

N2S3: User Guide

June 30, 2015

1 Getting started

1.1 Requirements

1.1.1 Java virtual Machine

First of all, you will need a Java Virtual Machine in order to run N2S3.
See : <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
(JDK and JRE)

1.1.2 Simple Build Tool

In order to manage easily the requiert packages, you have to install sbt :
<http://www.scala-sbt.org/release/tutorial/Setup.html>

1.1.3 Integrated Development Environment

Choose your own favorite IDE for scala programming, for example : <http://scala-ide.org/>.

You can also add the plugin into already installed eclipse from the market-place.

1.1.4 Subversion

In order to checkout the project repository, you must have a subversion client installed on your computer : <https://subversion.apache.org/> or `sudo apt-get install svn`.

The command to checkout is : `svn checkout http://forge.lifl.fr/Emeraude/svn/main/Software/N2S3v2`

1.2 Starting

After you pulled the project from the svn repository, you must build a sbt project for your eclipse. Open a terminal (or a cmd for window user), place you in the directory of N2S3, and type : `sbt eclipse`. It will download all the packages that are required for running the simulator.

You just have to add N2S3 project to your workspace in order to run it. You can create a new Scala Project with the same name if N2S3 is in your workspace, or you can use file -> import -> existing project into workspace

1.3 Example

Some example are provided, in experiment package. Such as VerticalMove(Dsl).scala, that will build a simple network in order to recognize if a pixel are going up or down.

You can see a description of the Domain Specific Language at the section 3

2 General Architecture

2.1 Packages

N2S3 contains several packages :

- core contains main common feature essential for simulator working
- models should contains different implemented models of neural network. this package extends the core package
- features contains additionnal functionality of the network (logging, visualisation)
- experiments contains user trying. this package use models and features package

2.2 Available features

3 Domain Specific Language

Experiment creation is done with command "Experiment.create(network : Network)". After this one, several section are possible. Here is an example :

```
Experiment.create(new Network)
  .configuration{ config => } // Configuration section
  .topology{ topology => } // Topology section
  .stage { stage => } // Stage section
```

Order of section are not important, but number of them is : It can be only one configuration and topology section, but 0 or more stage section.

For run the experiment, you simply need to call command execute() after all your section.

3.1 Configuration section

Section *configuration* allow to set global parameter of the experiment. here is the list of the available command :

- name : set name of the experimentation
- synchronizer : set the synchronizer manager type used in the simulation

Example of a *configuration* section :

```
Experiment.create(new Network)
  .configuration{ config =>
    config name "My Experiment"
    config synchronizer new MonoSynchronizer
    config neuron NeuronThreshold(5 Volt)
  }
```

3.2 Topology section

This section used to construct the network topology. A network is composed of input and layer. Each of them is identified by a string.

For add an input, the line command is "topology input("Input name") ofType ..." for the moment, there are three types of input : SimpleInput, MnistFile and AERFile

- SimpleInput which can send pre-synaptic and post-synaptic spike to an unique neuron
- MnistFile which can read in the mnist database file and send event on neurons of a layer
- MnistFile which can read an AER file and send event on neurons of a layer

Example of the inputs

```
Experiment.create(new Network)
  .topology{ topology =>
    // Simple Input which can be connected to only one neuron
    topology input "Input 1" ofType SimpleInput() connectedTo
      Neuron("Input Layer", 0)
    // Mnist
    topology input "Input 2" ofType MnistFile()
      from("images.idx3-ubyte", "labels.idx1-ubyte") connectedTo
      Layer("Input Layer")
    // Address Event Representation
    topology input "Input 3" ofType AERFile() from "data/file.aer"
      ofSize(width = 128, height=128) withAddress
      RetinaWithSignedSpikeInitialize.apply connectedTo
      Layer("Input Layer")
  }
```

It is possible to set parameters of the model depending layers. But available property depend of which are implemented in used model.

