
    COMPLETE MODEL;

    HARNESS(1) ATTACH ("L1", "LC", "L2", "L3");
    HARNESS(1) LEARNINGRATE 0.01

    HARNESS(1) ADDTRIAL ( () => {
        HARNESS CLEAR;
        HARNESS(1) FEEDFORWARD(1,1,0,0,0) BACKPROP(0,0,0,0,0);
        HARNESS(1) FEEDFORWARD(0,0,0,0,0) BACKPROP(0,0,0,0,0); // Delay period
        HARNESS(1) FEEDFORWARD(0,0,0,0,0) BACKPROP(0,0,0,0,0); // Delay period
        HARNESS(1) FEEDFORWARD(0,0,0,0,0) BACKPROP(1,1,0,0,0);
    })

    HARNESS(1) ADDTRIAL ( () => {
        HARNESS(1) CLEAR;
        HARNESS(1) FEEDFORWARD(0,0,0,1,1) BACKPROP(0,0,0,0,0);
        HARNESS(1) FEEDFORWARD(0,0,0,0,0) BACKPROP(0,0,0,0,0); // Delay period
        HARNESS(1) FEEDFORWARD(0,0,0,0,0) BACKPROP(0,0,0,0,0); // Delay period
        HARNESS(1) FEEDFORWARD(0,0,0,0,0) BACKPROP(0,0,0,1,1);
    })

    LOOP (400000) {
        HARNESS(1) SHUFFLETRIALS;
        HARNESS(1) EXECUTE;
        EVERY (10000) { HARNESS(1) PRINTSSE }
    }

    println("Probe 11000 on input, expect to get 11000 on output in 3 ticks.");
    HARNESS(1) CLEAR;
    HARNESS(1) PROBE (1,1,0,0,0);
    HARNESS(1) PROBE (0,0,0,0,0);
    HARNESS(1) PROBE (0,0,0,0,0);
    HARNESS(1) PROBE (0,0,0,0,0); println("^^ should be ~ 11000");

    println("Probe 00011 on input, expect to get 00011 on output in 3 ticks.");
    HARNESS(1) CLEAR;
    HARNESS(1) PROBE (0,0,0,1,1);
    HARNESS(1) PROBE (0,0,0,0,0);
    HARNESS(1) PROBE (0,0,0,0,0);
    HARNESS(1) PROBE (0,0,0,0,0); println("^^ should be ~ 00011");

    LIST DOTDIAGRAM(0.45,0.6) SHOW;
    EXPORT TO "./output"
}
```

5 NNQL Statement Syntax

The first section, entitled Network Definition Statements, describes commands that are used to define a network, such as `CREATE`, `INSERT`, and `COMPLETE`.

The next section describes Network Manipulation statements, which include commands that alter the structure or composition of an existing network.

The following section describes Input and Output statements that are important for moving data into and out of the simulation. These include commands such as `STIMULATE` and `RECORD`. It will also include commands that enable protocols for communication with external visualization programs.

5.1 Network Definition Statements

The commands in this section are used to create neural networks.

5.1.1 CREATE NUCLEUS Syntax

```
CREATE NUCLEUS "name"
CREATE NUCLEUS "name" COMPETITIVE nn%
```

PURPOSE

A Nucleus is a collection of neurons of potentially different types. The `CREATE NUCLEUS` command creates a nucleus into which neurons can be `INSERT`ed or between which `PROJECT`ions can be made.

For example:

```
CREATE NUCLEUS ″V1″
CREATE NUCLEUS ″V2″
CREATE NUCLEUS ″MT″
```

or

```
CREATE NUCLEI (″V1″, ″V2″, ″MT″)
```

creates three nuclei.

PARAMETERS

`COMPETITIVE`	Specifies the maximum number of neurons that can be co-active at any given time-step. If more than that number of neurons are active, those with the highest activity level (i.e., membrane voltage) are taken to be “winners” and are allowed to be active on that time-step.

5.1.2 INSERT NEURON Syntax

```
INSERT neuronType INTO "nucleus" [QUANTITY quantity] [LABELED "label"]
```

PURPOSE

This command instantiates simulated neurons of the given type and associates them with a nucleus.

PARAMETERS

`neuronType`	The type of neuron. There are several specific kinds of neurons built into NNQL: Izhikevich, Sigmoid, mock (Linear), McCulloch-Pitts (e.g., Linear with a threshold).
`QUANTITY`	The number of neuron units to insert.
`LABELED`	Label to be associated with the created neurons. This should be a single word without spaces. (Future versions may allow multiple labels to be specified with spaces.)

Neuron types

The Izhikevich class of integrate-and-fire neurons can be inserted using the following syntax:

