Dynamic discovery of static opinions

Andrè Casalis – Masters Degree in Economics
University of Turin – Alma universitas taurinensis

Index

1 Introduction

> *τὸ δὲ κινδυνεύει, ὦ ἄνδρες, τῷ ὄντι ὁ θεὸς σοφὸς εἶναι, καὶ ἐν τῷ χρησμῷ τούτῳ τοῦτο λέγειν, ὅτι ἡ ἀνθρωπίνη σοφία ὀλίγου τινὸς ἀξία ἐστὶν καὶ οὐδενός.* ***(“but the fact is, men of Athens, that only the god is really wise and by his answer he intend to show that the wisdom of men is in truth worth little or no value”) -Socrates in Plato, Apology***

1.1 General understanding of the idea

In the everyday life, every one of us receive information about the other people being in touch with and, alike, share personal details. All this amount of information is then used to make ideas and opinions about the world surrounding us –and about the others, as well.

The focus of this experiment is the aforementioned process of learning and making opinions, but on a more general degree, dealing with the aggregate level and the formation of a common knowledge by the individuals. This requires the definition of the problem on two different layers: a *micro*-phase when the individuals gather information about the world surrounding them, and a *macro*-phase, when the stock of information is used to synthetize opinions and shape beliefs. Each one will be carefully explained later.

In our work, we narrowed down the field of study to the formation of broad ideas about other people; thus, each individual, getting in touch with others, discover something that the collective intelligence (a being that is more than the mere sum of the individuals) will use to determine the *sort* of all the individuals populating the world.

1.2 The tools

The main tool used to realize this experiment is without doubt NetLogo[1], a multi-agent programmable environment, along with R[2], a very powerful software environment for statistical computing and graphics. Both of them are free, available for multiple platforms and easy to install.

Peculiarly, the model[3] itself has being written in NetLogo, while R is used via Rserve, a TCP/IP server which allows other programs to use facilities of R from various languages, in order to implement a Neural Network acting as the collective intelligence processing the raw data.

There should be no need to point out that for a full reproducibility of the experiment anyone should install and configure these software; anyway, a restricted version of the model, not empowered by R, was also realized and it is available as an online applet[4] as well as in the shape of a downloadable model[5] (although

[1] Wilensky, U. 1999. NetLogo. <http://ccl.northwestern.edu/netlogo/>. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, IL.
[2] R Core Team (2013). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. <http://www.R-project.org/>.
[3] May be downloaded from <web.econ.unito.it/terna/tesine/casalis/dynamic_discovery_of_static_opinions.nlogo> and seen working in a demonstrative video: <http://youtu.be/klxF9pdna40>.
[4] Available at <web.econ.unito.it/terna/tesine/casalis/index.htm>.
[5] The restricted version may be found at <web.econ.unito.it/terna/tesine/casalis/dynamic_discovery_of_static_opinions.nlogo>

lacking some of the features). You may want to refer to the paragraph 1.4.8 for the technical specificities of the online version.

1.3 The model

In our world, the agents are divided in two breeds (*Reds* and *Greens*), both having various opinions on a wide range of topics, along with some personal features. All these ideas and characters are summarized in a list of ten (by default, but easily modifiable) binary values, of which the first X[6] are considered "typical" and breed related, thus are equal for all the members of a breed (and different from those of the other breed).

Roaming around in the world, each agent learn something from whoever gets in touch with, enriching an information matrix that will be used as input for the training of the neural network. Once trained, the NN will try to determine the breed of each agent starting from their list of characteristics.

1.4 Technical side

1.4.1 Agents and memory – the memory matrices

We said that, when an agent learns something, it stores somewhere the new piece of information and that this will be used later by the neural network. We were speaking of the memory matrices, the core of the information storage system.

Each breed possesses two memory matrices, one for each breed the information are referring to. Although this structure multiplies the number of variables used, it is useful to maintain a certain degree of integrity in our data structure, with a 1:1 relation between the observer-observed breed combination for each matrix.

The matrices are created overabundant, with a large number of empty rows (full of the meaningless value *-1*, in fact) filled step by step by the agents. This method allows using huge matrices with almost no slowdown of the model, getting rid of any Java garbage collection handling problem.

The dimension of the matrix is set to be 20 times the number of agents rows, with the total number of properties plus one (the id of the agent having made that contribution) columns. The number of used rows is tracked and, when the maximum dimension is reached the matrix is all reset to *-1*.

All the matrix structure and features are made possible by the use of the Matrix[7] extension for NetLogo, a tool providing useful primitives to handle and manage matrix data structures inside NetLogo. This option was preferred to the (faster) "list of lists" structure due to the ability to extract a column of the matrix as a list and so being able to look up an agent id and, consequently, to easily find its last contribution.

1.4.2 Meeting others and updating memory – the *micro*-phase

Meet other agents and update the central memory are separated events. During the *meeting phase*, each agent check for the presence of other agents in his neighborhood (currently, on the same patch) and select what property it wants to share with them (random selection of the index of the property list); finally, it creates a directed link to the other agents. The link delivers information about the breed (the color of the link) and about the index and the value itself of the property (two attributes of the link).

[6] The user may select how many typical features are there for each breed.

[7] The Matrix extension for NetLogo was originally written by Forrest Stonedahl, with significant contributions from Charles Staelin (in particular, the forecast & regression primitives).

In the *update phase*, each agent checks if he has incoming links. If it is so, they retrieve the information and proceed to store them in the common memory. *Greens* and *Reds* behave the same way: each agent refers to the last row he inserted in the memory matrix and check the information already present in the same position of the new one. Now, three cases may arise:

a) That information is missing: that position is never being updated. The agent just fills it in.
b) That information is present and equal: that specific position was already updated in the past and with the same value of the new information; there is nothing to do, so the agent calls it a day and move on.
c) That information is present and different: that position is already filled, but with a different value; the agent create a copy of the entire row and update only the new information. This will become the new "last row" to refer to.

At the end of the process, the link is obviously asked to *die*.

1.4.3 Collective intelligences at work – the *macro*-phase

So, we have inspected the structure of the memory and discussed how the agents feed it during their wandering around the world. What happens next? The neural network is instructed on breed basis, so that it represents the ability of the given breed to assess whether an agent is either *Green* or *Red,* making use of all the data gathered by the agents of that breed. This operation gives birth to two sets of weights, one for each breed, used by the net to forecast an output (Green or Red) given an input (the set of properties of one specific agent).

At the end of the mapping process, each agent will possess a color of its own, a color as seen by the *Greens* and one as seen by the *Reds* (both attributes of the agent); further detail on the number of correct predictions are provided to the user –see section 3.1.5.

1.4.4 Weights reuse – advantage or liability?

The core tool of a neural network is, no doubt about this, the set of weights used to process the signal between the input and the hidden layer and then between the hidden layer and the output. Of course, the process of learning to accomplish a given task is no more than a tuning process of the weights, aiming to minimize the error.

The optimization logic is, in fact, quite simple: a first set of weights is chosen randomly and then a convergence is sought[8]. What happens, then, if the first set of weights is not random anymore, but the result of a previous optimization problem? The ability to do that is provided by the persistence of the variables in R workspace when accessed by another application. Rserve just keeps in memory all the variables created, weights set included, giving the possibility to use iteratively the previous optimal set to start a new optimization. The results of this choice are quite interesting and available in section 3.4.

[8] A lot of books have been written on this topic, so there would be no need to point out one or another. Anyway, if you need a clear and enjoyable explanation on the essence of a neural network, please refer to Terna, P. (2010): *Artificial Neural Networks (ANNs) Basics,* <http://eco83.econ.unito.it/terna/simoec13/cmap/annBasics.pdf>.
If you want something more detailed –and far more difficult- you may be interested in Hastie, T.; Tibshirani, R.; Friedman, J. (2009): *The Elements of Statistical Learning - Data Mining, Inference, and Prediction*, Stanford, California, Springer series in statistics; available here <http://www-stat.stanford.edu/~tibs/ElemStatLearn/>.

