An agent−based model, used to train an Artificial Neural Network, to forecast the spread of genetic diseases

Francesco Schepis

1 Introduction

1.1 How does it work?

The purpose of my work was to find a way to predict a possible forecast of a genetic diseases.
To do this I used a model to recreate a population of three different kind of agents: healthy agent, carriers agent and sicks agent. To simulate the spread of the disease, two different agents were chosen (without any exception)to form a couple, and then bred to create a new generation.
With the data of first and last population, I trained an Artificial Neural Network (ANN) to predict the diffusion.
The simulation was designed to observe 10 different kind of severity of disease, and to train efficiently, with a large variety of data, the ANN.
At the end there will be an ANN that, with four parameters (% of healthy, % of carriers,% of sick, sick-treshold) will be able to predict the final % of healthy, carriers and sicks after 20 generations.

1.2 The generalization of the simulation model

The model used to simulate the 20 generations is an introductive work, that has to be taken like a base, from which starting to product a model that observe a specific forecast of a particular disease. Let's see the hypothesis done in this generalized model.

For example: it required that the considered disease is not mortal, that the sick agent can be able to mate normally, and also that the disease forecast, from the agents parents to the agents sons, was made with a generalization of Mendelian's idea, thanks to the stochastic cross of the DNA's parents.
An other important generalization is about the sick-treshold, that define the health state of the agents. The idea is to summarise every big and small facet that characterize the disease, and convert all in a single value which representing the severity of the latter.
Naturally this required an important, and probably not negligible, generalization.
All of this to say that, to optimize the use of the ANN, is essential choose a disease that satisfied the hypothesis.

1.3 Ready to react

The main goal of this work is to discover a possible dangerous impact of a genetic disease in a population, so we can plan previously the best strategy to vanquish the disease, both in Economic and Social terms.

Indeed, if a dangerous debilitating and easily transmitted disease comes out, the first concern must be "which negative impact will be in a population?","How much this can affect the mundane life?".

Put simply: focusing on the individual, would be put at risk the regular daily life(not indifferent things in social terms) and in a global view, a society of sick people would not be productive.
Predicting the danger, the react can focus his economic and social forces on prevent the forecast of this particular disease.
There are many action that can be done: naturally finding a cure, but if this is not possible, another react might be done by genetic manipulation through the artificial fertilization or by birth control (obviously most influential in Social terms).

1.4 The combination of two languages to optimize the purpose

The work was made thanks to the use of two different programming languages: **NetLogo** and **Python.**

NetLogo is a free, open-source multi-agent programmable modelling environment, so as to introduce the stochastic behavior(fundamental for the issue).
Python is an interpreted high-level programming language for general-purpose programming.
The first is appropriate to simulate the agents interact and observe the changes in real time.
The second allow to do complicate calculations that NetLogo would not be able to do.
Thanks to the combination of these two programming language it was possible to get data from an agent based simulation, in NetLogo, and use these to train a ANN written in Python.
The code will now be explained focusing the attention in a first moment on the NetLogo part, then on the Python part to use the ANN and finally to the combination of both.

2 The NetLogo code

We write the code to create a random population of sick, carrier and healthy and take data of these 3 values and also of the sick-treshold(value choosen randomly between 10 and 20), then bred them together to product the new generation, it is repeated for 20 generations(ticks), and after that we take again data of the 3 values of sick, carrier and healthy of population.
After that the code clear all and repeat for 100 times, and take at the end two list, one of 404 values(initial % of sick, carrier, healthy and sick-treshold, for 101 times) and the other of 303 (final % of sick, carrier and healthy).
I repeat all to get 22 list of these to extend the data.
Let's see how work the button on the interface.

2.1 Start

First of all, we have to press the button **Start** to clear-all and set the variable "restart" to 0. This variable is use to count the list of value. It will be incremented until 10.

```
to start
    clear-all
    set restart 0
end
```

2.2 Data

Immediately after the **Start** button, we are going to press the **Data** button.
This button set the two list (X and Y),that will contain the initial and final data, like empty list to be filled; then resized the world, on the y axis, to represent, in a appropriate way, the 3 DNA of a healthy, sick and carrier, choosing randomly, in every generation.
The button set also the value “again” to 0.This variable is use to count the components of the 10 list.

```
to data
 set dataX []
 set dataY []
 resize-world 0 32 0 4
 set again 0
end
```