```
INSERT NEURON izhikevich(a,b,c,d,defaultI) INTO nucleus QUANTITY quantity
INSERT NEURON izhikevich("type") INTO nucleus QUANTITY quantity
INSERT NEURON IZ(a,b,c,d,defaultI) INTO nucleus QUANTITY quantity
INSERT NEURON IZ("type") INTO nucleus QUANTITY quantity
```

An Izhikevich neuron implements neurons with the following equations:

$$v' = 0.04v^2 + 5v + 140 - u + I \quad (1)$$

$$u' = a(bv - u) \quad (2)$$

with after-spike resetting,

$$\text{if } v \geq +30\text{mV, then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d. \end{cases} \quad (3)$$

where “type” selects the following parameters for a,b,c and d in the equations as provided by Izhikevich (2004):

```
"izhikevich_tonic_spiking"      a= 0.02;  b= 0.2;  c= -65; d= 6;    defaultI= 14;
"izhikevich_phasic_spiking"     a= 0.02;  b= 0.25; c= -65; d= 6;    defaultI= 0.5;
"izhikevich_tonic_bursting"     a= 0.02;  b= 0.2;  c= -50; d= 2;    defaultI= 15;
"izhikevich_phasic_bursting"    a= 0.02;  b= 0.25; c= -55; d= 0.05; defaultI= 0.6;
"izhikevich_mixed_mode"         a= 0.02;  b= 0.2;  c= -55; d= 4;    defaultI= 10;
"izhikevich_spike_freq_adapt"   a= 0.01;  b= 0.2;  c= -65; d= 8;    defaultI= 30;
"izhikevich_class_1"            a= 0.02;  b= -0.1; c= -55; d= 6;    defaultI= 0;
"izhikevich_class_2"            a= 0.2;   b= 0.26; c= -65; d= 0;    defaultI= 0;
"izhikevich_spike_latency"      a= 0.02;  b= 0.2;  c= -65; d= 6;    defaultI= 7;
"izhikevich_subthresh_osc"      a= 0.05;  b= 0.26; c= -60; d= 0;    defaultI= 0;
"izhikevich_resonator"          a= 0.1;   b= 0.26; c= -60; d= -1;   defaultI= 0;
"izhikevich_integrator"         a= 0.02;  b= -0.1; c= -55; d= 6;    defaultI= 0;
"izhikevich_rebound_spike"      a= 0.03;  b= 0.25; c= -60; d= 4;    defaultI= 0;
"izhikevich_rebound_burst"      a= 0.03;  b= 0.25; c= -52; d= 0;    defaultI= 0;
"izhikevich_thresh_var"         a= 0.03;  b= 0.25; c= -60; d= 4;    defaultI= 0;
"izhikevich_bistability"        a= 1;     b= 1.5;  c= -60; d= 0;    defaultI= -65;
"izhikevich_DAP"                a= 1;     b= 0.2;  c= -60; d= -21;  defaultI= 0;
"izhikevich_acommodation"       a= 0.02;  b= 1;    c= -55; d= 4;    defaultI= 0;
"izhikevich_inhib_i_spike"      a= -0.02; b= -1;   c= -60; d= 8;    defaultI= 80;
"izhikevich_inhib_i_burst"      a= -.026; b= -1;   c= -45; d= 0;    defaultI= 80;
```

A linear neuron (that is, one without a non-linear activation function) can be inserted using the following syntax:

INSERT NEURON mock INTO $nucleus$ QUANTITY $quantity$

The linear neuron performs a weighted sum of its inputs on a given time-step, and provides the result on its output.

$$y_j(t) = \sum_{i=1}^{n} w_{ij} \cdot Z_i(t-1). \tag{4}$$

A McCulloch-Pitts neuron can be inserted using the following syntax:

INSERT NEURON McCullochPitts($threshold$) INTO $nucleus$ QUANTITY $quantity$
INSERT NEURON McP($threshold$) INTO $nucleus$ QUANTITY $quantity$

It has the same activation function as the linear neuron:

$$y_j(t) = \sum_{i=1}^{n} w_{ij} \cdot Z_i(t-1). \tag{5}$$

But its output is thresholded so that firing occurs if the excitation exceeds a threshold θ_t :

$$Z_j(t) = \begin{cases} 1 & \text{if } y_j(t) > \theta_t \text{ or } x_j(t) = 1 \\ 0 & \text{otherwise,} \end{cases} \tag{6}$$

5.1.3 WEIGHTS DISTRIBUTION Syntax

The `WEIGHTS` statement sets a globally accessible specification for the values of synaptic weights. Subsequent `PROJECT` statements will use this specification for creating connections by default (this can be overriden by an in-line WEIGHTS keyword, see Section 5.1.4).

There are two equivalent ways to specify fixed (constant) weights:

```
WEIGHTS ALL value
WEIGHTS DISTRIBUTION EQUAL TO value
```

These statements specify randomly distributed weights with optional bounds:

```
WEIGHTS DISTRIBUTION distributionName PARAM1 value ...
WEIGHTS DISTRIBUTION NORMAL MEAN 1.0 STD 0.5
WEIGHTS DISTRIBUTION NORMAL MEAN 1.0 STD 0.5 LOWER 0.0
WEIGHTS DISTRIBUTION UNIFORM FROM 0.0 TO 1.0
```

NOTE: Weight distributions can be specified as part of a `PROJECT` statement, and a short form (i.e., abbreviation) for each weight distribution can be used in those statements. See `PROJECT` Syntax, section 5.1.4 on page 24.

PURPOSE

Sets the distribution of synaptic weights for subsequent projections made using `PROJECT` statements.

PARAMETERS

`EQUAL TO`	All of the weights are set to the same given constant value. However, these projections are not fixed to that constant value and can be changed by synaptic modification.
`NORMAL`	This keyword describes a normal distribution. It has parameters, `MEAN` (or `MU`) and `STD` (or `SIGMA`).
`UNIFORM`	This keyword describes a continuous uniform distribution. It has parameters `FROM` and `TO`.
`INTEGERS`	This keyword describes a discrete uniform distribution on the integers. It has parameters `FROM` and `TO`.
`GAMMA`	This keyword describes a gamma distribution. It has parameters `SCALE` and `SHAPE`.
`LOWER`	This keyword sets a lower bound for weights (ignored for `DISTRIBUTION EQUAL`). The bound is enforced by resampling, and a warning is printed if resampling exceeds 200 tries without finding a valid weight value.
`UPPER`	This keyword has the same behavior as `LOWER`, but sets an upper bound for weights.

5.1.4 PROJECT Syntax

PROJECT FULLY FROM NUCLEUS "$nucleus1$" TO "$nucleus2$"
PROJECT COPY FROM NUCLEUS "$nucleus1$" TO "$nucleus2$" WEIGHTS E(1.0)
PROJECT FULLY FROM NUCLEUS "$nucleus1$" TO "$nucleus2$" [DELAY 50]
PROJECT FULLY FROM NUCLEUS "$nucleus1$" TO "$nucleus2$" [WEIGHTS N($mean$, std)]
PROJECT FULLY FROM NUCLEUS "$nucleus1$" TO "$nucleus2$" [WEIGHTS E($value$)]
PROJECT FULLY FROM UNIT 1001 TO "$nucleus2$" [WEIGHTS E($value$)]
PROJECT FAN_IN FROM NUCLEUS "$n1$" TO "$n2$" PROBABILITY 0.10 WEIGHTS E(0.4) PLASTICITY "LEVY"
PROJECT FAN_IN FROM NUCLEUS "$n1$" TO "$n2$" PROBABILITY 0.10 WEIGHTS E(0.4) PLASTICITY "OJA"

PURPOSE

Establishes a feedforward projection from nucleus1 to nucleus2.

PARAMETERS

`DELAY`	The `DELAY` parameter adds constant conduction delay of the specified value (in terms of ticks, i.e., the time-step of integration) to each projection. If the time-step of integration of the simulation is set to 1 ms, then `DELAY 50` introduces a 50 millisecond time-step delay.
`DELAYS`	The `DELAYS` parameter provides randomly distributed delays for this projection. `DELAYS` takes a shorthand notation for the random distribution to be used to sample the conduction delays (see shorthand definition for `WEIGHTS` below). Note that sampled delays are cast to integers without rounding.
`WEIGHT`	The `WEIGHT` parameter specifies a fixed weight value for this projection. That is, `WEIGHT` $value$ is a synonym for `WEIGHTS` E($value$), using the distribution shorthand described below for `WEIGHTS`.
`PROBABILITY`	The `FAN_IN` and `FAN_OUT` projection types require a specification of the `PROBABILITY`. For example, a `FAN_IN` probability of 10% means that each postsynaptic neuron receives projections from 10% of the presynaptic neurons.

`WEIGHTS`	The `WEIGHTS` parameter provides a weights definition specific to this projection. This overrides the current global weights definition (if any) within the scope of the current `PROJECT` statement. The `WEIGHTS` parameter takes a shorthand notation for specifying either fixed or randomly distributed weights. The distribution is specified by N (normal), E (equal), U (uniform) or G (gamma). Each distribution takes parameters as follows: N(mean,std), E(value), U(lowerBound, upperBound), and G(scale,shape).
`PLASTICITY`	The PLASTICITY parameter, if present, indicates the kind of synaptic modification rule that is present. Parameters for the plasticity, like a learning rate constant or time-span of associativity, can be set at the level of the nucleus. ("LEVY" or its synonym "TEMPASSYMHEBB"; "OJA".)