1.4.5 Embedding R code in NetLogo

We said that we use the power of R to implement and tune the neural network (i.e. computing the optimal value of the weights of a nonlinear statistical model) and that the communication is made possible by Rserve. How do we talk to Rserve, though?

All the work is done by an extension of NetLogo named Rserve[9] providing some primitives capable of create variables in R from NetLogo, as well as to execute R code stored in a string variable. Thus, the creation of R scripts is possible editing, assembling and creating strings of commands and storing them in NetLogo variables. The only limitations are the obligation to execute complex controls in one single shot (an *if* control has to run all together in the same instruction) and the not-so-flexible strings handling of NetLogo compared to other environments.

1.4.6 Segregation – stable and unstable outcomes

Two very simple segregation procedures can be activated once the mapping has been done. The first urges the agents to seek other agents that their breed sees as equal, while the second is the opposite, forcing the agents to flee from agents belonging to a different breed.

Although, when the mapping is flawless, a stable situation is almost ever reached -a situation comparable to that of similar works in field of segregation- when the network shows labeling errors a stable equilibrium may not be reached; a green agent may be seen as red by all the other *Greens* and thus avoided. This may lead to him becoming lonely or to a flee-catch never ending situation. To avoid infinite loops, the maximum amount of tries to reach a satisfying situation is capped to 1000.

1.4.7 Randomness – when things just go random

With the information set and the position of each agent, along with their movements and their probability to communicate to each other all entrusted to the Entropy of the Universe, the random internal generators rise to the role of the unrecognized stars. To mend this, we made clear what seed the NetLogo random number generator is using, giving the user the ability of modify it or choosing under what conditions a new one has to be selected. This is not the whole story, though: even R, in selecting the initial weights for the Neural Network, uses a random generator, on which the user has no control on, due to the far greater complexity of the instruments and of the technical problems in the alignment of the two systems.

1.4.8 Online version – when R is not available

Being the model heavily integrated with R, it should be no surprise that the online version, available in an applet, works differently: all the neural network part is unused and, as a poor substitution, the user can set the probability with which each breed will assess a given color. This is established in the setting up of the model, in fact making completely useless all the information gathering process but allowing to evenly use the segregation procedures.

[9] Jan C. Thiele, Department Ecoinformatics, Biometrics and Forest Growth; University of Goettingen

2 Essentials of code

> ***Entium varietates non temere esse minuendas.***
> ***("The variety of entities should not be rashly diminished")***
> ***-Emmanuel Kant*, Critique of Pure Reason**

If you want a complete, exhaustive comment of the whole code, this is not what you are looking for. The intention of this section is to give a little insight on the most important parts of code from the point of view of the experiment meaning: the simple and plain *if-else* structure controlling whether something should be memorized or not is explained, while the very complicated, multiple lines, piece of code resetting the memory matrices is not. For any further need you may have, try reading the code itself, which comes with a very detailed comment endowment.

2.1 NetLogo code

2.1.1 Random seed handling

```
to startup
  set rnd-seed new-seed
end
```

Initialize a global variable with proper number usable as the random seed (*generated by the primitive new-seed*)

```
to setup
  ca
  if (new-seed-on-setup?) [ set rnd-seed new-seed ]
  if (abs rnd-seed) > 2147483648 [ set rnd-seed new-seed ]  ;; Check the value of the random seed: if out of limits, choose a proper value.
  random-seed rnd-seed          ;; Seed the random number generator.
  if (not rserve-init) [ stop ]   ;; Clean connect to R (reset memory, clean all weights) and stop if Rserve is not running
  set timeControl       1       ;; Used in the GO procedure
  set numberOfProperties 10       ;; Default, modify the parameter to increase the vector
  make-agents   ;; Create agents
  init-matrices  ;; Create matrices full of -1
  reset-ticks
end
```

Check if the user wants to change the random seed at every setup and that the seed in comprised in the legal range accepted by the random generator.

2.1.2 Micro-phase – meeting others

```
to check-neighbors ;; Create links to communicate with other turtles
  ask turtles with [other turtles-here != nobody]
```

```
  [ ;; Only the turtles with other agents on the same patch
    let tmpIndex random (numberOfProperties) ;; Return a number [0, numberOfProperties -1]; it's an index, so it's all good.
    let tmpValue item tmpIndex properties    ;; The value in that position.
    create-links-to (other turtles-here)     ;; The same value communicated to all the other agents
    [
      set color [color] of myself  ;; For clarity sake
      set index tmpIndex           ;; Link-own attribute
      set value tmpValue           ;; Link-own attribute
    ]
  ]
  check-matrices  ;; Prudential control on limit number of rows used
end
```

Note as the first thing is selecting a restricted subset of agents using as criterion the presence of other agents on the same patch. After this, the communication mechanic is trivial: the turtle share the same information with all the others creating a link towards them and store in its attributes the position and the value of the property it is communicating. Then, the routine to check if a matrix has reached its maximum dimension is called. This is due to the matrix is emptied if the next wave of data exchange would lend it to break the row number limit –remember section 1.4.1.

2.1.3 Micro-phase – updating memory

2.1.3.1 Retrieving information

```
to update-agents   ;; Read links and update the matrices.
  ask turtles with [count my-in-links > 0]
  [ ;; All turtles with incoming links.
    let comm-prob ifelse-value (color = red) [red-comm-prob] [green-comm-prob]   ;; Set the relevant value of communication probability.
    while [count my-in-links > 0]
    [ ;; Cycle all the links of each turtle.
      let aLinkToMe one-of my-in-links        ;; Select a link.
      let otherIndex  [index]   of aLinkToMe   ;; Extract the index of the property of the other turtle.
      let otherValue  [value]   of aLinkToMe   ;; Extract the value of the property of the other turtle.
      let otherColor  [color]   of aLinkToMe   ;; Extract the color of the other turtle.
      let myValue item otherIndex properties   ;; The value of the same-index agent property.
      if ((random-float 1) < comm-prob) [ update-memory otherIndex otherValue otherColor ]  ;; Call the instructions to update memory.
      ask aLinkToMe [ die ]   ;; Delete the link and start over (if there are more of them).
    ] ;; End while
  ]   ;; End ask
end
```

The first cycle is on a subset of agents having incoming links. The second is inside each agent environment, processing all the links. The code is fair simple: a link is randomly chosen between those still available and all the data it carries are extracted. Then, note the criterion on which it is decided whether memorize the

data or not: the probability of communication set by the user for the breed to which the agent belongs to (so, on the listening side). All further work is delegated to the second procedure, *update-memory*.

2.1.3.2 Filling the matrix

```
to update-memory [otherIndex otherValue otherColor]
[…]
 ifelse ( (myLastIndex = "") or  ;; No row to update at all OR already updated (so, not -1) with a different value
       ( (myLastValue != -1 ) and (myLastValue != otherValue) ) )
 [
  let tmpResults (fill-new-row myLastRow otherIndex otherValue dataMatrix numOfObs) ;; Reporter: report a list [updatedMatrix updatedCounter]
  set dataMatrix  first tmpResults  ;; Get the filled matrix
  set numOfObs    last  tmpResults  ;; Get the updated counter
 ]
 [ matrix:set dataMatrix myLastIndex otherIndex otherValue ] ;; Update last row with the new value, no new row inserted.
 […]
end
```

Hereby are omitted all the instrumental parts of code, stressing the core control of the procedure. This is the underlying logic: if the agent has a previous contribution to the information matrix, i.e. it does exist a row with its id number in the last column, and the value of the to-be-updated property is *-1* (indicating that that property is untouched yet), the value is just updated. If, otherwise, there is no previous row signed by the agent at all, or, on the other hand, there is, but with that property already updated (and of course different), a new row is filled via the instrumental procedure *fill-new-row*.