2.3 Setup

Now we can press the **Setup** button to create the population.
We can choose the **size-population** with the **slider** in the interface.
Every agents(turtles) had a particular variable: “**bits”.**
This variable is fundamental, it represent the DNA of the agents. It shall assign, to every single agent, a string with the dimension of the size, on the X axis, of the world (32 components). Every component of the string is chosen randomly between 0 or 1.
After that, the **Setup** button sets the agents hide, because we want to represent in the world the DNA, not the agents.
Then the variable “**sick-treshold”** is set to a random value between 10 and 20.This is a key variable to set the health state of the agent.
The **Setup button** has also different command to “clear” (clear-turtles, patches, drawing, all-plots, output, and reset-ticks) because every times, after 20 generations(ticks),it is recall to restart the simulation.

```
to setup
 clear-turtles
 clear-patches
 clear-drawing
 clear-all-plots
 clear-output
 reset-ticks
 create-turtles size-population [
   set bits n-values world-width [one-of [0 1] ]
  hide-turtle  ]
 set sick-treshold (10 + random 11)
end
```

2.4 Go

When the environment is set, the simulation can starts and we can press the **Go** button.
The code of **Go** button is complicate.
It starts with the command **diagnosi**

2.4.1 Diagnosi:

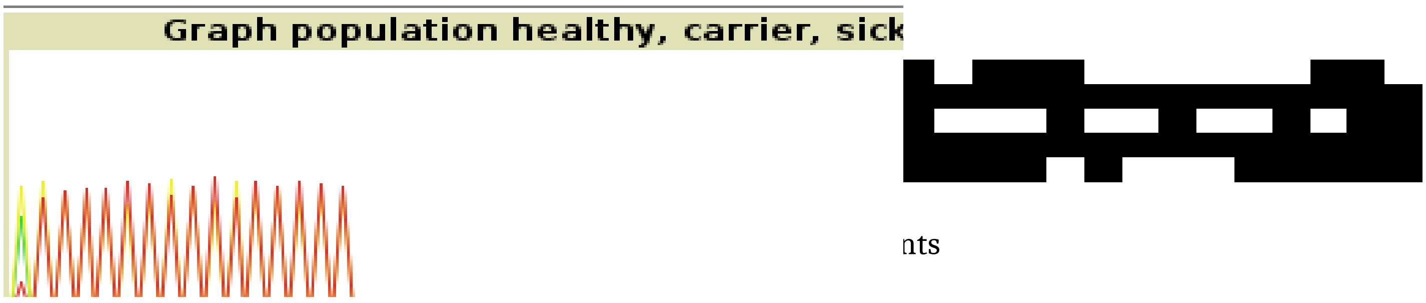
This command analyses the **bits** of every agents.
If the sum of all the 1 present in the string is minor or equal to the **sick-treshold,** the agent changes breed in **sicks**.
If the sum of all the 1 present in the string is greater than **sick-treeshold** but minor or equal to this one plus 5, the agent changes breed in **carrier**.
If the sum of all the 1 present in the string is greater than **sick-treshold** plus 5, the agent changes breed in **healthy**.
Now, if we are in the first generation (ticks = 0), the code add the value of the **%** of healthy, carrier and sick to the **X** list set on the setup command.
After that, another value of **sick-treshold**, that will be the same until the end of the 20 generations(ticks), is chosen randomly. Then this value is also add to the X list, infact it will be the sick-treshold baseline of this cycle.
In the end is called the **update-display** button.

```
to diagnosi
 ask turtles[
  ifelse length (remove 0 bits) <= sick-treshold
  [
   set breed sicks
    ]
  [
   ifelse length (remove 0 bits) <= sick-treshold + 5
   [
   set breed carrier
    ]
    [
   set breed healthy
    ]
 ]
 ]
 if ticks = 0 [
  let q1 count healthy * 100 * (1 / size-population )
  let q2 count carrier * 100 * (1 / size-population )
  let q3 count sicks * 100 * (1 / size-population )
  set dataX lput q1 dataX
  set dataX lput q2 dataX
  set dataX lput q3 dataX
  set sick-treeshold (10 + random 11)
  set dataX lput sick-treshold dataX
 ]
 update-display

end
```

2.4.1.2 Update-display

This command is used to represent, on the world, the DNA's of a sick, carrier and healthy agent for each generation. It chooses, randomly, one agent of three different breeds, and analyses the **bits** of them, changes the color of the patches(black or white) along the X axis. For example: if the second component of the bits of the sick agent is 1, the second patch along the X axis, in the Y value of the axis designed to represent the sicks'DNA, will be colored white, otherwise black.

```
to update-display
 let v one-of sicks
 if v != nobody[
 ask patches with [pycor = 0] [
  ifelse item pxcor ([bits] of v) = 1
    [ set pcolor white ]
    [ set pcolor black ]
 ]]

  let w one-of carrier
  if w != nobody[
  ask patches with [pycor = 2]
  [
  ifelse item pxcor ([bits] of w) = 1
    [ set pcolor white ]
    [ set pcolor black ]
 ]]

 let u one-of healthy
 if u != nobody[
 ask patches with [pycor = 4]
  [
  ifelse item pxcor ([bits] of u) = 1
    [ set pcolor white ]
    [ set pcolor black ]
 ]]

```



Once the command **diagnosi** and **update-display are executed**, the button **go** updates the graph which is represents the counts of the sicks, healthy and carrier for each generation.