Here is an example of topology section :

```
.topology { net =>
    net input("Input Tmpdiff128") ofType AERFile() from
        "data/aerdata/freeway.mat.dat" ofSize(width = 128,
        height=128) withAddress
        RetinaWithSignedSpikeInitialize.apply connectedTo
        Layer("Hidden Layer")

    net layer("Hidden Layer") has(neuron = 60, target =
        "Output Layer")
    net onLayer("Hidden Layer") set NeuronThreshold(500 Volt)
net onLayer("Hidden Layer") set NeuronRefractory(300 MilliSecond)
net onLayer("Hidden Layer") set NeuronInhibitRefractory(50
    MilliSecond)

net onLayer("Hidden Layer") set SynapseLTP(12 MilliSecond)
net onLayer("Hidden Layer") set SynapseLeak(450 MilliSecond)

net layer("Output Layer") has(neuron = 10)
net onLayer("Output Layer") set NeuronThreshold(1.5 Volt)
net onLayer("Output Layer") set NeuronRefractory(250 MilliSecond)
net onLayer("Output Layer") set NeuronInhibitRefractory(100
    MilliSecond)

net onLayer("Output Layer") set SynapseLTP(300 MilliSecond)
net onLayer("Output Layer") set SynapseLeak(300 MilliSecond)
}
```

3.3 Result section

This section is optionnal and it use to add a logger on result of the network. Here, an example of usage for vertical move example :

```
.result { result =>
    result init Array("Up", "Down")
}
```

You must manage it during the experimentation in two steps : Learning and Testing

3.4 Stage sections

An experiment can have several stage. The different stage will be executed in the order of declaration and in synchronous way (next stage will wait end of simulation of the previous stage before start).

Stage allow to start different type of plot and log, to generate input and some other stuff.

3.4.1 Input

In case of a manual input (without file for it), you must explicitly send them :

```
stage onInput ("Pixel 1 +") providePostSynapticSpike (time = 2*i
  MilliSecond, weight = 2 Volt)
stage onInput ("Pixel 2 -") providePostSynapticSpike (time = 2*i
  MilliSecond, weight = 2 Volt)
```

3.4.2 Logging

You have some logger available for experimentations. Each of these logger are used in the same way :

```
stage {plot|log} ({name of the logger}) follow {Logging you want}
  (betweenLayer|layer) ({names of layers}) (unit UnitTime)
```

Neurons Fire:

```
stage plot ("Neuron Fire Of OutputLayer") follow
  NeuronFirePlot layer ("Output Layer")
```

It will build a plot of the layer named "Output Layer" on the neuron which are fired. (NeuronFireLog to create a log file in output/log/neuronfire.log)

SynapsesWeight:

```
stage plot ("Weight of Synapses of OutputLayer") follow
  SynapseWeightPlot betweenLayer("Input Layer", "Output
  Layer")
```

As the same way, but for the weights of synapses which link the Input Layer and Output Layer. There is other logger available :

- *SynapseWeightRepartPlot* : it will build a graph with all weights of synapses at the end of the simulation. (no need to precise which layer you want, it will take the whole synapses of the network)
- *SynapseWeightAtEndPlot* : it will build a similar graph as *SynapseWeightPlot* but at the end of simulation (not during it). Be careful, you must use a new stage at the end like this :

```
.stage { stage =>
  stage onPlot("Synapse weight of output layer") run(0
    MilliSecond, 1500 MilliSecond)
}
```

Parameters of run are the border of the graph you want (from 0 MilliSecond to 1500 MilliSecond in this example). The name must match with the declared *SynapseWeightAtEndPlot*.

Follow a simulation You can add a display of neurons which are firing using :

```
stage plot ("I/O") follow ActivitiesOfLayers topology (Array((20,
  20, "Input Layer"), (2, 5, "Output Layer")))
```

topology keyword is waiting about an array of (Int,Int,String). The string must match with layers declared in topology stage. You must provide an (Int,Int,String) for each of them which the couple of Int represent the dimension of the Rect (number of neurons) you want to display (must match with the number of the layer ($x*y=nbNeuronPerLayer$

Names of layers must match with layers declared in the topology section.

About the unit of time, you can use *Second*, *MilliSecond* and *MicroSecond*. By default, all graph are in MilliSecond, but you can change this unit like this :

```
stage plot ("Weight of Synapses of OutputLayer") follow
  SynapseWeightPlot betweenLayer("Input Layer", "Output Layer") unit
  Second
```

It will look for the same thing, but with Second as unit for the X axis.