To be as clear as possible, let's use an example and suppose that a red agent (the number *151*) has a *green* incoming link with an index of *7* and a value of *0*. The link is processed, the probability condition is satisfied and then the *update-memory* procedure is invoked. The agent *151* selects the matrix containing the observations of the red agents about the green ones, check if there are previous contributions of its own; a few possibilities may occur:

I. There are no rows signed *151* (with *151* in their last column): a new row is then signed; this would be full of *-1*, except for the 7th position, which would be *0*.
II. There is at least a row of our agent and its 7th propriety is *-1* (i.e. never touched before): the propriety is updated and no further action is required.
III. There is a row of our agent, but the 7th propriety is already *0*: nothing happens.
IV. The last row of our agent has *1* in the 7th propriety. A new row is inserted, starting from the last one: the new one will be a copy of the old, except for the 7th propriety, being now equal to *0*.

2.1.3.3 Fill in a new row

```
to-report fill-new-row [myLastRow otherIndex otherValue dataMatrix numOfObs]
 if (myLastRow != "")
 [   ;; There is a previous contribution, so I copy those values in the new used row.
  while [length myLastRow > 0]
  [   ;; I update each value of the new row with the last values
```

```
    let tmpIndex (numberOfProperties + 1) - (length myLastRow)  ;; Iterate from 0 to 10 (because with length 0 while-cycle skips)
    matrix:set dataMatrix numOfObs tmpIndex (first myLastRow)   ;; Update the value with the first element of the list.
    set myLastRow butfirst myLastRow                            ;; Get rid of the first element of the list.
  ] ;; End while
 ] ;; End if
 ;; There is no a previous row to copy, I just fill-in a new one from scratch
 matrix:set dataMatrix numOfObs otherIndex otherValue   ;; Update the property of interest.
 matrix:set dataMatrix numOfObs numberOfProperties who  ;; Update the id of the agent (superflous if the new row is the copy of a previous row.)
 report list dataMatrix (numOfObs + 1)        ;; Report the values as a list.
end
```

This procedure operates in a slight redundant way: if there is a previous row inserted by the agent and passed as an argument, it fills another row using those values. Then, by default, update the property and the agent id: this is superfluous if the row was a copy of an old one (the last column already has the id), but ensures a strong and flexible structure if it is the case in which the row is brand new (and so it needs also the signature of the contributing agent).

2.2 R code

The code here detailed is related only to the red breed; although this may be an arbitrary choice, the insight is diminished in nothing: the green breed related code is equal by all the extents, with the only difference of the matrices used as data source and of consequent variable names.

2.2.1 Finding weights – red breed

```
to find-red-weights
 ;; Transform matrices in lists
 let tmpRedFromRed      matrix:to-column-list (matrix:submatrix redFromRedInfo      0 0 redFromRedRows numberOfProperties)
 let tmpGreenFromRed    matrix:to-column-list (matrix:submatrix greenFromRedInfo    0 0 greenFromRedRows numberOfProperties)
 rserve:putlist   "redFromRed" tmpRedFromRed
 rserve:putlist   "greenFromRed" tmpGreenFromRed
 ;; Break in two steps the creation of the input matrix.
 rserve:eval      "redInput1 <- matrix(unlist(greenFromRed), ncol=10)"
 rserve:eval      "redInput2 <- matrix(unlist(redFromRed), ncol=10)"
 rserve:eval      "redInput <- rbind(redInput1, redInput2)"
 rserve:eval      ( word "redOutput <- cbind(c(rep(c(1,0)," greenFromRedRows "),rep(c(0,1)," redFromRedRows ")))" )
 rserve:eval      "redOutput <- matrix(redOutput,ncol=2,byrow=TRUE)"
 ;; Implement the reuse of weights.
 let tmpCodeStr1 ""
 let tmpCodeStr2 "redWeights <- nnet(x=redInput, y=redOutput, size=6, maxit=1000, trace=FALSE)"
 let tmpCodeStr3 ""
```

```
[…]
 rserve:eval (word tmpCodeStr1 tmpCodeStr2 tmpCodeStr3) ;; Create the weights -red breed.
end
```

In the original matrices we keep track of the id of every single contributing agent. This data is useless in the instruction of the neural network and is discarded by the means of a sub-matrix selection, allowing to exclude the last column and to select only the rows truly filled by the agents (leaving all the unused, full of *-1*, rest of the matrix). This sub-matrix is, in the same line of code, transformed in a NetLogo nested list. The operation is repeated for the second matrix containing the data gathered from the red breed.

The lists are directly stored in R and from now on, processed in the R workspace. They are transformed into matrices, with the trick of flatten them to avoid misreading of the data by the use of the *unlist()* R function. The two matrices are then joined, simply appending all the rows of one to the other, to give birth to the unique input matrix. This workaround is necessary due to the long to be perfect compatibility between data in NetLogo and in R.

The output matrix has one row for each row of input, a vector of only two elements, *(0, 1)* for data about red agents, *(1, 0)* for data about green ones. In this way, the network is trained to recognize the data patterns related to each breed, matching them to two different vectors. The use of a single number output (*0* for a color, *1* for the other) is impossible in this kind of situation, as would imply a magnitude order between the data patterns –although a prediction error is still present; see sections 3.1.5.2 and 3.2.1.

The computation of the optimal weights is done in itself by the *nnet()* function, belonging to the library *nnet*[10]; this function uses a single hidden layer made of six nodes (as set by us) and the maximum number of iterations to reduce the error is set to 1000. The last argument, *trace*, controls the verbosity of output (here turned off).

The creation of the weights relies on the evaluation of three joint strings; this kind of weird structure finds its reason in the reuse of the previous weights, as explained in the next section.

2.2.2 Reusing weights – red breed

```
to find-red-weights
[…]
 ;; Implement the reuse of weights.
 let tmpCodeStr1 ""
 let tmpCodeStr2 "redWeights <- nnet(x=redInput, y=redOutput, size=6, maxit=1000, trace=FALSE)"
 let tmpCodeStr3 ""
 if (reuse-red-weights?)
 [  ;; Create a control in R, so that even the first cycle is fine (because there are no old weights).
  set tmpCodeStr1 "if(exists('redWeights')) { redWeights <- nnet(x=redInput, y=redOutput,
Wts=redWeights$wts, size=6, maxit=1000, trace=FALSE) }else{"
  set tmpCodeStr3 "}"
 ]
 rserve:eval (word tmpCodeStr1 tmpCodeStr2 tmpCodeStr3) ;; Create the weights -red breed.
end
```

[10] Venables, W. N. & Ripley, B. D. (2002): *Modern Applied Statistics with S. Fourth Edition*. Springer, New York. ISBN 0-387-95457-0

If the user chooses to reuse the weights, an additional control is implemented in R. As said, one of the limitations of use R via Rserve from NetLogo is the obligation to run the *if* blocks all at once in the same evaluation: if the control on the Boolean *true/false* variable is true, the once empty string variables are set so that:

I. *tmpCodeStr1* contains the *if* control and body, plus the *else* opening.
II. *tmpCodeStr2* remains unchanged, being now the *else* body.
III. *tmpCodeStr3* is now the closure of the *else* statement.

The union of these three variables gives now a complete *if-else* statement, reusing the weights only if they are defined, so to get rid from the problem of potential errors raised trying to use weights not (yet or anymore) existent.

2.3 Online version specificities

As already said, the online version cannot rely on the power of R, so is shut out from the ability of instructing neural networks. An *escamotage* to this problem is found by allowing to the user to choose a probability for a correct labeling. This, of course, changes radically the model structure.

