Final project, ME 249:

Proposal: the evolutive game:

Amaury de Guillebon

Table of contents :

Introduction:

Since the appearance of COVID 19 more than 2 years ago, our habits have changed and the debate on the question of collective immunity has become an unavoidable issue for today's politicians. With this project, I propose to **code an evolutive game simulating the basic rules of epidemiology** and **an artificial intelligence** that should be able **to predict the outcome** of this game.

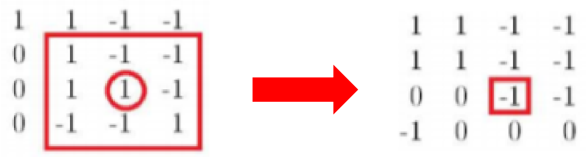
Visualization of an epidemic:

Consider a grid randomly filled with -0, 0 and 1. These numbers represent the state of a population at time t, when it realizes that there is an epidemic. -1 corresponds to infected, 0 to neutral and 1 to immune.

We sum a square with its 8 neighboring squares and distinguish 3 cases:

- If the sum is <0 then this cell becomes a -1.
- If the sum is =0 then this cell becomes a 0.
- If the sum is >0 then this cell becomes a 1.
- Optional: -1 cannot switch to 1 in one step and 1 cannot switch to -1 in one step.

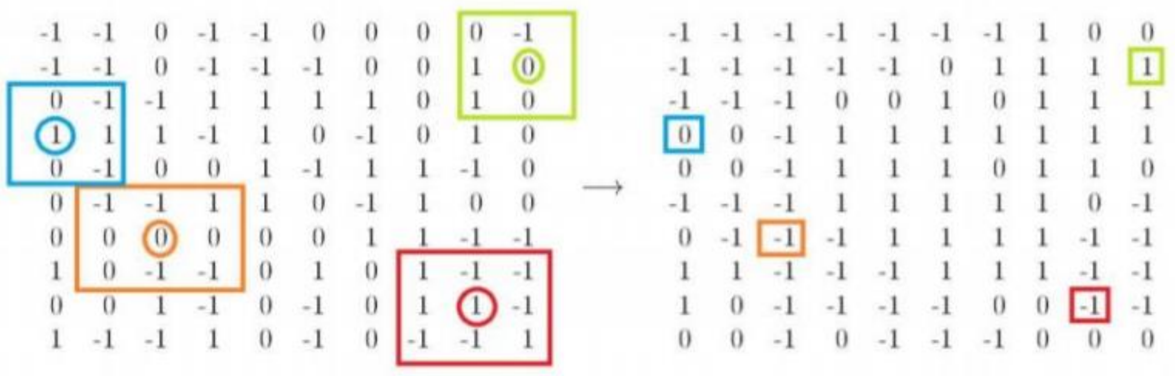
<u>Example:</u>



Here the sum is: sum = 1-1-1+1+1-1-1-1-1+1 = -1
The circled square becomes a -1.

This operation is repeated on all the squares and a new grid is obtained which corresponds to the evolution of the previous grid. We should be careful with the sides and the corners which are special cases.

<u>Example:</u>



Once all the boxes were updated, we performed a step that can be likened to one epidemic day. Therefore, we will repeat these steps as much as possible until we obtain a final stable grid that will correspond to the final stage of the epidemic, which can take several days, months or years.

Objectives of the project:

- Implement a convolution network allowing from any grid to predict the state at the next step, then a convolution network allowing to predict the final state.
- Understand and adapt the different functionalities offered by a network in order to optimize the result.
- Interpret the results and explain how this information can be useful in predicting the real evolution of an epidemic.

I) Creation of the data:

The goal here is to translate the basic rules of the evolution of an epidemic into the evolutive game presented in the introduction.