3.4.3 Result

For each event, you must send to the logger result which one you give to the network :

```
stage onResult() event("Up", 2*i MilliSecond)
```

The logger will count event and after the stage Learning, it will attribute a label (which are in the Array given at the construction) at one neuron. After the learning stage, you can pass in test stage like this :

```
stage onResult() test()
```

At the end of the test stage, you have a report in the console.

4 Write your own experiment in scala

4.1 Introduction

This section is only for scala user, which want to write their own experiment in scala.

4.2 Basics

First of all, we will see how to create the network.

- 1: You need to build the network, as an object :

```
val net = new Network()
```

- 2: Then, you must describe the topology with a Seq[Int] :

```
net.nbNeuronPerLayer = Seq(4, 2)
```

- 3: Finally, you set the type of synchronization you want :

```
net.initiateNetwork(new MonoSynchronizer())
```

After that, you can add a specially layer manager. By default, the layer-Manager do nothing. You can add inibition on a layer with a Winner Take All manager as this :

```
net.setLayerManager(1, net.system.actorOf(Props[WinnerTakeAll]))
```

To Remove it, you just put a default LayerManager :

```
net.setLayerManager(1, net.system.actorOf(Props[LayerManager]))
```

4.3 Input

4.3.1 Input from file

N2S3 provide two input from file : AER file, and file of handwritten digit from Mnist database (see : <http://yann.lecun.com/exdb/mnist/>).

AER In order to use AER file, you must create the input generator like this :

```
val input = net.system.actorOf(Props(new InputJAERFile(net, filename,
    0, 128, 128, RetinaWithSignedSpikeInitialize.apply)), name =
    "Input")
```

It will build an actor InputJAERFile.

- net : the network of the experimentation
- filename : the path to the input file
- 0 : index of the target layer of the input
- 128,128 : the size of the input layer
- RetinaWithSignedSpikeInitialize.apply : the address mode

There is 3 address mode :

- CochleaInitialize : for audio input
- RetinaInitialize : for DVS input, without difference between positive and negative spike
- RetinaWithSignedSpikeInitialize : same as before, but with the difference

After that, you need to Initialize the input with :

```
Await.result(ask(input, Initialize()), timeout.duration)
```

Mnist In the same way, you can read file from mnist database with this :

```
val input = net.system.actorOf(Props(new InputMnist(net,
    "data/train-images.idx3-ubyte", "data/train-labels.idx1-ubyte", 1,
    0)), name = "Input")
```

- net : the network of the experimentation
- string,string : these two string are the path for file to be read, one for inputs themselves(array of spikes), and the other for the label for each input
- 1 : the size of chunk of the file you want to read (more it is huge, more the reading will be long, but less often)
- 0 : index of the target layer of the input
- boolean : a last parameters is available to switch between two mode

Mode of input :

- false(default) : It will transform the intensity of pixels into frequency (more it is white, more we send spike)
- true : It will transform the weight of spike in function of the intensity of the pixel.

Do not forget to Initialize (as for AER input) your actor.

4.3.2 Manual Input

As for Input from file, you must create an actor, and then ask to connect it to the right neuron, because it won't create a layer of input neuron as "automatic" input. See the example :

```
val pixel1Positive = net.system.actorOf(Props(new
    SimpleInput(net.synchronizer.getSynchronizerOfInput()), name =
    "Pixel_1_Positive")
```

First we create the actor, and then we connect it to the first neuron of the first layer :

```
Await.result(ask(pixel1Positive,
    SimpleInputConnectWith(net.neuronsActorRef(0)(0))),
    timeout.duration)
```

4.4 Logging/Monitoring

Now, let see how to monitor your experiment. All of logger are actor. Graph can use different unit for the time : *MicroSecond*, *MilliSecond* and *Second*

4.4.1 Plot

You can create a lot of logger which will build plot during, or after the simulation.