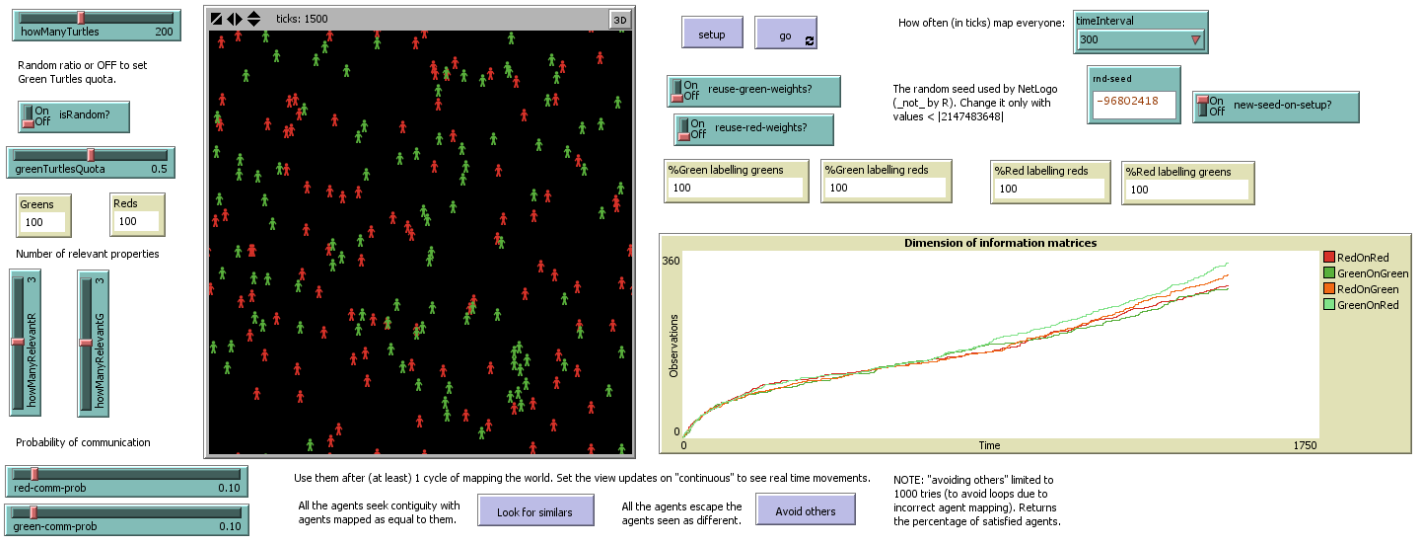
```
to make-reds [n]
 create-reds n
 [
  […]
  ; On-Line version code only
  set seenByRed ifelse-value (random-float 1 < reds-see-red) [red][green]
  set seenByGreen ifelse-value (random-float 1 < greens-see-red) [red][green]
 ]
end

to make-greens [n]
 create-greens n
 [
  […]
  ; On-Line version code only
  set seenByRed ifelse-value (random-float 1 < reds-see-green) [green][red]
  set seenByGreen ifelse-value (random-float 1 < greens-see-green) [green][red]
 ]
end
```

The attributes used to store the result of the identification by the means of the neural network are now filled with a value which is due to a simple probabilistic control against the value set by the user. This operation is done in the agent creation moment (called by the *setup* procedure), thus making useless the *micro*-phase of data gathering. Despite this, is a good option to amend the lack of R.

3 Experiment results

In God we trust, all others bring data.
-William Edwards Deming



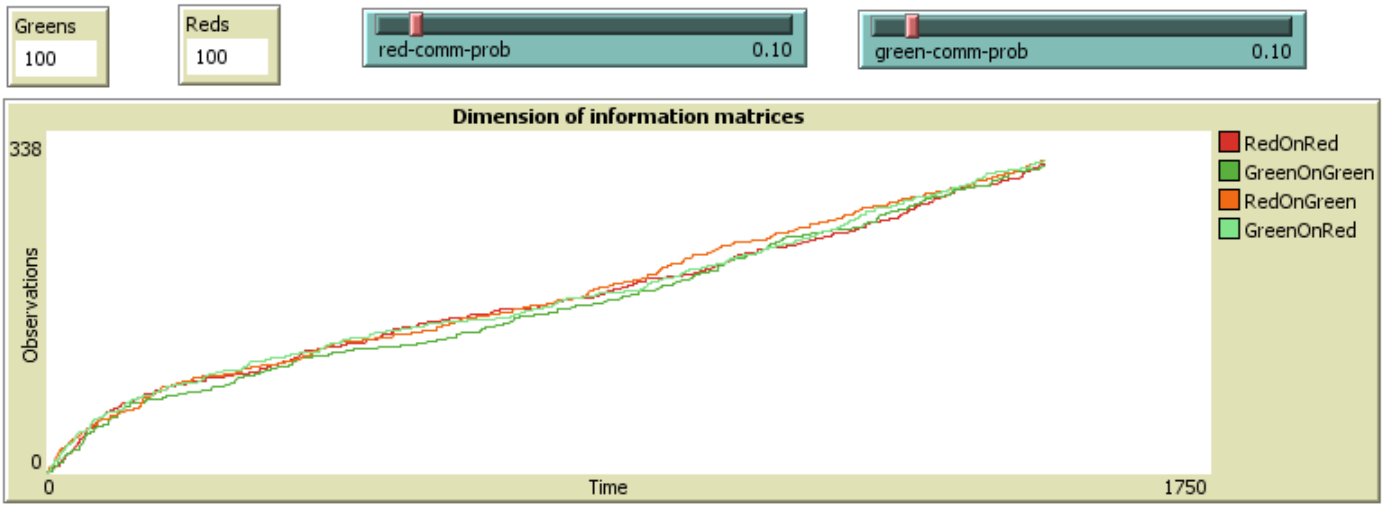
We have to say, straight and clear, that the model just works. The agents get knowledge and information, the data matrices grow up well and the neural network learns to distinguish the *Reds* from the *Greens*. This may be seen as a problem: the errors in the identification of the agents color fade away with amazing speed, the instruction of neural network is fast and there is no adjunctive value. Let's see why this first impression is absolutely wrong.

3.1 Data matrices

As expected, the speed with which the matrix fills up is tied to the probability of communication of each breed and to the total number of agents. While the role of the first is obvious (the more probably a breed would share information, the more likely there will be a lot of data about it), we must not forget that any change to the total number of agents affects both the extensive side, modifying the probability of meeting another agent, and the intensive side, affecting the number of agents on each patch.

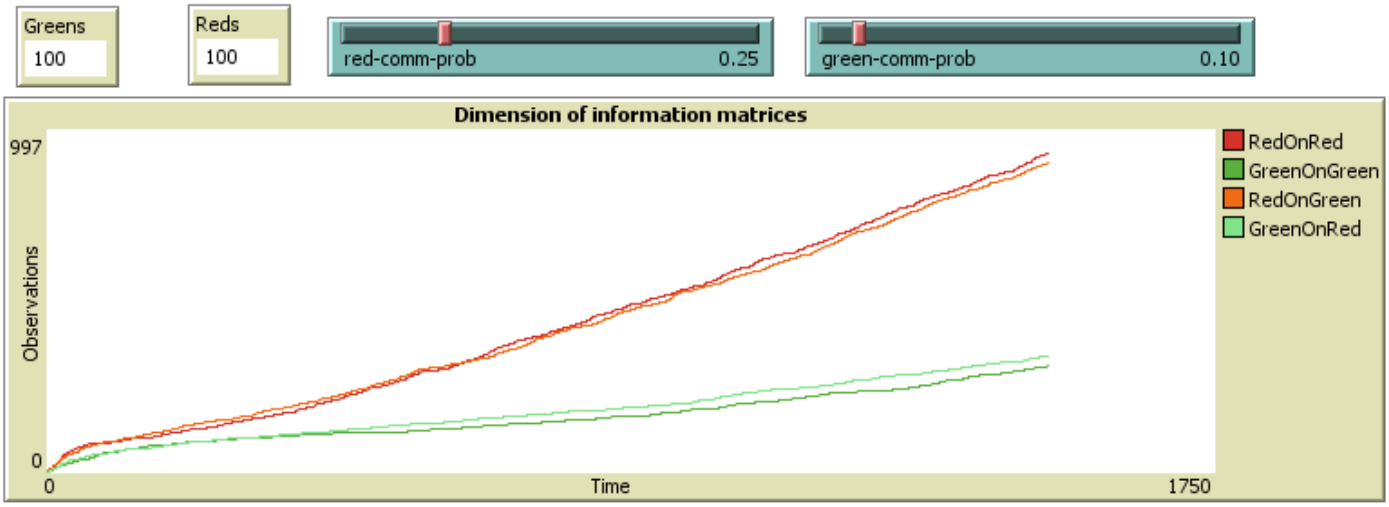
Hereby some cases follow, with the relevant graphic attached -all saved after 1500 ticks for consistency. Keeping in mind all that was said earlier about the memorization system structure, it would be plain to understand that *Dimension of information matrices* is a plot of the number of rows truly used for each matrix, graphically representing the evolution of the same variable used to track the contributions number: the real matrix rows number is fixed and determined in the *setup* procedure.