A Step 1: the libraries

Before writing any code, it is necessary to import the libraries that can help us to code. Of course, the choices of which tools to use have been made during the realization of the project and is not due to coincidence.

```python
import matplotlib.pyplot as plt
import numpy as np
import math as ma
import numpy.random as rd
import sklearn.neural_network as nn #not used
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, SeparableConv2D
from keras.layers import Flatten, Dense #not used
from keras.applications.vgg16 import VGG16 #not used
from keras.preprocessing.image import load_img, img_to_array
from keras.applications.vgg16 import preprocess_input #not used
from numpy import zeros, newaxis
import statistics
from statistics import mean
```

figure 1.

Due to the different choices made during the coding, some libraries will not be used in the final version of the code.

B Step 2: le original grids:

Now we have to create a function able to generate a grid **randomly filled with -1, 0 and 1**, according to the instructions presented in the introduction.

As a reminder:

- -1 = infected.
- 0 = at risk.
- 1 = safe or immuned.

To do so we create the following function called “init”:

```
def init (N): # we initialize an array randomly filled with -1; 0 and 1
  a=0
  T=np.zeros ((N,N)) # creation of a table of dimension 2 and size NxN
  for i in range (N):
    for j in range (N):
      a= rd.randint (-1,2) # we fill this table with a random value -1, 0 or 1
      T[i,j]=a
  return(T)
```

figure 2.

This function creates an array of size N x N filled with zeros, and then change all the zeros into a randomized value between -1, 0, 1 thank to the Random library.

By using this function, we create a grid representing what could be an epidemic at day 1.

Since a single epidemic would not be a sufficient set of data, we then create a function that creates **a list of these starting grids**. This way we will have a starting list that will serve as a reference for the whole project (it is called L0).

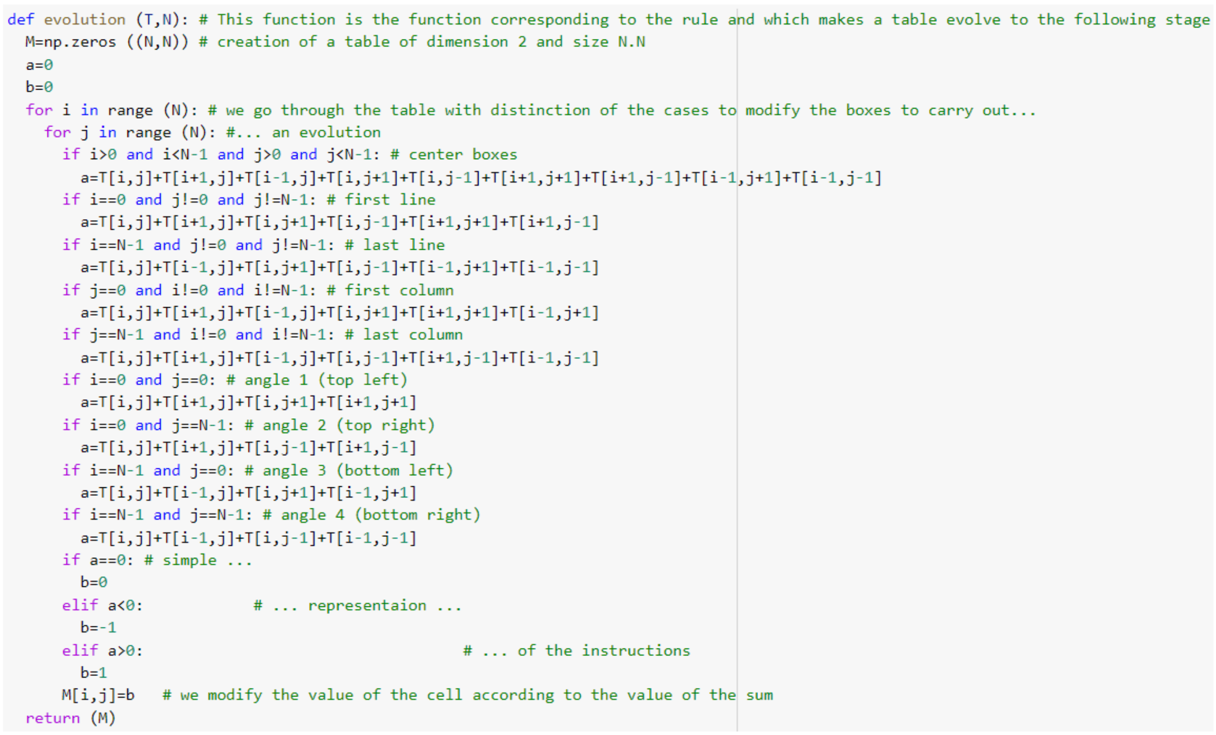
```
def listeT0 (N): # The purpose of this function is to create a list of K start tables.
  L0=[]
  for k in range (1000): # Here the list will be composed of K = 1000 tables
    L0.append(init(N))  # These tables will conform to the previous init function
  return(L0)

L0= listeT0 (N) # We call this list of starting tables L0 (important for the continuation)
print(len(L0))
```