Fig2: The variation of the population(sick%,carrier%,healthy%) in function of the generations

Than the button **go** asks to all agents to change breed in **oldgens** (old-generation) and to executes the command **new-generation**

2.4.2 New-generation

This command ask to the agents to choose another partner, from the other agents, to mix randomly their DNA, to create a new-generation and then die.
The command to make the DNA's cross is **crossover.**

2.4.2.1 Crossover

This command takes the two **bits** of the agents parents, divides randomly the string in two parts, equal from a parent to another, and creates two new **bits** for the two new agents that will be obtained.
The first **new-bits** will be given from the first part of the bits of the first agent and the second of the second agent.
The second **new-bits** will be given from the first part of the bits of the second agent and the second part of the first agent.

```
to-report crossover [bits1 bits2]
  let split-point 1 + random (length bits1 - 1)
  report list (sentence (sublist bits1 0 split-point)
                        (sublist bits2 split-point length bits2))
              (sentence (sublist bits2 0 split-point)
                        (sublist bits1 split-point length bits1))
end
```

Now the agents sons can be create.
Through the command **hatch** the two agents sons are created,the first with the first new bits(**item 0)** from **crossover** command , and the second with second new bits (**item 1**). And then they are set with a **newgens** (new-generation) breed.
After that the agents parent **die.**

```
to new-generation
  let parent2  one-of other oldgens in-radius 1000
  let child-bits crossover ([bits] of self) ([bits] of parent2)
```

```
 hatch 1
  [  set bits item 0 child-bits
     set breed newgens
  ]
ask parent2 [ hatch 1
  [  set bits item 1 child-bits
     set breed newgens
      ]
]
ask parent2 [die]
die
end
```

Then the button **go** execute the command **tick.**
When the button **go** is executing, if the **ticks** (generations) are equals to 20, the code add 1 to the value of variable **again** and add to the list Y the last three value % of healthy, carrier and sick. If the variable **again** is minor or equal to 100 the command **setup** and **go** are re-executed, or else respectively one of ten different lists of dataX and dataY, chosen respect to the value of variable **restart**, are set equals to the X and Y, respectively. Then the value of **restart** is incremented of 1.
At the end, if the value of **restart** is minor or equal to 11, then the command **data setup** and **go** will be executed, or else the code will **stop.**
This is the entire code of **go**

```
to go
 if ticks = 20 [
   set again again + 1
   diagnosi
   let z1 count healthy * 100 * (1 / size-population )
   let z2 count carrier * 100 * (1 / size-population )
   let z3 count sicks * 100 * (1 / size-population )
   set dataY lput z1 dataY
   set dataY lput z2 dataY
   set dataY lput z3 dataY
       ifelse again <= 100
   [
     setup
     go]
   [
     if restart = 0
    [ set X  dataX
      set Y dataY
       ]
     if restart = 1
    [ set X1 dataX
      set Y1 dataY
      ]
     if restart = 2
    [ set X2 dataX
      set Y2 dataY
```

```
       ]
      if restart = 3
     [ set X3 dataX
        set Y3 dataY
      ]
      if restart = 4
     [ set X4 dataX
        set Y4 dataY
      ]
      if ricomincia = 5
     [ set X5 dataX
        set Y5 dataY
      ]
      if restart = 6
     [ set X6 dataX
        set Y6 dataY
      ]
      if restart = 7
     [ set X7 dataX
        set Y7 dataY
      ]
      if restart = 8
     [ set X8 dataX
        set Y8 dataY
      ]
      if restart = 9
     [ set X9 dataX
        set Y9 dataY
     ]
      if restart = 10
     [ set X10 dataX
       set Y10 dataY
     ]
     set restart restart + 1
      type "restart=" print restart
     ifelse restart <= 11
      [ data
        setup
      go]
      [stop]
]
 ]
 diagnosi
 set-current-plot "Graph population healthy, carrier, sicks"
 set-current-plot-pen "healthy"
 plot count (healthy)
 set-current-plot-pen "carrier"
 plot count (carrier)
 set-current-plot-pen "sicks"
 plot count (sicks)
```

```

ask turtles[
  set breed oldgens]

ask oldgens[new-generation]
tick

end
```

3 The Python code

It is the moment to start speaking about the Python code write (in emacs-script) to use the ANN.
As I said before, the use of Python is justified by his computing power.
In fact Python is really comfortable to work with matrix and array in n-dimension.
Before explaining the code, let's do a little introduction to the Artificial Neural Network that I used.