3.1.1 Plain vanilla – 200 agents



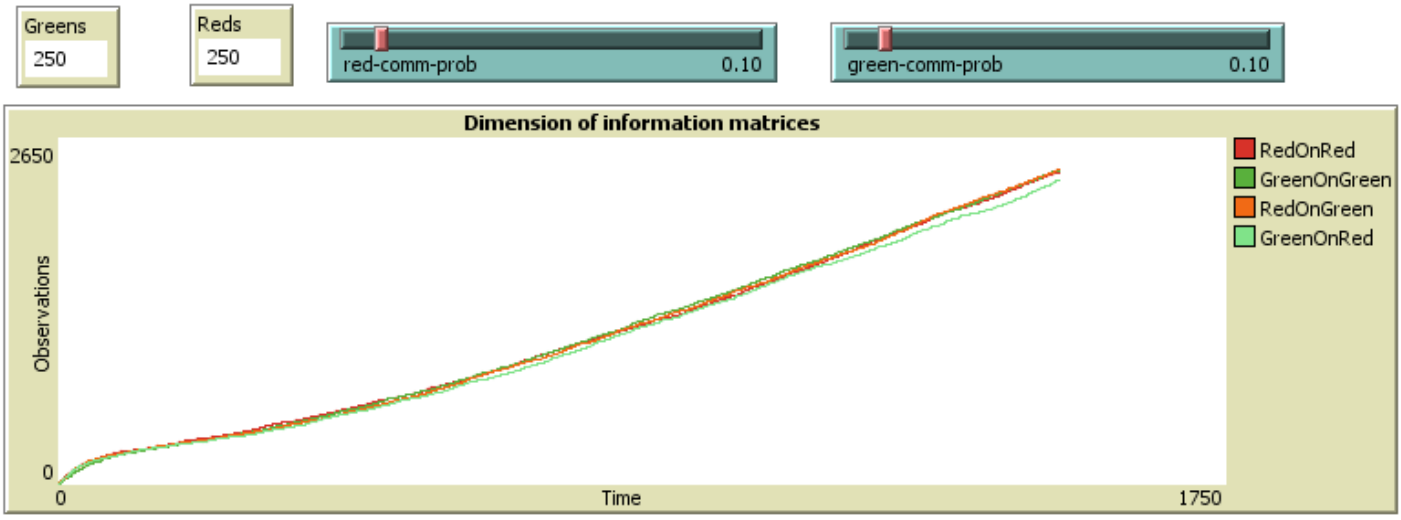
Here it is the simplest configuration of all: a medium-small number of agents equally distributed on the two breeds, with no difference in the *communication probabilities*. We will use this situation as a sort of benchmark against which compare the other option sets.

3.1.2 Probability rises – 200 agents



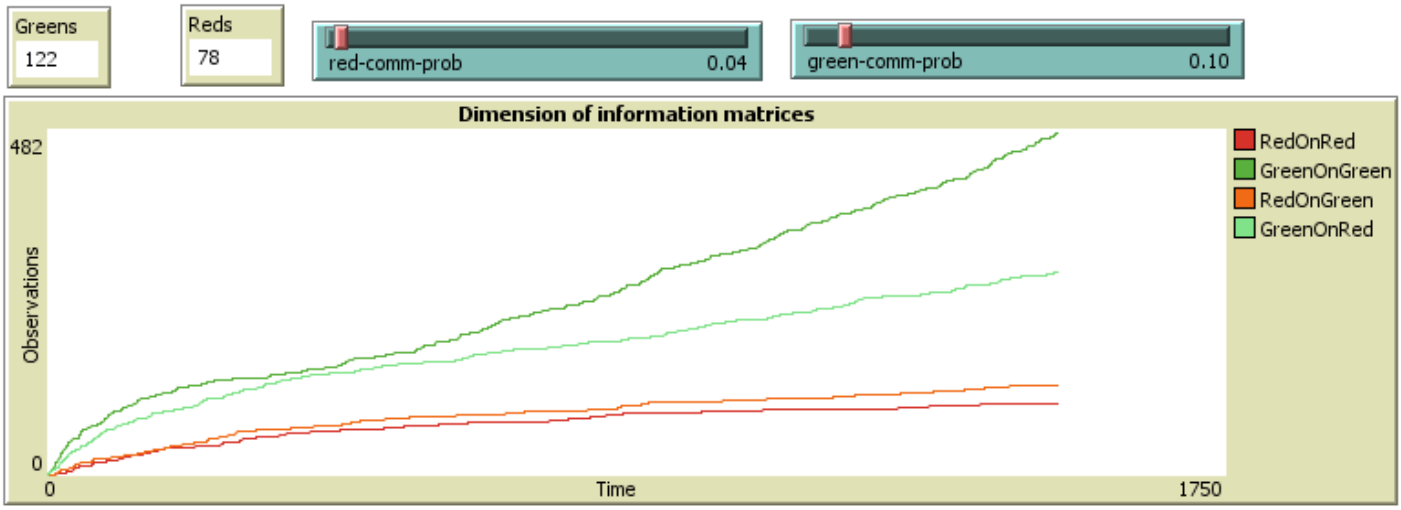
The probability that a red agent would listen what another agent tells him is now more than doubled, till a solid 25%. The effect on the matrices is evident and expected: the red agents are now much more receptive, so they feed the matrices much more often making them grow to about the triple than the green-fed matrices.

3.1.3 Number rises – 500 agents



Probabilities are even again, but the number of agents has now jumped to 500. Looking to our benchmark, the situation may not appear to be so different until but looking to the magnitude side. The matrices have now more than 2500 observations each, 8 times and over the benchmark case. This large rise is, as we already said, to be ascribed to the double effect on both the intensive and the extensive margin of the agents' number variations.

3.1.4 Go random – 200 agents



Now, it is time to test our insights about the relation between the agents number, the probability of communication and the dimension of the information matrices: in this peculiar case, we have more *Greens* than *Reds* (a randomly determined quantity) and the probability of communication of red agents is lower than that of green ones.