PROJECTION TYPES

The `COPY` projection type provides a 1-to-1 projection mapping. It therefore requires that the source and target nuclei have the same number of neurons. This is useful for setting up context layers in Simple Recurrent Networks (also called "Elman networks").

Other connectivity patterns will be implemented (currently, only `FULLY` , `COPY` and `FAN_IN` are implemented.) Some of these connectivity patterns include:

PROJECT RANDOMLY FROM NUCLEUS "$nucleus1$" TO "$nucleus2$" PROBABILITY $percentage$

PROJECT TOPOGRAPHICALLY FROM NUCLEUS "$nucleus1$" TO "$nucleus2$" PROBABILITY $percentage$

PROJECT LOCALLY FROM NUCLEUS "$nucleus1$ TO "$nucleus2$" PROBABILITY $percentage$

PROJECT UNIFORMLY FROM NUCLEUS "$nucleus1$" TO "$nucleus2$" PROBABILITY $percentage$

PLASTICITY: LEVY or TEMPASSYMHEBB

`LEVY` or its synonym `TEMPASSYMHEBB` plasticity specifies a temporally-asymmetric Hebbian-like learning rule inspired by spike-timing dependent plasticity or STDP, which is a temporal contiguity requirement for long-term potentiation (Bi and Poo, 1998; Levy and Steward, 1983; Markram et al., 1997). Each weight from neuron i to neuron j is updated only on time-steps when the post-synaptic neuron j is active (i.e., $Z_j = 1$). The update depends on the difference between the current weight w_{ij} and a running average of afferent presynaptic activity,

$$w_{ij}(t+1) = w_{ij}(t) + \mu \cdot Z_j(t)\Big(\bar{Z}_i(t-1) - w_{ij}(t)\Big), \tag{7}$$

where the running averager $\bar{Z}_i(t-1)$ saturates when the presynaptic neuron fires, and decreases exponentially otherwise, i.e.,

$$\bar{Z}_i(t) = \begin{cases} 1 & \text{if } Z_i(t) = 1, \\ \alpha \cdot \bar{Z}_i(t-1) & \text{otherwise.} \end{cases} \tag{8}$$

The use of a running average of afferent presynaptic activity for weight modification is intended to mimic a saturate-and-decay model of the NMDA receptor. Equation 4 indicates that the running average of firing activity is updated on every time-step for each neuron i. The quantity α is a time-constant of the decay (i.e., of the modeled NMDA receptors) (NNQL: `NUCLEUS ``ALPHA''`). The parameter $\mu$ is a synaptic weight modification rate constant or "learning rate" (NNQL: `NUCLEUS ``SYNMODRATE''`).

PLASTICITY: OJA

`OJA` plasticity specifies a normalized Hebbian (co-activation) learning rule that is guaranteed to be stable. The present implementation assumes the neuron's activations are linear weighted sums of their inputs.

$$\Delta \mathbf{w} \;=\; \mathbf{w}_{n+1} - \mathbf{w}_n \;=\; \eta\, y_n(\mathbf{x}_n - y_n \mathbf{w}_n), \tag{9}$$

According to Wikipedia (page last updated April 20, 2011), this is derived by taking a standard Hebbian rule:

$$w_i(n+1) \;=\; w_i + \eta\, y(\mathbf{x}) x_i \tag{10}$$

and implementing normalization so the afferent weights are restricted in the range of 0 to 1 (0 means no afferent weight, 1 means the only one afferent neuron with a weight):

$$w_i(n+1) \;=\; \frac{w_i + \eta\, y(\mathbf{x}) x_i}{\left(\sum_{j=1}^{m} [w_j + \eta\, y(\mathbf{x}) x_j]^p\right)^{1/p}} \tag{11}$$

which can be factored into:

$$w_i(n+1) \;=\; \frac{w_i}{\left(\sum_j w_j^p\right)^{1/p}} \;+\; \eta \left(\frac{y x_i}{\left(\sum_j w_j^p\right)^{1/p}} - \frac{w_i \sum_j y x_j w_j}{\left(\sum_j w_j^p\right)^{(1+1/p)}} \right) \;+\; O(\eta^2) \tag{12}$$

This equation simplifies to Oja's rule :

$$w_i(n+1) \ = \ w_i + \eta \, y(x_i - w_i y) \tag{13}$$

if we make the following three assumptions:

(A) we have a small learning rate where the O(n2) higher order terms go to zero,

$$O(\eta^2) = 0 \tag{14}$$

(B) we have linear weighted neurons, that is, the output of the neuron is equal to the sum of the product of each input and its synaptic weight, or

$$y(\mathbf{x}) \ = \ \sum_{j=1}^{m} x_j w_j \tag{15}$$

(C) we also specify that our weights normalize to 1 (which will be a necessary condition for stability)

$$|\mathbf{w}| \ = \ \left(\sum_{j=1}^{m} w_j^p \right)^{1/p} \ = \ 1 \tag{16}$$

5.1.5 COMPLETE MODEL Syntax