3.1 Artificial Neural Network

An ANN is based on a collection of connected units or nodes called artificial neurons(a simplified version of biological neurons in an animal brain). Each connection (a simplified version of a synapse) between artificial neurons can transmit a signal from one to another. The artificial neuron that receives the signal can process it and then send it to artificial neurons connected to it.

In common ANN implementations, the signal at a connection between artificial neurons is a real number, and the **output** of each artificial neuron is calculated by a **non-linear function** of the sum of its inputs: **Activation function**. For my ANN i used the sigmoid function

$$F(A)=\frac{1}{1+e^{-A}}$$

The main reason why we use sigmoid function is because it exists between **(0 to 1).** Therefore, it is especially used for models where we have to **predict the probability** as an output. Since probability of anything exists only between the range of **0 and 1,** sigmoid is the right choice.

Artificial neurons and connections typically have a **weight** that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Artificial neurons may have a **threshold** such that only if the aggregate signal crosses that threshold the signal is sent. Typically, artificial neurons are organized in **layers**. Different layers may perform different kinds of transformations on their inputs. Signals travel from the first (input), to the last (output) layer, possibly after traversing the layers multiple times.

For my code I used an **ANN feedforward** with only a single **hidden layer**.

To train my ANN I used a **supervised learning** (i.e I give data input with which the ANN calculate an output, which can compare and correct with a data output,that we know yet, given).

The way to implement the weights was the **Backpropagation algorithm.** The backpropagation algorithm looks for the minimum of the **error function**(function made from the comparison between the expected values and the effective values) in weight space using the method of **gradient descent**.

Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. To find a local minimum of a function using gradient descent, one takes steps proportional to the *negative* of the gradient (or of the approximate gradient) of the function at the current point.

The "**backwards**" part of the name Backpropagation stems from the fact that calculation of the gradient proceeds backwards through the network, with the gradient of the final layer of weights being calculated first and the gradient of the first layer of weights being calculated last. Partial computations of the gradient from one layer are reused in the computation of the gradient for the previous layer. This backwards flow of the error information allows for efficient computation of the gradient at each layer versus the naive approach of calculating the gradient of each layer separately.

3.2 Prepare the data for the training.

The best way to train a supervised ANN is to divide the Dataset in two part: trainset and test set. This was the first step to write my code, preparing the Data to train the neural net and then to test it.

Dealing with eleven list of 404 values in input and eleven list of 303 values in output, I covert this list in **four matrices**. Two for the **training** : a matrix in input of 880 rows and 4 columns (% healthy,% carrier,%sick,sick-treshold) and a matrix for the output of 880 rows and 3 columns(% healthy,% carrier,%sick). The other two for the **test**: a matrix in input of 231 rows and 4 colomns(% healthy,% carrier,%sick,sick-treshold) and a matrix for the output of 231 rows and 3 columns(% healthy,% carrier,%sick).

These are the functions, present in the emacs file named **DataMatrix.py**, used to prepare the Data for the test and the train (only for the input, for the output is the same).

```
def trainset_X(dataX,dataX1,dataX2,dataX3,dataX4,dataX5,dataX6,dataX7,dataX8,dataX9,dataX10):
    X = numpy.matrix(dataX.reshape(101,4))
    X1 = numpy.matrix(dataX1.reshape(101,4))
    X2 = numpy.matrix(dataX2.reshape(101,4))
    X3 = numpy.matrix(dataX3.reshape(101,4))
    X4 = numpy.matrix(dataX4.reshape(101,4))
    X5 = numpy.matrix(dataX5.reshape(101,4))
    X6 = numpy.matrix(dataX6.reshape(101,4))
    X7 = numpy.matrix(dataX7.reshape(101,4))
    X8 = numpy.matrix(dataX8.reshape(101,4))
    X9 = numpy.matrix(dataX9.reshape(101,4))
    X10= numpy.matrix(dataX10.reshape(101,4))
    #split training set from test set
    train_X = X[:80,:4]
    train_X1 = X1[:80,:4]
    train_X2 = X2[:80,:4]
    train_X3 = X3[:80,:4]
    train_X4 = X4[:80,:4]
    train_X5 = X5[:80,:4]
    train_X6 = X6[:80,:4]
    train_X7 = X7[:80,:4]
    train_X8 = X8[:80,:4]
    train_X9 = X9[:80,:4]
    train_X10 = X10[:80,:4]
    FinalX = numpy.concatenate((train_X,train_X1,train_X2,train_X3,train_X4,train_X5,train_X6,train_X7,train_X8,train_X9,train_X10))
    return FinalX
def testset_X(dataX,dataX1,dataX2,dataX3,dataX4,dataX5,dataX6,dataX7,dataX8,dataX9,dataX10):
    X = numpy.matrix(dataX.reshape(101,4))
    X1 = numpy.matrix(dataX1.reshape(101,4))
    X2 = numpy.matrix(dataX2.reshape(101,4))
    X3 = numpy.matrix(dataX3.reshape(101,4))
```