figure 4.

In this function we can change the range of k depending on the number of data we want. Of course this choice will influence the accuracy of our neural network later.

C Step 3: the evolution of the grids:

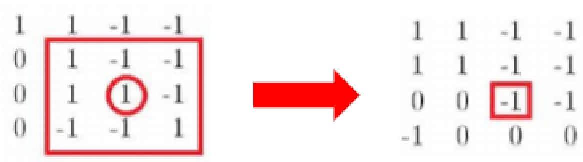
We then code a function that **makes a grid evolve to the next stage** according to the rule presented in the introduction. Great care is taken to treat the special cases of the sides and corners of the grid.



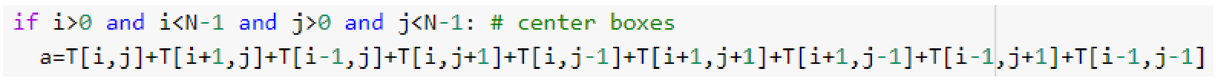
```
def evolution (T,N): # This function is the function corresponding to the rule and which makes a table evolve to the following stage
  M=np.zeros ((N,N)) # creation of a table of dimension 2 and size N.N
  a=0
  b=0
  for i in range (N): # we go through the table with distinction of the cases to modify the boxes to carry out...
    for j in range (N): #... an evolution
      if i>0 and i<N-1 and j>0 and j<N-1: # center boxes
        a=T[i,j]+T[i+1,j]+T[i-1,j]+T[i,j+1]+T[i,j-1]+T[i+1,j+1]+T[i+1,j-1]+T[i-1,j+1]+T[i-1,j-1]
      if i==0 and j!=0 and j!=N-1: # first line
        a=T[i,j]+T[i+1,j]+T[i,j+1]+T[i,j-1]+T[i+1,j+1]+T[i+1,j-1]
      if i==N-1 and j!=0 and j!=N-1: # last line
        a=T[i,j]+T[i-1,j]+T[i,j+1]+T[i,j-1]+T[i-1,j+1]+T[i-1,j-1]
      if j==0 and i!=0 and i!=N-1: # first column
        a=T[i,j]+T[i+1,j]+T[i-1,j]+T[i,j+1]+T[i+1,j+1]+T[i-1,j+1]
      if j==N-1 and i!=0 and i!=N-1: # last column
        a=T[i,j]+T[i+1,j]+T[i-1,j]+T[i,j-1]+T[i+1,j-1]+T[i-1,j-1]
      if i==0 and j==0: # angle 1 (top left)
        a=T[i,j]+T[i+1,j]+T[i,j+1]+T[i+1,j+1]
      if i==0 and j==N-1: # angle 2 (top right)
        a=T[i,j]+T[i+1,j]+T[i,j-1]+T[i+1,j-1]
      if i==N-1 and j==0: # angle 3 (bottom left)
        a=T[i,j]+T[i-1,j]+T[i,j+1]+T[i-1,j+1]
      if i==N-1 and j==N-1: # angle 4 (bottom right)
        a=T[i,j]+T[i-1,j]+T[i,j-1]+T[i-1,j-1]
      if a==0: # simple ...
        b=0
      elif a<0:          # ... representaion ...
        b=-1
      elif a>0:                               # ... of the instructions
        b=1
      M[i,j]=b   # we modify the value of the cell according to the value of the sum
  return (M)
```

figure 5.