```
COMPLETE MODEL
```

PURPOSE

This command ensures that the model is completely assembled prior to executing it. (Although currently required, this command will be deprecated in future versions of NNQL.)

PARAMETERS

None.

5.2 Network Manipulation Statements

The commands in this section are used to manipulate neural networks once they have been created.

5.2.1 EXECUTE MODEL Syntax

```
EXECUTE MODEL NUMTICKS
```

PURPOSE

This command executes the model for the specified number of ticks, e.g., EXECUTE MODEL 1000.

While a model is executed, all connected Stimulators and Recorders are updated. That is, on every time-step, each Stimulator stimulates the neurons to which it is connected. Similarly, each Recorder records a sample from the neurons to which it is connected.

PARAMETERS

`NUMTICKS`	The number of ticks to run the model for.

5.2.2 UPDATE NUCLEUS Syntax

```
UPDATE NUCLEUS “name” SET “parameter” TO “n”
```

PURPOSE

Sets a parameter that applies to all units within the nucleus.

For example:

```
UPDATE NUCLEUS ”CA3” SET ”SYNMODRATE” TO ”0.005”
```

PARAMETERS

`SYNMODRATE`	Synaptic modification rate for learning mechanisms like STDP.
`ALPHA`	Time constant for associative synaptic modification in STDP.

5.2.3 HARNESS Syntax

HARNESS ("NAME1") HARNESS ("NAME1") ATTACH ($''inputLayer''$, $''hiddenLayer''$, $''outputLayer''$)
HARNESS ("NAME1") ATTACH ($''inputLayer''$, $''contextLayer''$, $''hiddenLayer''$, $''outputLayer''$)
HARNESS ("NAME1") CLEAR
RECENT_HARNESS CLEAR
RECENT_HARNESS LEARNINGRATE $n.nn$
RECENT_HARNESS ADDTRIAL ($''inputPattern.txt''$, $''outputPattern.txt''$)
RECENT_HARNESS ADDTRIAL (List(a,b,c,d), List(x,y,z,w))
RECENT_HARNESS SHUFFLETRIALS
RECENT_HARNESS EXECUTE
RECENT_HARNESS PROBE (List(a,b,c,d))
HARNESS ("NAME1") PROBEFILE ($''filename''$)
RECENT_HARNESS PRINTSSE

PURPOSE

Registers a set of nuclei for learning via backpropagation of error. A harness can be attached either to a three-layer perceptron or a simple recurrent network.

5.3 Data Input and Output Statements

This commands in this section describe how data is moved into and out of the simulations.

Additional commands that will be described in this section (as they are implemented) include EVENT streams, TCP/IP modular connections, audio file input/output, and image sequence input/output.

5.3.1 DESCRIBE Syntax

```
DESCRIBE nucleus
```

PURPOSE

This command prints out information about a nucleus, including the properties of the neurons that can be SELECTed.

PARAMETERS

None.

5.3.2 SELECT Syntax

```
SELECT (VOLTAGE) FROM "nucleus" NOW
SELECT (VOLTAGE,SPIKED) FROM "nucleus" NOW
SELECT (VOLTAGE,SPIKED) FROM "nucleus" INTO "results.txt"
```

PURPOSE

This command prints out data from the neurons in a nucleus, including its neuronal contents, and average connectivity.

PARAMETERS

None.

5.3.3 STIMULATE NUCLEUS Syntax