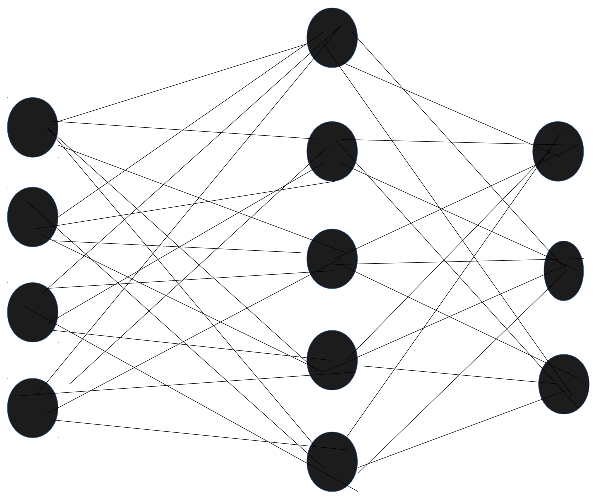
```
X4 = numpy.matrix(dataX4.reshape(101,4))
X5 = numpy.matrix(dataX5.reshape(101,4))
X6 = numpy.matrix(dataX6.reshape(101,4))
X7 = numpy.matrix(dataX7.reshape(101,4))
X8 = numpy.matrix(dataX8.reshape(101,4))
X9 = numpy.matrix(dataX9.reshape(101,4))
X10= numpy.matrix(dataX10.reshape(101,4))

test_X = X[81:,:4]
test_X1 = X1[81:,:4]
test_X2 = X2[81:,:4]
test_X3 = X3[81:,:4]
test_X4 = X4[81:,:4]
test_X5 = X5[81:,:4]
test_X6 = X6[81:,:4]
test_X7 = X7[81:,:4]
test_X8 = X8[81:,:4]
test_X9 = X9[81:,:4]
test_X10 = X10[81:,:4]
FinalTestX = numpy.concatenate(( test_X,test_X1,test_X2,test_X3,test_X4,test_X5,test_X6,test_X7,test_X8,test_X9,test_X10))
return FinalTestX
```

3.3 Training and testing

Let’s see some part of the code, that I used to train the ANN, to understand the theory explained before.

The ANN has 3 layers: the first is for the input (4 nodes) the second is the hidden layer (5 nodes) and the last is for the output(3 nodes)

Inputs layer hidden layer output layer



These are two parts of the code of the training function train_neural present in the emacs file Train_Test_NeuralNet.py

```
for k in range(nh ): # nh represents the 5 nodes of the hidden layer
    A = 0
    for i in range(nh − 2):# nh−2 represents the 3 nodes in the input layer
    # A is the sum of the outputs of the hidden layer, and wh is a matrix that represents the weights of the inputs layer nodes to the hidden layer nodes
        A = A + wh[k,i]*X[s,i]/100# here it sums the %
    A = A + wh[k,nh − 2]*B #here it sums the sick−treshold
    A = A + wh[k,nh − 1]#here it sums the treshold of the layer
    H[k] = 1/(1 + numpy.exp(−A))#this is the sigmoid function, the activation function that will be used to the output layer
for j in range (output_y):#outputy is the numer of the output nodes
    A = 0
    for k in range(nh):# nh represents the 5 nodes of the hidden layer
    # A is the sum of the outputs of the output layer, and wy is a matrix that represents the weights of the hidden layer nodes to the outputs layer nodes
        A = A + wy[j,k]*H[k]
    A = A + wy[j,nh]
    Y[j] = 1/(1 + numpy.exp(−A))
    Output_Matrix[s,j] = Y[j]#the result of the outputs became a rows in a Output matrix
```