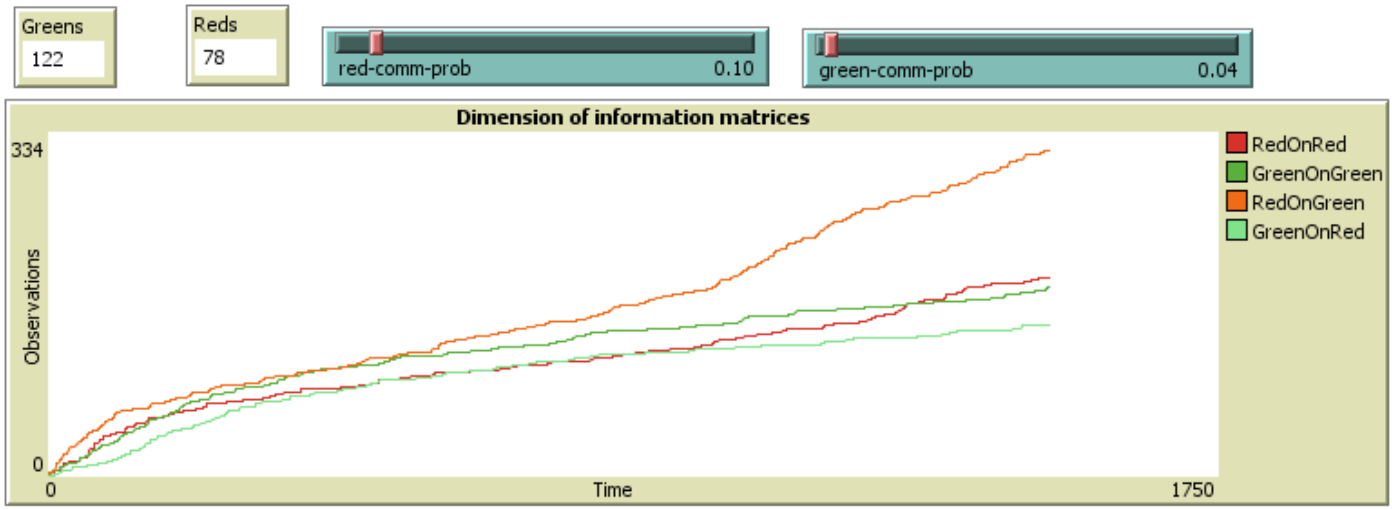
Ever since the first look, all goes as we expected:

a) The matrix of *green observations on green agents* is the biggest, benefitting from both the higher number of agents observed and the higher probability of update the matrix itself.
b) The matrix of *green observations on red agents* is the second in line, penalized by the relative scarcity of red agents whose information to look at.

c) Detached by an appreciable length, it comes the *red observations on green agents* matrix, penalized by both the number of agents updating the matrix and by the lower probability they would choose to do so.

d) Last, the *red observations on red agents*, which suffer a logic disadvantage in the face of odds of finding two red agents on the same patch and a double drawback from the lower probability of updating the matrix: information of a breed on the breed itself are retrieved when two identical agents stand on the same space; if both of them have a very low probability to update the matrix, here comes the trouble.

What would happen if, *ceteris paribus*, we swap the probabilities? In a situation with more green agents, but less apt to save the information they get, and red agents much more willing to update the memory matrix?



And here it is the answer. Partially, it reflects what we already learned: the information set from red agents is now larger, as they update the matrix with a higher frequency, and the largest matrix is the one of observation on green agents, being the majority in the world. Maybe unexpected is the magnitude of the change: with no variation of sort to the number of agents, a simple swap in probabilities almost completely reverse the order of magnitude of the matrices, teaching a lesson of carefulness when a change in probability values, albeit slight, is done.

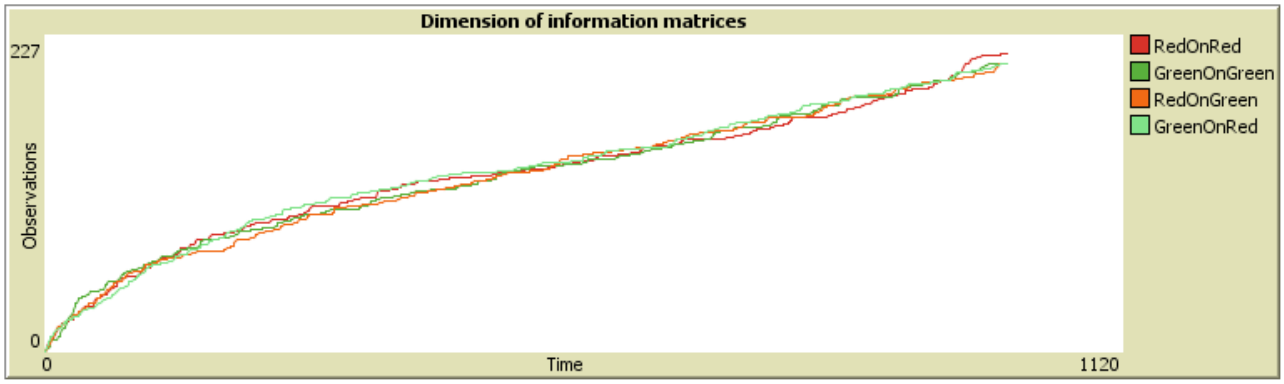
3.1.5 How do the matrices affect the mapping?

Given that the central point of the experiment is the ability of a collective intelligence to recognize the breed of each agent using a neural network, we must ask ourselves whether the dimension of the matrices is relevant or not to the purposes of the neural net mechanism.

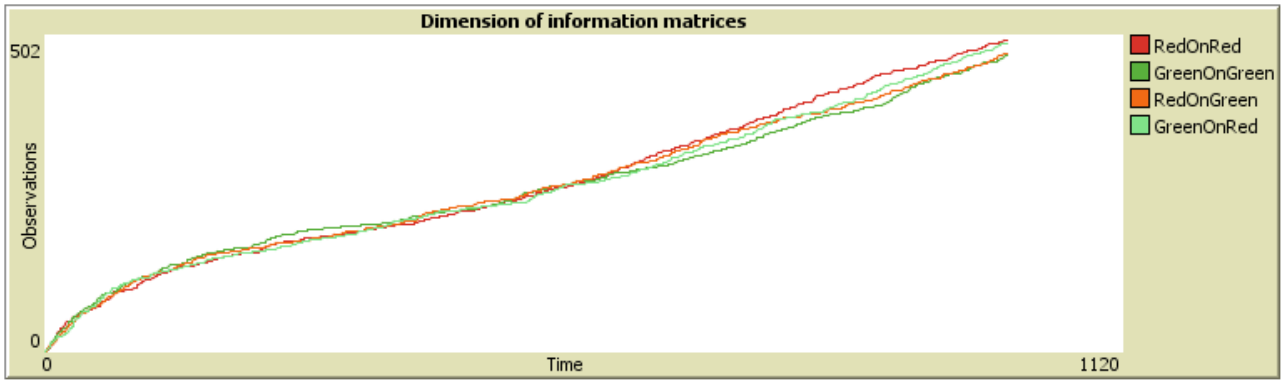
The theoretical answer, without doing any simulation, is straightforwardly *yes*. Being the network a nonlinear statistical model with the task of identifying breed-related patterns, it is obviously affected by the amount of *information* available. The growing rate of the matrix is then a critical variable to take in account when we want to make inference starting from those data: being the variability of the agent property vector limited, bigger the data set, better the work of the neural net. There is a strange effect, though, which is worth discussing –see sections 3.1.5.2 and 3.2.