```
STIMULATE NUCLEUS "nucleus1" FROM "filename"
STIMULATE NUCLEUS "nucleus1" FROM "filename" POFFNOISE 0.10
STIMULATE NUCLEUS "nucleus1" FROM "monoSoundFile.WAV"
STIMULATE NUCLEUS "nucleus1" FROM "image.PNG"
STIMULATE NUCLEUS "nucleus1" FROM "image.PNG@widthxheight"
STIMULATE NUCLEUS "nucleus1" FROM "image.png@20x20"
STIMULATOR "sound" ADD (0,0,1,0,0,0,1,1,1,0,0)
STIMULATE NUCLEUS "nucleus1" NAMED "pairs"
STIMULATOR "pairs" ADD(1,0) ADD(1,1) ADD(0,0) ADD (1,0)
```

A Stimulator provides input (clamping current) to the associated neurons in the specified Nucleus. If the input is from a text file, each column of the file becomes associated with each neuron N (for as many columns C are in the file; C can be smaller than N). If there are more columns than neurons, only the first columns are used. If there are more neurons than columns, all of the columns are used and the remainder of the neurons in the nucleus are not stimulated by the contents of the file.

Images. Stimulators support loading of PNG image graphics; these are recognized by having the .PNG extension. Note that NNQL currently converts the image to grayscale using the formula: 30% of red, 59% of green, 11% of blue. NNQL can also scale an image to the desired pixel dimensions when you add the suffix @widthxheight, e.g., "image.png@20x30". This scales the image to be 20 pixels wide by 30 pixels high. The input to the network layer is a single time-step vector consisting of each raster line of the image.

Audio. Stimulators support loading of mono audio files; these are recognized by having the .WAV extension. NNQL passes the audio files through a Fast-Fourier Transform (FFT) to yield normalized spectrograms which are suitable for feeding into neural networks as currents. The input to the network layer is a list of vectors, one for each time point resulting from the FFT. Each element of the vector represents a frequency band. In the present version of NNQL, the FFT is parametrized so that the audio is transformed into 50 frequency band bins, and 100 time-steps. Only the first second (1 s) of audio is processed from each sample. Future versions of NNQL will allow custom specification of these parameters.

PARAMETERS

`FROM`	Name of the file from which to read the currents to stimulate the nucleus with.
`POFFNOISE`	Probability with which to deactivate any particular element in a time-step vector (set the current to 0 for) an input.

5.3.4 RECORD FROM Syntax

RECORD FROM ″$nucleusName$″ INTO ″$tracename$″
RECORDER ″$tracename$″ SAVETO ″$filename$″
RECORD FROM ″LAYER1″ INTO ″MyTraces1″
RECORDER ″MyTraces1″ SAVETO ″traces1.txt″

RECORD FROM sets up a Recorder, which samples current from the neurons in the associated Nucleus. T

RECORDER x SAVETO y saves the traces in the recorder into a file.

5.3.5 LIST MODEL Syntax

The LIST command outputs textual and graphical descriptions of models built in NNQL.

```
LIST MODEL
```

PURPOSE

This command lists all the components in a model, the statements that created it, along with applicable references to the literature.

PARAMETERS

None.

5.3.6 LIST DOTDIAGRAM Syntax