First, see the list of online plot :

- `NeuronsPotentialsLogGraph` : it will track the update of the potential of neurons.
- `NeuronsFireLogGraph` : it will track the neurons which are firing
- `SynapsesLogGraph` : it will track the update of the weights of synapses.

Those Plots have the same construction :

```
NeuronsFireLogGraph(step: Int = 50, unit : TimeUnitType = MilliSecond)
```

By default, the step between two point is 50, and the unit of time is MilliSecond. You can change them as the construction, or let by default:

```
val logger = net.system.actorOf(Props(new NeuronsPotentialsLogGraph()))
```

Here, without any parameters, it will take the default value.

```
val logger = net.system.actorOf(Props(new SynapsesLogGraph(10, Second)))
```

Here, I want a point each 10 Second on my plot.

After you build your actor, you must Subscribe him to want it have to look at : Which neurons/synapses of which layers, and which event? Here, an example of a way to Subscribe your logger :

```
for (i <- 0 until net.nbNeuronPerLayer(1))  
  net.neuronsActorRef(1)(i) ! Subscribe(NeuronFireEvent(), logger)
```

It will Subscribe logger to the `NeuronFireEvent()` of each neurons of the layer 1.

`nbNeuronPerLayer` and `neuronsActorRef` are members of the class `Network`. `nbNeuronPerLayer` is an array, which contains the numbers of neuron for the layer at the index `i` (`net.NeuronPerLayer(1)` returns the number of neurons in the layer 1). In the same way, `neuronsActorRef` contains `ActorRef` for each neurons in each layers (`net.neuronsActorRef(i)(j)` returns the `ActorRef` of the `j` neuron in the `i` layer)

There is 3 event available :

- `NeuronFireEvent()` : When a neuron Fire.
- `NeuronPotential()` : When a potential of neuron change.
- `SynapseWeightEvent(id)` : When the id synapse of neuron change.

Now, see the list offline plot :

- `SynapsesWeightRepartition(sync: ActorRef)` : This actor will build a plot at the end of the simulation of the whole last weight of synapses. You must give him the synchronizer at the construction (Does not work with mutiple synchronizer). You should Subscribe it to all synapses of the network to have a great plot at the end :

```

for {
  k <- 0 until net.nbNeuronPerLayer.size - 1
  i <- 0 until net.nbNeuronPerLayer(k)
  j <- 0 until net.nbNeuronPerLayer(k + 1)
} yield {
  Await.result(ask(net.neuronsActorRef(k)(i),
    GetConnectionId(net.neuronsActorRef(k + 1)(j))),
    timeout.duration) match {
    case ReturnConnectionId(list) => list.foreach { id =>
      net.neuronsActorRef(k + 1)(j) !
      Subscribe(SynapseWeightEvent(id), loggerReport) }
  }
}

```

This code will Subscribe the logger to every synapses in the network.

- SynapsesGraph : This is graph is the same as the online(with same args : *SynapsesLogGraph*) but you must send him a message to run :

```

loggerAtEnd ! Run(21000 Millisecond, 26000 Millisecond)
loggerAtEnd ! Run(0 Millisecond, 10000 Millisecond)

```

Here, it will display 2 plot, between the two borders you gave.

Finally, we will see the *NeuronInputOutputGraph* which it needs more explanation.

First of all, it is a online plot. It will display a black square for each neuron in the network, and when one of them fire, it will turn to white. Let's see the args of the construction

```

class NeuronInputOutputGraph(net: Network, topo :
  Array[(Int,Int,String)], input : ActorRef = null)

```

- net : Network of the simulation
- topo : An array which is contains the description of the network as you want to display it with a name, for each layers.
- input : in case of use a Mnist input, you can give it to add it to the plot.

Here, see two example of construction :

```

val logger = net.system.actorOf(Props(new NeuronInputOutputGraph(net,
  Array((20,20,"N0") , (2,5,"N1")), input)))

```

```

val logger = net.system.actorOf(Props(new NeuronInputOutputGraph(net,
  Array((1,4,"N0") , (1,2,"N1")))))

```

You don't have to Subscribe it, it will do it itself to the whole neurons of the network.

4.4.2 Log

In the same way, you record the change in the network into file (output/log).