Let’s see how work the back propagation and gradient descent:

```
#Error Output Layer
for j in range (output_y):
    Err =(0.5)*(D[s,j]/100 – Y[j])**2# Main squared error of ouput values and expected values
    DeltaY[j] = Err*Y[j]*(1 – Y[j]) #derivative of the sigmoid
#Error hidden layer
#Thanks to the Backpropagation, we can find the error of the hidden layer outputs starting from the error of the output layer outputs
for k in range(nh):
    Err = 0
    for j in range(output_y):
        Err = Err + DeltaY[j]*wy[j,k]
    DeltaH[k] = Err*H[k]*(1 - H[k])
#modify weights of output layer
for j in range(output_y):
    for k in range(nh):
        c[j,k] = wy[j,k]
#Here start the gradient descent, the steps rate to find the minimum is Eps
        wy[j,k] = wy[j,k] + Eps*DeltaY[j]*H[k] + m*d[j,k] #this last value is the Momentum(will be explain later)
        d[j,k] = c[j,k] - wy[j,k]
    c[j,nh] = wy[j,nh ]
    wy[j,nh] = wy[j,nh] + Eps*DeltaY[j] + m*d[j,k]
    d[j,nh] = c[j,nh] - wy[j,nh]
#modify weights hidden layer
for k in range(input_x-1):
    for i in range(nh - 2):
        a[k,i] = wh[k,i]
        wh[k,i] = wh[k,i] + Eps*DeltaH[k]*X[s,i]/100 + m*b[k,i]
        b[k,i] = a[k,i] -wh[k,i]
    a[k,nh-2] = wh[k,nh-2]
    wh[k,nh-2]=wh[k,nh-2] +Eps*DeltaH[k]*B + m*b[k,nh-2]
    a[k,nh-1] = wh[k,nh-1]
    wh[k,nh-1] = wh[k,nh-1] + Eps*DeltaH[k] + m*b[k,nh-1]
b[k,nh-1] = a[k,nh-1] - wh[k,nh-1]
```

In neural networks, we use gradient descent optimization algorithm to minimize the error function to reach a global minimum. However in real cases the error surface is more complex, may comprise of several local minimum. In this case, you can easily get stuck in a local minimum and the algorithm may think you reached the global minimum leading to sub-optimal results. To avoid this situation, we use a **momentum** term in the objective function, which is a value between 0 and 1 that increases the size of the steps taken towards the minimum by trying to jump from a local minimum. If the momentum term is large then the learning rate should be kept smaller. A large value of momentum also means that the convergence will happen fast. But if both the momentum and learning rate are kept at large values, then you might skip the minimum with a huge step. A small value of momentum cannot reliably avoid local minimum, and can also slow down the training of the system. Momentum also helps in smoothing out the variations, if the gradient keeps changing direction.

To control the train, the code shows the **maximum error** between the outputs and the expected outputs, after every cycle of **train**. When the error is quite small,we can stop the train and start the **test**. This function is always present in the emacs file Train_Test_NeuralNet.py

```
def ErrTrain(FinalY,Output_matrix):
    ErrMaxs = FinalY/100 - Output_matrix
    M = 0.
    for i in range(ErrMaxs.shape[0]):
        for j in range(ErrMaxs.shape[1]):
            if abs(ErrMaxs[i,j]) > M:
                M = abs(ErrMaxs[i,j])
return M
```

The code of the test function is equal to the train function but without the code part of the implementation of the weights. And also this present an error function to control the reliability of the ANN.

3.4 Using the ANN

Once that the train and the test parts are over, we can use the ANN ,with the weights update, with new values of which we don’t know the outputs. The code of the ANN is in the emacs-file ANN.py and it is written to take 4 values (%healthy,% carrier, %sick, sick-treshold) in input, and gives 3 values in output (%healthy,% carrier, %sick) after 20 generations.

4 NetLogo extension to Python

To operate the ANN, write in Python, in the NetLogo code, it was necessary downloading the extension from the github link:
https://github.com/qiemem/PythonExtension

Here there is also some information about how it works and some demos as example.

If you want some easy instruction to download it, you can find at page 19 of : http://terna.to.it/econophysics18/cmap/Notes&Links17-18.pages.pdf

Thanks to this extension I could used my ANN in NetLogo. Let’s see how this works.

We can see in the right part of the NetLogo interface code, there are other buttons. These are used for the Python part of the code. The first button to press is data_for_the_neural_net.