```
LIST DOTDIAGRAM(threshold, scale) SHOW;
LIST DOTDIAGRAM(1,0.1) SHOW;
```

Outputs a diagram of the model as a directed graph, see Figure 2 on page 40. The output is in DOT format, and can be plotted by applications such as GraphViz.

Graph Edges. Positive weights are indicated by red lines with an inverted-triangle line termination and negative weights are indicated by blue lines with a T termination. Only weights whose absolute value exceeds the threshold specified by the *threshold* parameter are displayed. The thickness of the lines indicates the numerical value of the weight, scaled by the *scale* parameter.

Labels. Each unit is labeled with its unique ID number (starting by default from 1001) as well as any labels that have been applied to it (see labeled keyword in INSERT NEURON syntax, section 5.1.2 on page 19. All units are surrounded by a box representing their nucleus. The nucleus is also labeled by its name.

PARAMETERS

`SHOW` Uses GraphViz (if installed) to visually display the graph.

NOTES

Future versions of this command will allow you to specify which nuclei to plot.

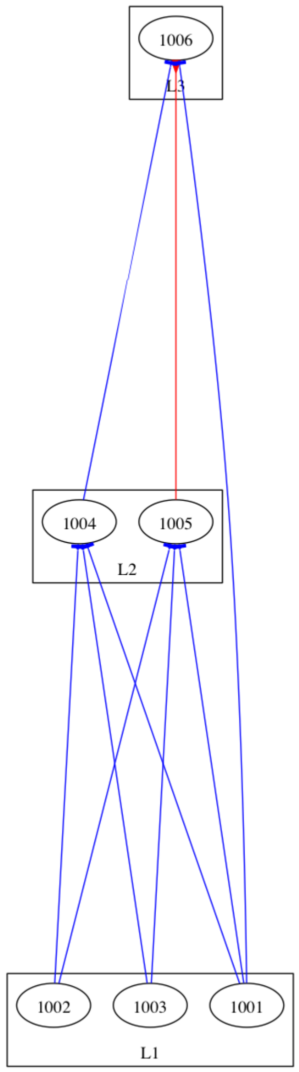

Figure 2: **Example DOT diagram of a three-layer network.** This is an example network diagram generated by NNQL. The numbers on each neuron reflect the unique ID number assigned to the neuron in the current simulation

5.4 Plotting

A Recorder exposes functions that allow plotting. In particular, currents for the recorded neurons can be plotted using the makePlot command, or a single raster plot of all the neurons can be generated by using makePopRaster[5].

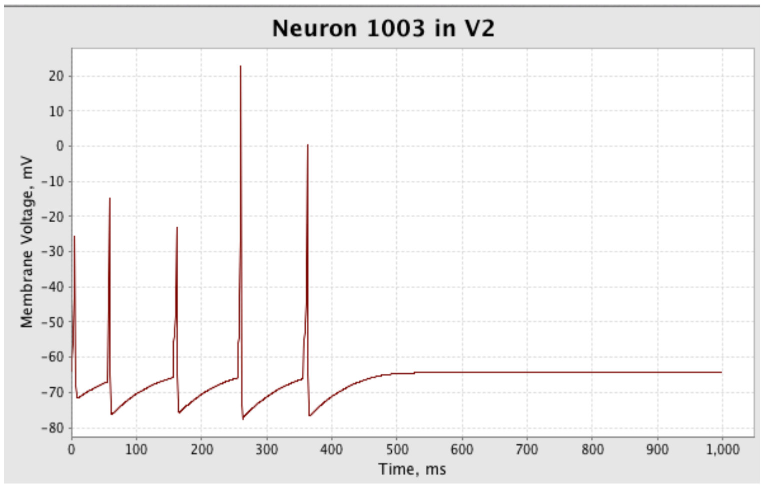

Figure 3: **Example voltage plot.** This is an example voltage plot generated by NNQL by calling makePlot on a RECORDER (e.g., RECENT_RECORDER or ALL_RECORDERS).

[5]In the future, a makeTrialEventRaster will be implemented.

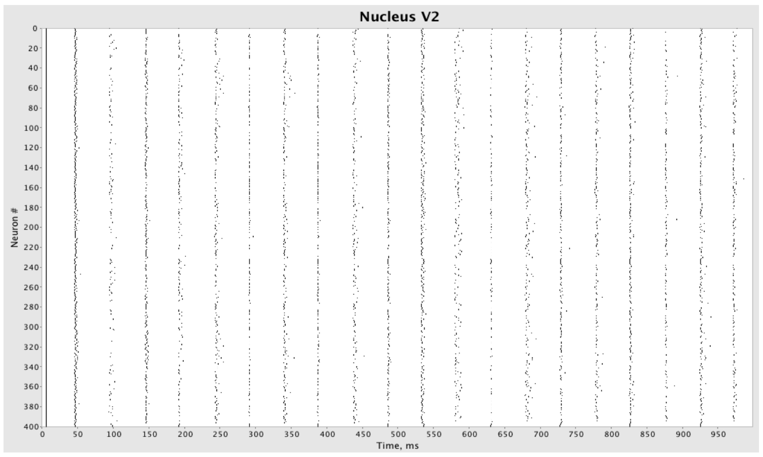

Figure 4: **Example raster plot.** This is an example population raster generated by NNQL by calling make-PopRaster on a RECORDER (e.g., RECENT_RECORDER or ALL_RECORDERS).

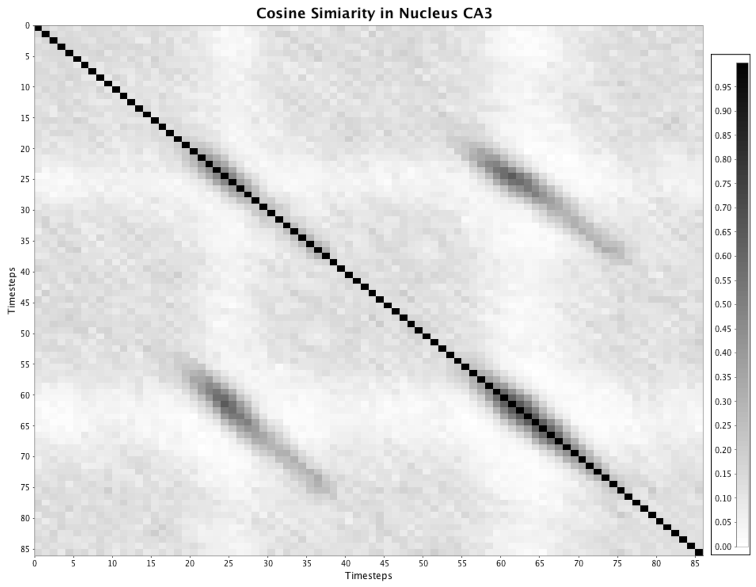

Figure 5: **Example similarity plot.** This is an example cosine similarity plot generated by NNQL by calling makeSelfSimilarity on a RECORDER (e.g., RECENT_RECORDER or ALL_RECORDERS). The repeated fuzzy pattern in the upper-right corner indicates successful sequence recall.

5.4.1 RECORDER Syntax

```
RECORDER “recorder1” makePlot
RECORDER “recorder2” listSpikeCounts
RECORDER “recorderX” makePopRaster
RECORDER “recorderName” makeSelfSimilarity

ALL_RECORDERS makePlot, etc...
RECENT_RECORDER makePlot, etc...
```

NOTE: Instead of addressing a RECORDER by name, one can also access the most recently created recorder (RECENT_RECORDER), or collection of all recorders (ALL_RECORDERS).

PURPOSE

Recorders allow the user to store firing patterns from a NUCLEUS into memory for later analysis, plotting, or saving into a file.

COMMANDS

`makePlot`	Makes plots of the voltages of all the neurons associated with the Recorder(s).
`listSpikeCounts`	Displays the spike counts of all spiking neurons associated with the Recorder(s).
`makePopRaster`	Generates a population raster graph of all the neurons associated with the Recorder(s).
`makeSelfSimilarity`	Displays a similarity plot of all recorded activity against itself. Ideal if the stimulator has recorded both training and testing activity.

5.5 Profiling

While running models it can often be of interest to quantify how long particular parts of simulations take. It can also be of interest to know the degree to which simulation time increases as, for example, network connectivity is increased.