- `NeuronsFireLogText` : will record in `output/log/neuronfire.log` neurons which fires.
- `NeuronsPotentialLogText` : will record in `output/log/neuron.log` potentials of neurons which changes.
- `SynapsesLogText` : will record in `output/log/synapses.log` weights of synapses which changes.

Do not forget to Subscribe your logger at the event.

4.4.3 Result

There is a particular logger, which can test if a network learnt or not.

```
val loggerRes = net.system.actorOf(Props(new LogResult(output :  
    Array[ActorRef], label : Array[String])))
```

output is the layer of output of the network, and label is an array which is contains labels (or pattern) that the network will recognize.

There is two step in this logger : Learning and Testing. You need to send to it messages during the simulation :

- `loggerRes ! Synchronization(net.synchronizer.getSynchronizerOfInput())`

This message is used to know when to switch in the Testing stage.

- `loggerRes ! EventTime("Up", i * 2 Millisecond)`

This one, provide to the logger which event you are sending to the network.

- `net.synchronizer.waitComputation()`

In order to have a perfect synchronization, you will need to wait the computation(according to the Synchronization (1st message))

- `loggerRes ! TestKnowledge(net.synchronizer.getSynchronizerOfInput())`

In order to mark the begin of the Testing stage.

At the end of the simulation, you will a report in the console.

5 Create a new model

N2S3 allow you to create new neural network models. The main step is to inherit from `core.Network` and override abstract function :

- `createNeuron` which build specialized model neuron
- `sendInputEvent` which manage an input reception

If models need more feature, it will need redefine more behavior

- `XMLBuilder` need to specialize construction of the network

Each models can define it's own `Property` and event. Each property need to inherit from `Property` class need to be process in the `Neuron.processProperty` function for setting new values of properties.

Event need to be added via `addEvent` function. after that, each of them can be trigger with `triggerEvent` function. This one will send the message given on second parameter to all observer of this event.

A Example of Vertical Move in the DSL

```
object VerticalMoveDSL extends App {  
  
  type model = fr.cristal.emeraude.n2s3.models.qbg.Network  
  
  Experiment.create(new model)  
    .configuration { config =>  
      config name "Dsl exemple"  
      config synchronizer new MonoSynchronizer  
  
      config neuron NeuronThreshold(1 Volt)  
    }  
  .topology { net =>  
    net input ("Pixel 1 +") ofType SimpleInput() connectedTo  
      Neuron("Input Layer", 0)  
    net input ("Pixel 1 -") ofType SimpleInput() connectedTo  
      Neuron("Input Layer", 1)  
    net input ("Pixel 2 +") ofType SimpleInput() connectedTo  
      Neuron("Input Layer", 2)  
    net input ("Pixel 2 -") ofType SimpleInput() connectedTo  
      Neuron("Input Layer", 3)  
  
    net layer ("Input Layer") has (neuron = 4, target = "Output Layer")  
    net onLayer("Input Layer") set NeuronThreshold(1 Volt)  
  
    net layer ("Output Layer") has (neuron = 2)  
  }  
  .result { result =>  
    result init Array("Up", "Down")  
  }  
}
```

```

}
.stage { stage =>
  stage name "Learning"

  stage plot ("Synapse Repartition") follow SynapseWeightRepartPlot
  stage plot ("Weight of Synapses of OutputLayer") follow
    SynapseWeightPlot betweenLayer("Input Layer", "Output Layer")
    unit Second
  stage plot ("Synapse Weight") follow SynapseWeightAtEndPlot
    betweenLayer ("Input Layer", "Output Layer")
  stage log ("NeuronFire") follow NeuronFireLog layer ("Output Layer")

  stage onLayer("Output Layer") set WinnerTakeAllManager

  for(i <- 0 to 5000) {
    if (i % 2 == 0) {
      stage onResult() event("Up", 2*i MilliSecond)
      stage onInput ("Pixel 1 +") providePostSynapticSpike (time =
        2*i MilliSecond, weight = 2 Volt)
      stage onInput ("Pixel 2 -") providePostSynapticSpike (time =
        2*i MilliSecond, weight = 2 Volt)
    } else {
      stage onResult() event("Down", 2*i MilliSecond)
      stage onInput ("Pixel 1 -") providePostSynapticSpike (time =
        2*i MilliSecond, weight = 2 Volt)
      stage onInput ("Pixel 2 +") providePostSynapticSpike (time =
        2*i MilliSecond, weight = 2 Volt)
    }
  }
}

.stage { stage =>
  stage name "Testing"
  stage onResult() test()
  stage onLayer ("Output Layer") set NoManager

  val r = scala.util.Random

  for (i <- 0 to 5000) {
    if (r nextBoolean ) {
      stage onResult() event("Up", 4*i MilliSecond)
      stage onInput ("Pixel 1 +") providePostSynapticSpike (time =
        4*i MilliSecond, weight = 2 Volt)
      stage onInput ("Pixel 2 -") providePostSynapticSpike (time =
        4*i MilliSecond, weight = 2 Volt)
    } else {
      stage onResult() event("Down", 4*i MilliSecond)
      stage onInput ("Pixel 1 -") providePostSynapticSpike (time =
        4*i MilliSecond, weight = 2 Volt)
      stage onInput ("Pixel 2 +") providePostSynapticSpike (time =
        4*i MilliSecond, weight = 2 Volt)
    }
  }
}