If we look at the evolution of any box of the center following the rules presented in the introduction we have:



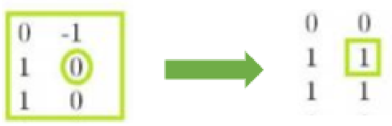
Here the sum is: sum = 1-1-1+1+1-1-1-1-1+1 = -1
The circled square becomes a -1.
Thus, to locate such a box we use an if condition, then we apply the rule presented:



```
if i>0 and i<N-1 and j>0 and j<N-1: # center boxes
  a=T[i,j]+T[i+1,j]+T[i-1,j]+T[i,j+1]+T[i,j-1]+T[i+1,j+1]+T[i+1,j-1]+T[i-1,j+1]+T[i-1,j-1]
```

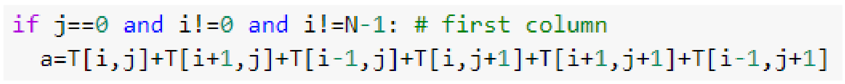
figure 6.

If we look at the evolution of any box located on the last column and following the rules presented in the introduction we have:



Here the sum is: sum = -1+0+1+1+0 = 1
The circled square becomes a 1.

Thus to locate such a box we use an if condition, then we apply the rule presented:

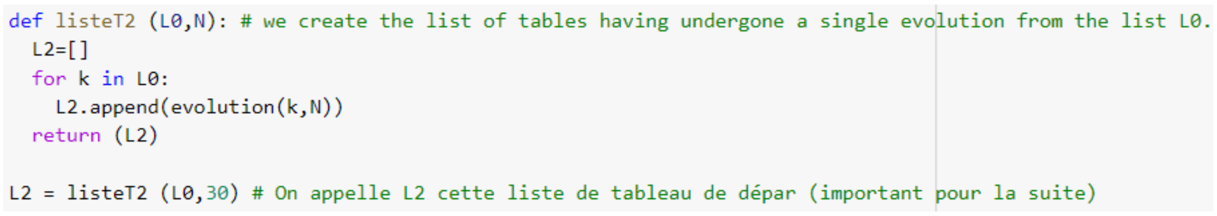


```
if j==0 and i!=0 and i!=N-1: # first column
  a=T[i,j]+T[i+1,j]+T[i-1,j]+T[i,j+1]+T[i+1,j+1]+T[i-1,j+1]
```

figure 7.

Note how the variable a is computed differently depending on the location of the box we are studying.

This function “evolution” allows us to make a grid evolve. We then implement a new function able to make evolve the already existing starting list L0 (see previous page).

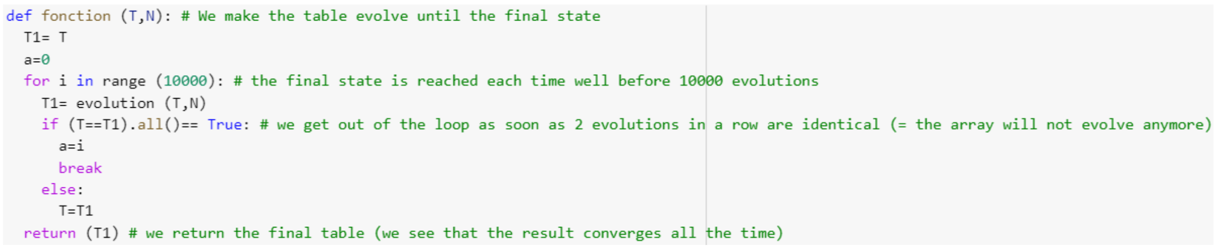


```
def listeT2 (L0,N): # we create the list of tables having undergone a single evolution from the list L0.
  L2=[]
  for k in L0:
    L2.append(evolution(k,N))
  return (L2)

L2 = listeT2 (L0,30) # On appelle L2 cette liste de tableau de dépar (important pour la suite)
```

figure 8.