3.1.5.1 Shape of filling trend

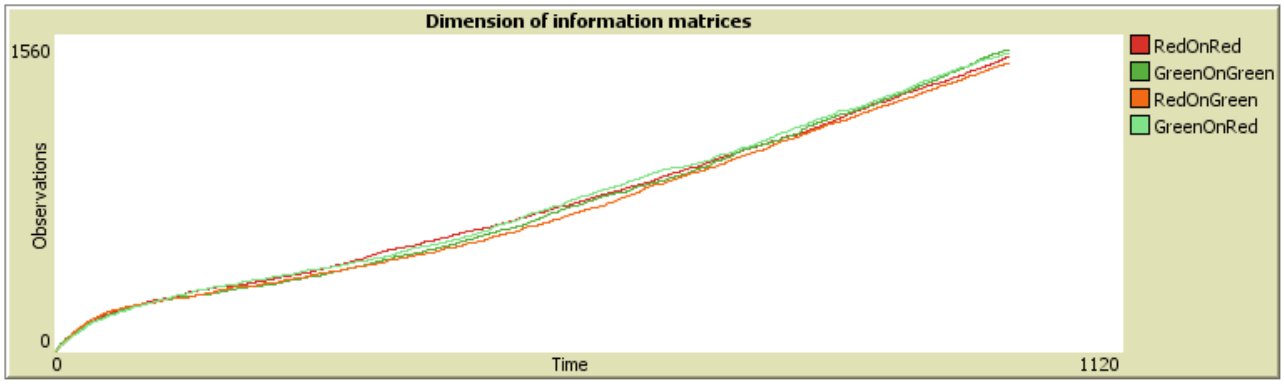
Looking at the examples above, someone may notice the peculiar shape of the dimension graph and ascribe it to the fate whim. To get rid of doubts, let's look at the graph produced after 1000 ticks for 200, 300 and 500 agents, communication probabilities equal and unchanged.



200 Agents

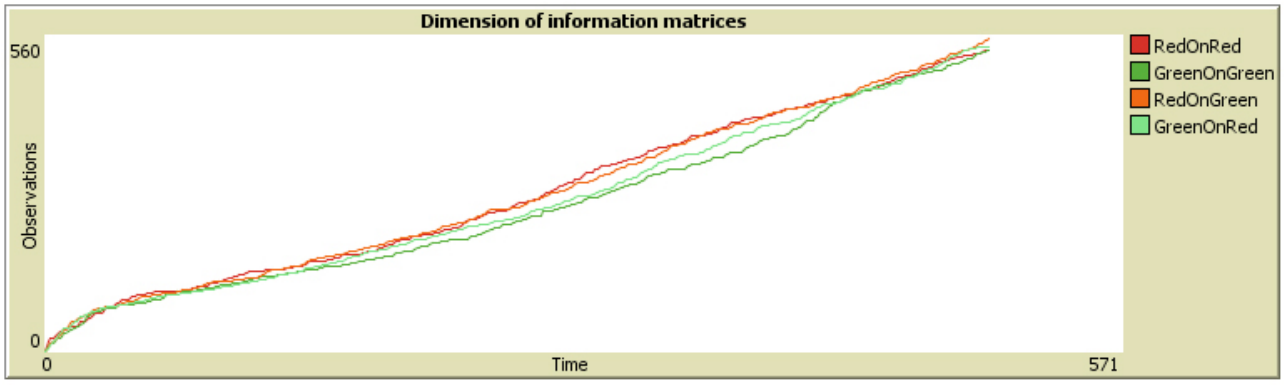


300 Agents

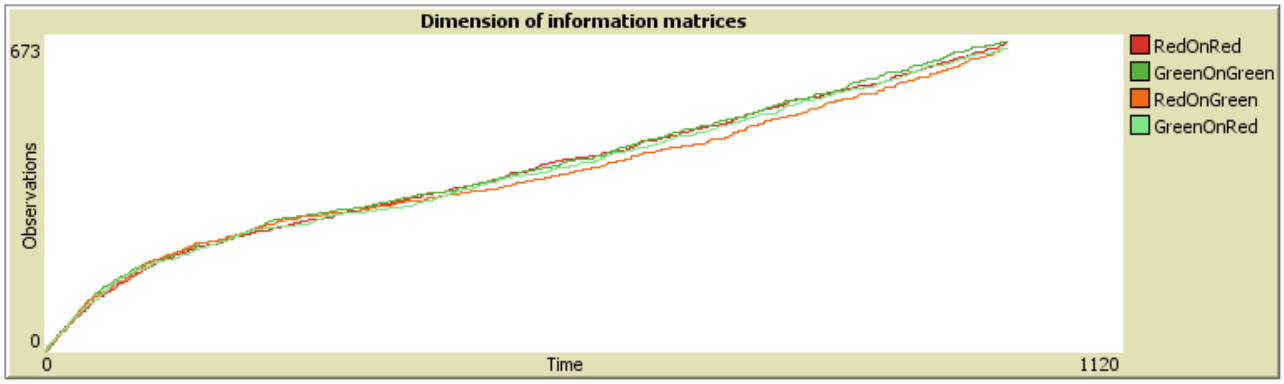


500 Agents

No, there is no casualty here. To test the persistency of this S shape, we can also try to change the probabilities of communicating, although leaving them equal; a disparity in them would always produce the same pattern, but it would be far less evident for mere graphic scale reason, with the breed blessed by a higher probability gathering information much faster than the other.



200 Agents; both probabilities set to 50%



500 Agents; both probabilities set to 5%

We can try to give an explanation having in mind the system of memorization used and the cases in which a new row is filled: when no previous row exist and when, although the existence of a row, the peculiar propriety is already filled with a (different) value. In the first moments, then, being the matrices empty, the number of rows used rises fast, followed by a moment in which there is more *updating the existing rows* than *filling new ones*: this is the slope change observable for every set of options. After this moment, equilibrium between new rows creation and old rows updating is reached: in the long run, the *last row* of an agent is entirely full of values, with no *-1*. Every new piece of information has thus the 50% (it's a binary values system) odds of generating a new row (yes, if it's different, not otherwise).

3.1.5.2 Asymmetric mapping goodness – first come, better serve

An unexpected and very interesting result regards the goodness of predictions on the agents color between the two breed. Taking the benchmark case, with the same number of agents per breed, the same probability of communication and a very similar filling trend for the matrices, we would expect similar results in prediction outcomes. And we would be wrong.

It is indeed observable a visible difference between the percentage of good mapping for *Greens* and for *Reds*, with the latter collecting a series of very bad results until the matrices reach a dimension close to the total number of the agents (which, recalling the construction method, would make the row number of the input matrix roughly the double of the number of agents).

An explanation of this weird behavior was found relying in the vector identifying the breeds: putting the dimension of the matrix aside, it appears to be easier for the net to match data with the vector *(1, 0)* rather than the vector *(0, 1)*. Exploring further this peculiar side of the problem would have lead our research outside its original boundaries, so we just give the fact along with the intuitive motivation without supplying any evidence. Further details are anyway available in the following section.

3.2 Mapping the world

After all we said about the information matrices and the impact they have on the neural net instructing process, we have at least one certainty: we need to put a lot of care in the choice of the number of agents and, most of all, of the probabilities of communication; rise them too high and you will be flooded by data obscuring the role of other modifications, lower them too much and the results will be bad no matter what. The problem is, to put it bluntly, that the neural net is way too efficient and, when reached a certain amount of information, it just never fails.

Starting from the plain vanilla benchmark model (200 agents, probabilities both set to 10%), we will try to modify the amount of distinctive properties and to study the effect of a reuse of the weights; we will perform these experiments one at a time to be sure what cause ascribe the effect to.

3.2.1 Significant properties

At first, we have to understand how the mapping system really works. We have already seen in depth the structure of the input and that of the output matrix. The mechanic of the neural net wants that, once the weights are found and computed, they can be used to predict an output, given an input. In this experiment, the input is made by the list of properties of a certain agent, while the output is the aforementioned vector of two elements.