```
PROFILER ENABLE
```

The `enable` command enables profiling for the current simulation.

```
PROFILER START "identifierString"
PROFILER STOP "identifierString"
```

The `start` and `stop` commands should surround the commands to be profiled. It is possible to next `start` and `stop` commands.

```
PROFILER REPORT
```

Once the simulation is completed, the `report` command will output a report of the total amount of time spent between each respective identifierString.

6 Stimulus Generation

Stimulation can be loaded from text files. NNQL also provides some basic stimulus generation capabilities built in to the language, which are described in this section.

6.1 SEQ_GENERATOR Syntax (Sequence generator)

```
SEQ_GENERATOR N(1024) SIZE(10) STUTTER(1) SHIFT(3) SEQLEN(10) FILENAME(”sequence1.txt”) RUN
SEQ_GENERATOR SIZE(10) CURRENT(20) STUTTER(2) SHIFT(3) SEQLEN(20) FILENAME(”sequence2.txt”) RUN
```

The Sequence generator will activate groups of units in sequence over time. The number of time-steps is equal to SEQLEN x STUTTER.

PARAMETERS

`N`	Total number of input currents to stimulate (should not exceed the number of neurons in the target nucleus).
`SIZE`	Maximum number of input currents per pattern time-step (i.e., 10 neurons at a time).
`STUTTER`	Number of time steps per pattern (i.e., number of time-steps to keep stimulating the same neurons).
`SHIFT`	Amount of shift after each stutter is completed. If STUTTER = SIZE, then orthogonal (non-overlapping) groups of neurons are stimulated.
`SEQLEN`	How many patterns.
`FILENAME`	Name of file to save to.
`RUN`	Command to execute the sequence generator.

7 Future Directions

This section discusses enhancements and features that could be developed in future versions of NNQL.

7.1 Conductance-based modeling

Here we specify the effects of specific neurotransmitters like glutamate, GABA, and dopamine, on particular types of cells. Additional syntax will be introduced, including:

- PROJECT FROM x TO y RELEASING z
- EXPRESS x IN NUCLEUS y

Examples:

```
CREATE NUCLEI ("SNpc", "STRIATUM")
INSERT NEURON IZ("phasic_spiking") INTO "SNpc" QUANTITY 100 LABELED "DAergic"
INSERT NEURON mediumSpiny(I,V) INTO "STRIATUM" QUANTITY 10000 LABELED "substanceP"
INSERT NEURON mediumSpiny(I,V) INTO "STRIATUM" QUANTITY 10000 LABELED "enkephalin"
EXPRESS "D1" IN "STRIATUM substanceP"
EXPRESS "D2" IN "STRIATUM enkephalin"
PROJECT FULLY FROM "SNpc DAergic" TO "STRIATUM" RELEASING ("DOPAMINE", "GLUTAMATE")
```

References

Bi, G. Q. and Poo, M. M. (1998). Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. *J Neurosci*, 18(24):10464–72.

Izhikevich, E. M. (2004). Which model to use for cortical spiking neurons? *IEEE Trans Neural Netw*, 15(5):1063–70.

Izhikevich, E. M. (2007). Solving the distal reward problem through linkage of stdp and dopamine signaling. *Cereb Cortex*, 17(10):2443–52.

Levy, W. B. (1996). A sequence predicting ca3 is a flexible associator that learns and uses context to solve hippocampal-like tasks. *Hippocampus*, 6(6):579–90.

Levy, W. B. and Steward, O. (1983). Temporal contiguity requirements for long-term associative potentiation/depression in the hippocampus. *Neuroscience*, 8(4):791–7.

Markram, H., Lübke, J., Frotscher, M., and Sakmann, B. (1997). Regulation of synaptic efficacy by coincidence of postsynaptic aps and epsps. *Science*, 275(5297):213–5.

Mitman, K., Laurent, P., and Levy, W. (2003). Defining time in a minimal hippocampal ca3 model by matching time-span of associative synaptic modification and input pattern duration. In *Neural Networks, 2003. Proceedings of the International Joint Conference on*, volume 3, pages 1631–1636. IEEE.