D Step 4: end of the game:

The only thing missing to finish the game is a function that **makes the grids evolve until they stabilize.** This can translate to an epidemic evolving until the end of one wave.

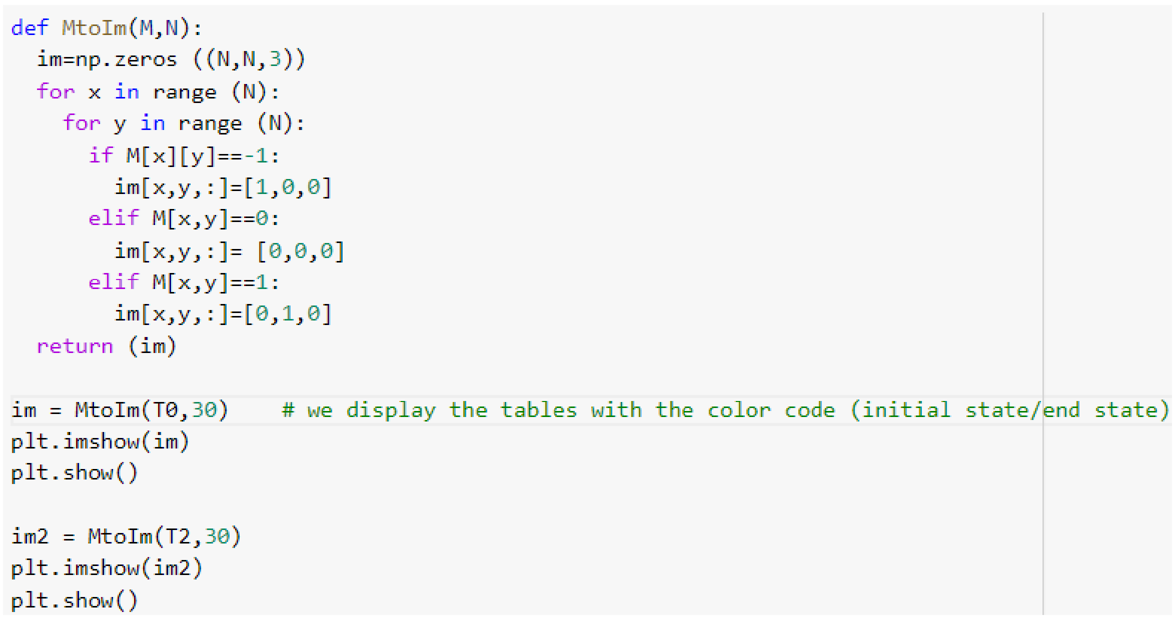


```
def fonction (T,N): # We make the table evolve until the final state
  T1= T
  a=0
  for i in range (10000): # the final state is reached each time well before 10000 evolutions
    T1= evolution (T,N)
    if (T==T1).all()== True: # we get out of the loop as soon as 2 evolutions in a row are identical (= the array will not evolve anymore)
      a=i
      break
    else:
      T=T1
  return (T1) # we return the final table (we see that the result converges all the time)
```

figure 9.

In order to verify the correct functioning of the game and its supposed stability, a code is created to display the grids in question. A visual approach allows a clear comparison.

Here is the code allowing a clear visualization:



```
def MtoIm(M,N):
  im=np.zeros ((N,N,3))
  for x in range (N):
    for y in range (N):
      if M[x][y]==-1:
        im[x,y,:]=[1,0,0]
      elif M[x,y]==0:
        im[x,y,:]= [0,0,0]
      elif M[x,y]==1:
        im[x,y,:]=[0,1,0]
  return (im)

im = MtoIm(T0,30)    # we display the tables with the color code (initial state/end state)
plt.imshow(im)
plt.show()

im2 = MtoIm(T2,30)
plt.imshow(im2)
plt.show()
```

And here is the result for a 30x30 grid:

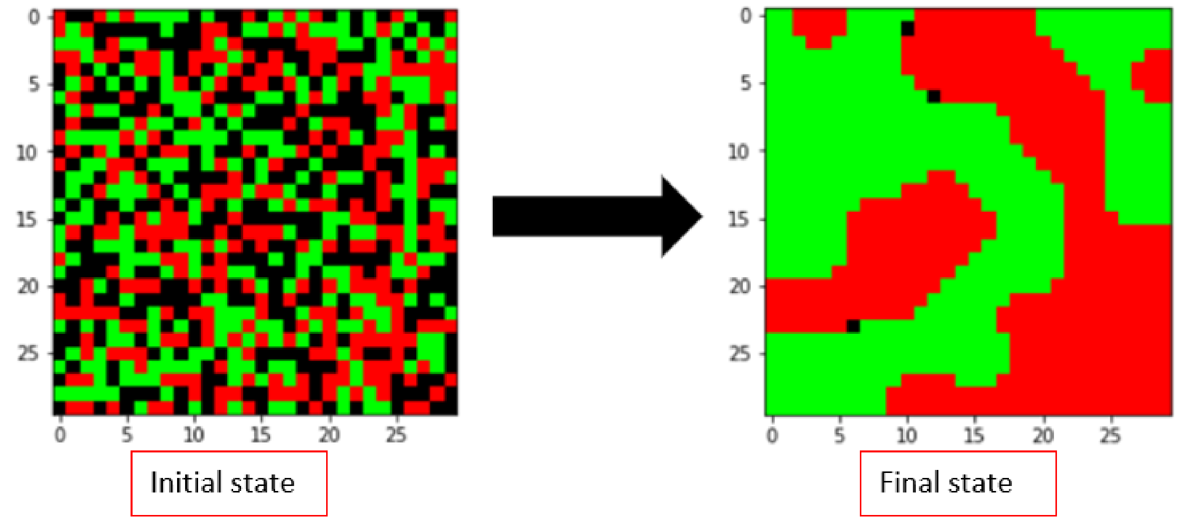


figure 10.

The color code is as follows:

- red = -1 = infected.
- black = 0 = at risk.
- Green = 1 = safe or immune.

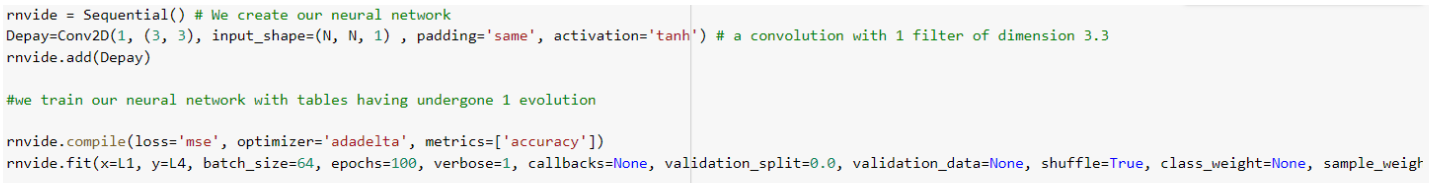
The results are quite consistent with what happens during a real epidemic: clear separation of the contaminated foci (cluster) with the so-called green foci (immune or healthy people). Therefore we can deduce that the evolution game represents accurately the evolution of an epidemic.

II) First network (a convolution to a filter), comparison of two successive states:

The goal of this part is to realize a neural network **able to predict the following state of any grid**. If we translate this idea to our problem, we are creating a neural network able to predict any following day to a given state of the epidemic.

A First try and first fail:

For this we use the functions: Sequential (), Conv2D, compile() and fit().



```
rnvide = Sequential() # We create our neural network
Depay=Conv2D(1, (3, 3), input_shape=(N, N, 1) , padding='same', activation='tanh') # a convolution with 1 filter of dimension 3.3
rnvide.add(Depay)

#we train our neural network with tables having undergone 1 evolution

rnvide.compile(loss='mse', optimizer='adadelta', metrics=['accuracy'])
rnvide.fit(x=L1, y=L4, batch_size=64, epochs=100, verbose=1, callbacks=None, validation_split=0.0, validation_data=None, shuffle=True, class_weight=None, sample_weigh
```

figure 11.