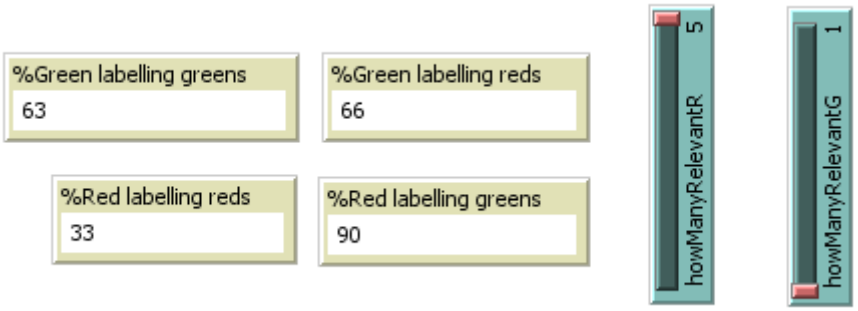
Now here comes the trick: NetLogo decide the color forecasted by the neural net just looking at the biggest element of the two. In this way, there will always be a forecasted color: *red* if the first element is bigger than the second, *green* otherwise. Also, the diminution of the number of relevant properties has a softened effect on the goodness of the prediction: the error produces an increment of green predictions, pushing up the percent of rightful *seen as green* agents.

To cut a long story short, lowering the relevant properties number has little or even no effect: the neural network is just too efficient and reducing the distinctive pattern is only a minor hindrance on its way, an effect that we can't, by the way, correctly asses due to the control structure. So, was the implementing of this functionality only a complete waste of code, memory and time? Luckily, it was not.

If lowering the amount of meaningful variables is meaningless, raising them still has sense and produces foreseeable results. You can almost think to the list of properties as the genetic code of the agents, with the head of the sequence being the information common to all the members of a given species. If more genes specific of one species are added, the individuals become far more similar one to each other and then (even more) easily recognizable.

Plain vanilla; 200 agents, probabilities set to 10% - After 100 ticks

As expected, the number of rightful green predictions is high (there is also the error on the prediction of red agents), while the red agents are poorly labeled by both breeds. Now, let's see what happens if the amount of relevant variables is changed.



Plain vanilla; 200 agents, prob. to 10% - After 100 ticks

The ability to recognize the green agents is still intact, despite the singular relevant property (there is still the error on the identification of the *Reds*). The big change is visible on the identification of red agents, which is now significantly higher, despite the already said difficulties affecting the process of recognition of the red breed. This result confirms the intuition on the role of this setting.

3.2.2 Reuse of weights

Setting to *On* the option to reuse the weights means subtracting from the random generator the choice of the initial weight set and proceed to the optimization starting from the already optimized weights of the previous network instructing routine. This non-random choice of the initial sets of weights to optimize produces three main outcomes, only appreciable in the dynamic of the working model.

3.2.2.1 Slowness in improving predictions

Use again the old weights means trying to reduce the error adding new data from which make inference, but using an already convergent set of weight. This softens the impact of the new data, tying the result to the previous dataset having produced the weights.

3.2.2.2 Persistence of good predictions

A one line recipe: get to a good prediction gathering a large amount of data, and then reuse those weights, tuned on your large matrices. Now it is possible to empty the matrix and equally get flawless results; the new tuning processes are just irrelevant, as if the strength of the old weights, empowered by all the previous observations, was too much for the new data set to make a sensible change.

3.2.2.3 Pride of convergence: the absolute prejudice modeled

Sometimes, it happens. The already mentioned slowness and persistence combine themselves and nail the correct predictions ratio to some number, with almost no (or anyhow very small) fluctuations. Albeit rare – it seems to show itself about one time out of twenty- this could be seen as an expression of a prejudicial attitude of one breed: the opinions of the collective intelligence (the weight-set) are so strong that no matter how many data we add, they will not change.

4 Conclusions and new beginnings

> ***Whenever an affirmative proposition is apt to be verified for actually existing things, if two things, howsoever they are present according to arrangement and duration, cannot suffice for the verification of the proposition while another thing is lacking, then one must posit that other thing.***
> ***-Walter Chatton,* Lectura**

The focus of this experiment was testing the capability of a neural network to act as a collective intelligence[11], able to recognize agents starting from their own set of opinions and/or features and the earlier detailed results prove that, in what was the original scope, the model was a success. There are, however, some more words we may spend to both contextualize the model in the field of the behavioral research and mention interesting additions which may be made.

4.1 Opinion dynamics and consensus formation

Any quick search for the last studies in the field agent-based models applied to *opinion and consensus* will probably get the same answer: the starring role is played by the Bounded Confidence and, even more, by the Relative Agreement structures.
However, the field of application is not, as It may looks, exclusively sociological: under the label of *opinion dynamics* comes a huge variety of models ranging from –of course- sociology to hard sciences, from the study of a population of human beings, to pure mathematical applications, and both the BC and the RA models are easily adaptable to modeling physical problems.

Different by many lengths, the models of this broad class focus on two main cornerstones: agents interact among themselves and they do change their opinions –with modes, methods and rules heavily varying among models. In the first hinge is pivotal in our work, is in the approaching the second that all the difference emerges: our agents are stubborn.

The term itself, *opinion dynamics*, suggests a dynamic behavior of the opinion, which is absolutely absent in this case. It is not a simplification, but a main difference in goals and aims: the focus of our work was not to study the issue of the *consensus formation*, nor we were trying to achieve a form of convergence in the global opinion of a population, but rather to study after how long and how well a population would achieve a knowledge good enough to be used to recognize its own kind among all the others. There is a dynamic, of course, but it is centered in the information gathering mechanism, while the opinion was absolutely static.

4.2 Possible extensions

Several simplifications were made, in the experiment ideation process as well as in the coding of the NetLogo model, due to the nature of this dissertation and the boundaries set by our technical limitations. We will anyhow provide a few hints of the wonderful problems contained *in nuce* in our work waiting to be explored and studied.

[11] According to the working definition used by the MIT - Center of Collective Intelligence: *Collective intelligence is groups of individuals doing things collectively that seem intelligent.* The whole document is currently available at: <http://cci.mit.edu/about/MaloneLaunchRemarks.html>

4.2.1 Widening the scope

So, we have a simple, yet working, model which is not focused on how the agents modify their opinions, but on how good they are to learn something, even with the possibility of add a bit of prejudice. We could easily study the effect of a mystifying propaganda, centered on spreading false information about a different breed: agents both have this false information and can make their own experience; will they be able to overcome the false data –and how long will it take?

Another example: we keep track of what agents contribute to the memory matrices. What if the other breed could identify the most contributing agents and obliterate their memory (erasing their contribution from the common data storage)? How heavily the network ability would be affected?

Finally[12], take the moves from the simple segregation procedures implemented and suppose having more than one breed, each with a different segregation rule, so that the breed A can't tolerate the breed B, but goes just fine with the breed C; B flees from both A and C; C members can't endure to be close to their own kind. Then, apply those rules not to the real breed, but to the breed guessed by the collective intelligences.

4.2.2 Editing the code

Of course, this peculiar category of possible expansions is nearly infinite, as it is stated. Without intent to set boundaries to the fantasy on anyone, there are some minor modifications to the code which would allow to explore different sides of this experiment.

As an example, take the forecast control, the yet debated *if-else* statement, and modify it so to point out one of the breed only if the output value lies in some confidence interval around the expected result, leaving all the others empty. It would be possible to track down the error evolution and study it in function of the amplitude of the confidence intervals. Then make them dynamic, maybe ruled by endogenous variables.

Another possible tweak would be a few lines of code dumping the data on a text file, to then proceed with a Monte-Carlo simulation (maybe directly using R) on the goodness of predictions result with a locked NetLogo random seed and still varying random initial weights. The same could be done with several hidden layer dimensions, to study the efficiency of the network with statistical tools.

Finally, one could try to slightly modify the parameters ruling the number of properties or, much more interesting, the kind of variables: what would happen, if, instead of a binary 0/1 kind of features, we would have some values taken from the set of real numbers?

Try and explore, but be careful.

> ***Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.***
> ***-Douglas Hofstadter,* Gödel, Escher, Bach: An Eternal Golden Braid.**

[12] To not bore the reader and stay close to the object of the model; but imagine to make the agents gather information about the patches and act (build a house, a farm, a factory) according to what they think (the network forecast) there would be in a given patch.