```

```
.stage { stage =>
  stage onPlot("Synapse Weight") run(0 Millisecond,1500 Millisecond)
}
.execute
}
```

B Example of Vertical Move in Scala

```
object VerticalMove extends App {

  println("Create Network ...")

  val net = new Network()
  net.nbNeuronPerLayer = Seq(4, 2)
  net.initiateNetwork(new MonoSynchronizer())

  println("Create inputs ...")

  implicit val timeout = Timeout(10 seconds)
  import ExecutionContext.Implicits.global

  val pixel1Positive = net.system.actorOf(Props(new
    SimpleInput(net.synchronizer.getSynchronizerOfInput()), name =
    "Pixel_1_Positive"))
  Await.result(ask(pixel1Positive,
    SimpleInputConnectWith(net.neuronsActorRef(0)(0))),
    timeout.duration)

  val pixel1Negative = net.system.actorOf(Props(new
    SimpleInput(net.synchronizer.getSynchronizerOfInput()), name =
    "Pixel_1_Negative"))
  Await.result(ask(pixel1Negative,
    SimpleInputConnectWith(net.neuronsActorRef(0)(1))),
    timeout.duration)

  val pixel2Positive = net.system.actorOf(Props(new
    SimpleInput(net.synchronizer.getSynchronizerOfInput()), name =
    "Pixel_2_Positive"))
  Await.result(ask(pixel2Positive,
    SimpleInputConnectWith(net.neuronsActorRef(0)(2))),
    timeout.duration)

  val pixel2Negative = net.system.actorOf(Props(new
    SimpleInput(net.synchronizer.getSynchronizerOfInput()), name =
    "Pixel_2_Negative"))
  Await.result(ask(pixel2Negative,
    SimpleInputConnectWith(net.neuronsActorRef(0)(3))),
    timeout.duration)

  println("Configure network ...")

  //Inibition on the layer 1 (the output)
  net.setLayerManager(1, net.system.actorOf(Props[WinnerTakeAll]))

  //Create a logger Res
  val loggerRes = net.system.actorOf(Props(new
    LogResult(net.neuronsActorRef(net.nbNeuronPerLayer.size - 1),
    Array("Up", "Down"))))
```

```

//Subscribing the logger res to the output neurons
for (i <- 0 until net.neuronsActorRef(1).length)
  net.neuronsActorRef(1)(i) ! Subscribe(NeuronFireEvent(), loggerRes)

//Synchronization to change step (first learning)
loggerRes ! Synchronization(net.synchronizer.getSynchronizerOfInput())

//logger plot
val loggerAtEnd = net.system.actorOf(Props(new SynapsesGraph(5000)))
val logger = net.system.actorOf(Props(new SynapsesLogGraph))
val loggerRepart = net.system.actorOf(Props(new
  SynapsesWeightRepartition(net.synchronizer.getSynchronizerOfInput())))
//logger text
val loggerNeuron = net.system.actorOf(Props(new NeuronsFireLogText))

//Subscribing the LoggerRepart to the whole synapses in the network
for {
  k <- 0 until net.nbNeuronPerLayer.size - 1
  i <- 0 until net.nbNeuronPerLayer(k)
  j <- 0 until net.nbNeuronPerLayer(k + 1)
} yield {
  Await.result(ask(net.neuronsActorRef(k)(i),
    GetConnectionId(net.neuronsActorRef(k+1)(j))), timeout.duration)
  match {
    case ReturnConnectionId(list) => list.foreach {id =>
      net.neuronsActorRef(k+1)(j) !
        Subscribe(SynapseWeightEvent(id), loggerRepart)}
  }
}

//Subscribing the logger and logger at end to the 8 synapses of output
for (i <- 0 until 4) {
  net.neuronsActorRef(1)(0) ! Subscribe(SynapseWeightEvent(i), logger)
  net.neuronsActorRef(1)(0) ! Subscribe(SynapseWeightEvent(i),
    loggerAtEnd)
  net.neuronsActorRef(1)(1) ! Subscribe(SynapseWeightEvent(i), logger)
  net.neuronsActorRef(1)(1) ! Subscribe(SynapseWeightEvent(i),
    loggerAtEnd)
}

//Subscribing the loggerNeuron to the whole neuron in the network
for (i <- 0 until net.nbNeuronPerLayer.size-1)
  for (j <- 0 until net.nbNeuronPerLayer(i))
    net.neuronsActorRef(i)(j) ! Subscribe(NeuronFireEvent(),
      loggerNeuron)

println("Start simulation ...")

println("Learn ...")

val nbLearn = 1500

for (i <- 0 until nbLearn) {
  if (i % 2 == 0) {