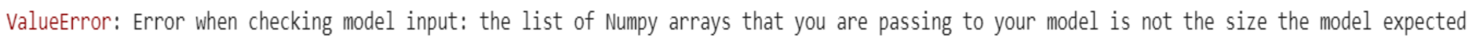
<u>Used functions:</u>

- Sequential(): we use a Sequential model to setup our neural network.
- Conv2D: we do not use a Dense layer because the **Dense Layer uses a linear operation**, meaning every output is formed by the function based on every input. Since the evolution of our data has nothing linear, we switched to **Conv2D** which is more adapted to our problem (**performs more calculations, non-linear**).
- Compile(): we use compile to configure the model for training.
- fit(): we use fit() to train the model with a fixed number of epochs.

Note that the "input shape" respects the type of data we use (arrays of dimension NxNx1), and that the activation function has been changed into 'tanh' (after several tests) instead of the classical K.relu or K.elu since the evolution of the data is not linear. The 'tanh' is in fact a hyperbolic tangent activation function, which corresponds better to our problem.

Unfortunately **the result is not conclusive**: `ValueError: Error`

The reason for this failure would be the dimensions of L0 and L2 which do not correspond to the necessary dimension for the use of such functions:



```
ValueError: Error when checking model input: the list of Numpy arrays that you are passing to your model is not the size the model expected
```

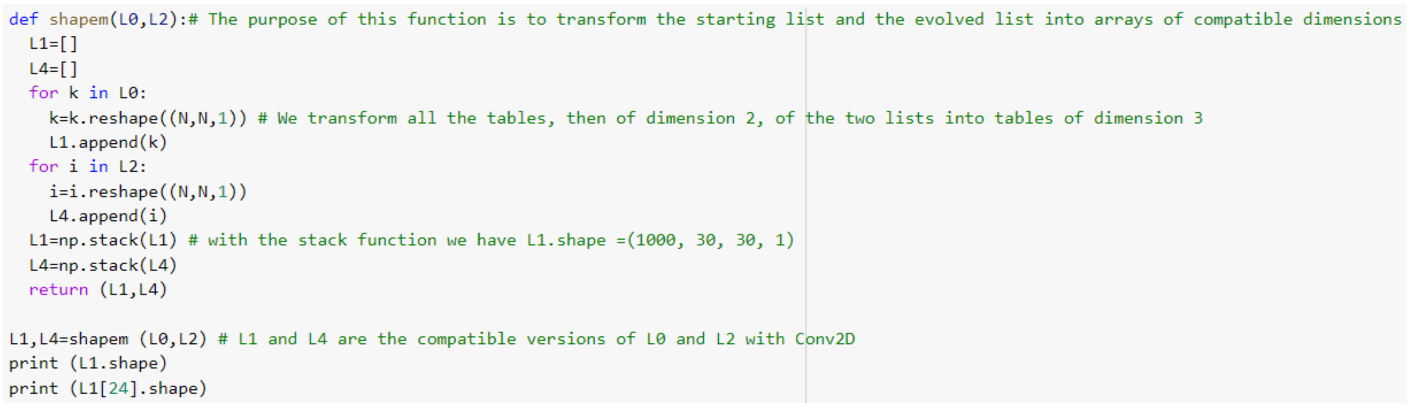
B data adaptation:

Following the previous failure, we decide to write a function whose goal would be to transform the lists L0 and L2 into two arrays of dimensions compatible with the functions used.

After various tests and researches we arrive at the following conclusion: L0 and L2 being lists of 1000 arrays of dimensions 2, it is necessary to create two arrays of dimensions (1000, 30, 30 ,1)

This corresponds roughly to an array of 3 dimensional arrays (N.N.1).