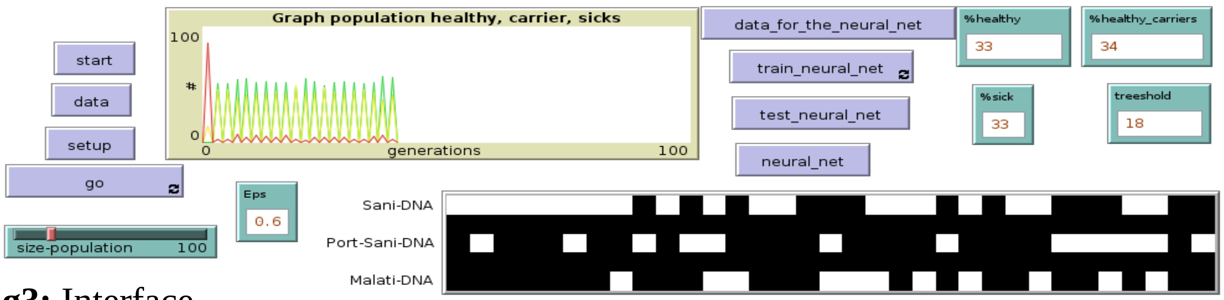


Fig3: Interface

The first part of my NetLogo code, in which I use Python’s extension, is this , and it prepares the data for the train.

```
1. to data_for_the_neural_net
2.   py:setup py:python3 #This is the command that open the python3 session
   # to interact with Python session i have to use the command py:run and py:set
3.  (py:run
4.    "import numpy"
5.    "import matplotlib"
```

6. "import DataMatrix") #this last is my Python script, in which there are many function to prepare the Data for the train
7. py:set "dataX" X
8. py:set "dataX1" X1
9. py:set "dataX2" X2
10. py:set "dataX3" X3
11. py:set "dataX4" X4
12. py:set "dataX5" X5
13. py:set "dataX6" X6
14. py:set "dataX7" X7
15. py:set "dataX8" X8
16. py:set "dataX9" X9
17. py:set "dataX10" X10
18. py:set "dataY" Y
19. py:set "dataY1" Y1
20. py:set "dataY2" Y2
21. py:set "dataY3" Y3
22. py:set "dataY4" Y4
23. py:set "dataY5" Y5
24. py:set "dataY6" Y6
25. py:set "dataY7" Y7
26. py:set "dataY8" Y8
27. py:set "dataY9" Y9
28. py:set "dataY10" Y10
29. py:set "wh" wh
30. py:set "wy" wy

#As a library, to use the function on my Python script , I have to call the function using <namescript>.function

31. (py:run
32. "FinalX = DataMatrix.trainset_X(numpy.array(dataX), numpy.array(dataX1), numpy.array(dataX2), numpy.array(dataX3), numpy.array(dataX4) , numpy.array(dataX5),

```
numpy.array(dataX6), numpy.array(dataX7), numpy.array(dataX8), numpy.array(dataX9), numpy.array(dataX10))"
"FinalY = DataMatrix.trainset_Y( numpy.array(dataY),numpy.array(dataY1),numpy.array(dataY2),numpy.array(dataY3),numpy.array(dataY4),numpy.array(dataY5),numpy.array(dataY6),numpy.array(dataY7),numpy.array(dataY8),numpy.array(dataY9),numpy.array(dataY10))"
"FinalTestX = DataMatrix.testset_X( numpy.array(dataX), numpy.array(dataX1), numpy.array(dataX2), numpy.array(dataX3), numpy.array(dataX4) , numpy.array(dataX5), numpy.array(dataX6), numpy.array(dataX7), numpy.array(dataX8), numpy.array(dataX9), numpy.array(dataX10))"
"FinalTestY = DataMatrix.testset_Y( numpy.array(dataY),numpy.array(dataY1),numpy.array(dataY2),numpy.array(dataY3),numpy.array(dataY4),numpy.array(dataY5),numpy.array(dataY6),numpy.array(dataY7),numpy.array(dataY8),numpy.array(dataY9),numpy.array(dataY10))"
 "wh = numpy.matrix(numpy.random.random((5,5)).reshape(5,5))"
 "wy = numpy.matrix(numpy.random.random((3,6)).reshape(3,6))"
 "wh = wh/10"
 "wy = wy/10"
)
end
```

The second button to press is train_neural_net, this calls the two function train_neural and ErrTrain of the Python script **Train_Test_NeuralNet.py** and this is the code:

```
to train_neural_net
  (py:run
   "import numpy"
   "import matplotlib"
   "import Train_Test_NeuralNet")
```

```
 let M 0
 py:set "Output_Matrix" Output_Matrix
 py:set "M" M
 py:set "Eps" Eps
(py:run
  "Output_Matrix = Train_Test_NeuralNet.train_neural(FinalX,FinalY,wh,wy,Eps) "
  "M = Train_Test_NeuralNet.ErrTrain(FinalY,Output_Matrix)")
type " the error in sample is =" show py:runresult "M"
end
```