```



```

    loggerRes ! EventTime("Up", i * 2 MilliSecond)
    net.synchronizer.getSynchronizerOfInput() !
        InputPostSynapticSpike(i * 2 MilliSecond, pixel2Negative, 1
            Volt)
    net.synchronizer.getSynchronizerOfInput() !
        InputPostSynapticSpike(i * 2 MilliSecond, pixel1Positive, 1
            Volt)
} else {
    loggerRes ! EventTime("Down", i * 2 MilliSecond)
    net.synchronizer.getSynchronizerOfInput() !
        InputPostSynapticSpike(i * 2 MilliSecond, pixel2Positive, 1.0f
            Volt)
    net.synchronizer.getSynchronizerOfInput() !
        InputPostSynapticSpike(i * 2 MilliSecond, pixel1Negative, 1.0f
            Volt)
}
}

net.synchronizer.waitComputation()

println("Test ...")

//Removing the WinnerTakeAll for test
net.setLayerManager(1, net.system.actorOf(Props[LayerManager]))

//We pass into test in the logger Res
loggerRes ! TestKnowledge(net.synchronizer.getSynchronizerOfInput())

val r = Random
val nbTest = 5000

//The event are sent randomly
for (i <- nbLearn until nbLearn + nbTest) {
    if (r.nextBoolean) {
        loggerRes ! EventTime("Up", 4 * i MilliSecond)
        net.synchronizer.getSynchronizerOfInput() !
            InputPostSynapticSpike(i * 4 MilliSecond, pixel2Negative, 1
                Volt)
        net.synchronizer.getSynchronizerOfInput() !
            InputPostSynapticSpike(i * 4 MilliSecond, pixel1Positive, 1
                Volt)
    } else {
        loggerRes ! EventTime("Down", i * 4 MilliSecond)
        net.synchronizer.getSynchronizerOfInput() !
            InputPostSynapticSpike(i * 4 MilliSecond, pixel2Positive, 1
                Volt)
        net.synchronizer.getSynchronizerOfInput() !
            InputPostSynapticSpike(i * 4 MilliSecond, pixel1Negative, 1
                Volt)
    }
}

net.synchronizer.waitComputation()

```

```
//Running the logger at the end in the specified window of time
loggerAtEnd ! Run(0 Millisecond, 1500 Millisecond)

println("End simulation...")
}
```