This button is set to show the max error of the ANN every cycle. When the error is quite small, we can stop and press the button test_neural_net ,which uses the function test_neural and ErrTest of the Python script **Train_Test_NeuralNet.py**

```
to test_neural_net
let N 100
py:set "Output_Matrix_T" Output_Matrix_T
py:set "N" N
(py:run
  "import numpy"
  "import matplotlib"
  "Output_Matrix_T = Train_Test_NeuralNet.test_neural (FinalTestX,wh,wy)"
  "N = Train_Test_NeuralNet.ErrTest (FinalTestY,Output_Matrix_T)")
type " The error out of sample is =" show py:runresult "N"
end
```

When the ANN is ready, we can finally choose 4 data in inputs (% healthy,% carrier,%sick,sick-treshold) and put them in the right boxes, press this time the button neural_net and see the outputs of the ANN.

The function used by the button is ANN of the Python script **ANN.py** and this is the code:

```
to neural_net
let Output 1
let Xp []
```

```
set Xp lput  %healthy Xp
set Xp lput %healthy_carriers Xp
set Xp lput %sick Xp
let strh treeshold  ; range of sick-treeshold is 1 to 12
py:set "Xp" Xp
py:set "Output" Output
py:set "strh" strh

(py:run
  "import numpy"
  "import matplotlib"
  "import ANN"
  "Output =  ANN.ANN(numpy.array(Xp),wh,wy,strh)"
  )
show py:runresult "Output"
end
```

5 Results

Before using the ANN we have to be sure that the train ended well. This means that the error in sample (that will appear during the train in the command center) and the error out of sample (that will appear during the test in the command center) must be around a values of 0,15.

To see the result of the ANN, I stopped the train with an error in sample of 0,1552 and an error out of sample of 0,1720. I analyzed the output of the ANN ,by choosing 3 inputs of population(%healthy,%carrier,%sick), with particular focus on how it changed for different sick-treeshold. Let's see the results

Input (%healthy, %carrier, %sicks)	Output ANN Sick-treshold 9	Output ANN Sick-treshold 12	Output ANN Sick-treshold 15	Output ANN Sick-treshold 18	Output ANN Sick-treshold 20
(33,34,33)	(78,26,0)	(31,56,9)	(9,55,40)	(1,20,79)	(0,11,89)
(66,33,1)	(79,25,0)	(32,55,9)	(9,55,38)	(1,20,79)	(0,11,89)
(1,33,66)	(78,27,0)	(30,56,11)	(8,55,39)	(1,21,78)	(0,11,89)
(10,80,10)	(77,27,0)	(31,56,10)	(8,56,39)	(1,20,78)	(1,11,88)
(10,10,80)	(77,27,0)	(30,56,10)	(8,55,40)	(1,19,78)	(0,11,89)
(80,10,10)	(78,26,0)	(31,55,11)	(8,56,39)	(1,20,79)	(0,11,88)

First of all we can observe that the sum of the output is not always 100, this is justified by the fact that the output of the neural net has an error.

What the output of the ANN tells to us is clear by the fact that, for **different inputs,** the **output of the ANN is almost the same for the same sick-treshold.** For the ANN, if the danger of the disease has this particular value, almost **regardless of the initial population**, the **forecast will be it.**

The Data given to the ANN for the train are about different diseases: from a **common disease**, with a normal forecast power(minimum sick-treshold), to a **dangerous disease**(maximum sick-treshold), with a high forecast power. So we can observe, by the output of the ANN, that it is necessary **paying particular attention to the diseases falling within the sick-treeshold higher or equals of 15**, because they have a stronger forecast.

Succeeding in approximating the dangerous of the forecast of a disease in a single value like the sick-treeshold, thanks to an **ANN** like this we could **catalog diseases potentially dangerous**, in terms of forecast, so that focusing the **economic forces** on a risk control plan to **prevent the spread.**

5.1 And now?

Independently by the results, there is another aim that this article wants to show.

It shows the potential of the use of a simulation model, to create a dataset, to train a machine (supervised) learning algorithm. Usually, for the supervised learning, are used real (big) dataset formed over the years, i.e. about subjects that are already known. But what it can do with a dataset about a subject partially unknown (small dataset)?

The use of a simulation model could represent a dataset of a disease known, but most important can be used to create a (theoretical) dataset about a new dangerous disease.