We could thus deduce the shapem function which, as its name suggests, allows us to reach the desired dimensions:

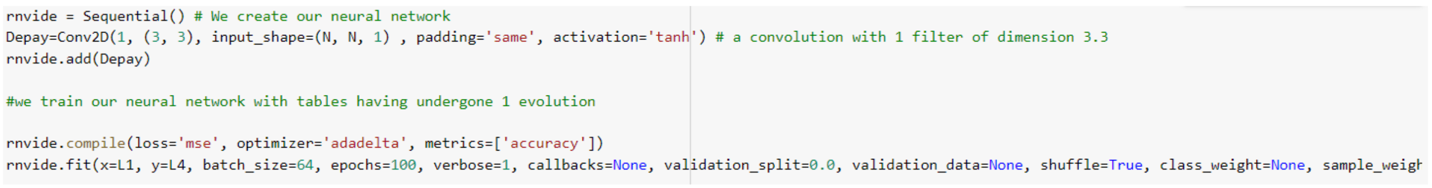


```
def shapem(L0,L2):# The purpose of this function is to transform the starting list and the evolved list into arrays of compatible dimensions
  L1=[]
  L4=[]
  for k in L0:
    k=k.reshape((N,N,1)) # We transform all the tables, then of dimension 2, of the two lists into tables of dimension 3
    L1.append(k)
  for i in L2:
    i=i.reshape((N,N,1))
    L4.append(i)
  L1=np.stack(L1) # with the stack function we have L1.shape =(1000, 30, 30, 1)
  L4=np.stack(L4)
  return (L1,L4)

L1,L4=shapem (L0,L2) # L1 and L4 are the compatible versions of L0 and L2 with Conv2D
print (L1.shape)
print (L1[24].shape)
```

figure 12.

C Second test and first success:

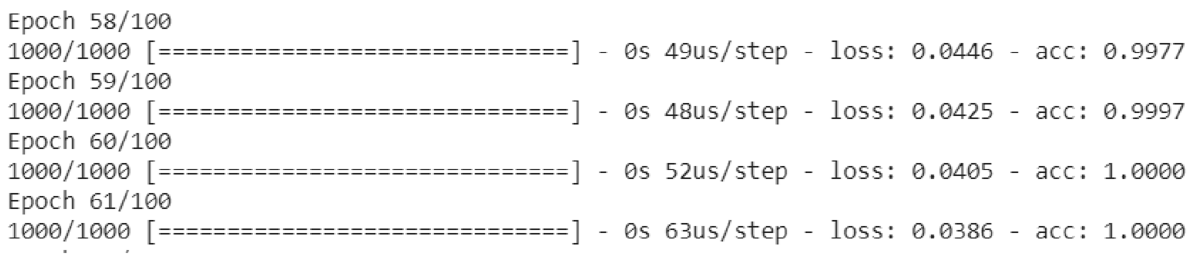
We can now create our first network, made of a single filter convolution:



```
rnvide = Sequential() # We create our neural network
Depay=Conv2D(1, (3, 3), input_shape=(N, N, 1) , padding='same', activation='tanh') # a convolution with 1 filter of dimension 3.3
rnvide.add(Depay)

#we train our neural network with tables having undergone 1 evolution

rnvide.compile(loss='mse', optimizer='adadelta', metrics=['accuracy'])
rnvide.fit(x=L1, y=L4, batch_size=64, epochs=100, verbose=1, callbacks=None, validation_split=0.0, validation_data=None, shuffle=True, class_weight=None, sample_weigh
```

figure 13.

We obtain an accuracy of 100% from the 60th epoch:



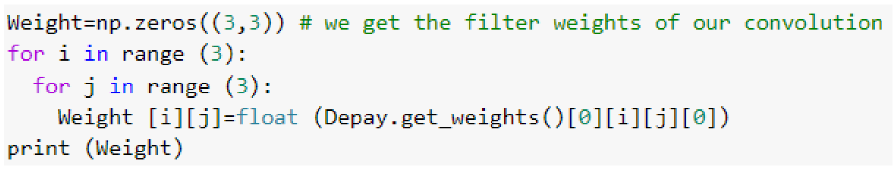
```
Epoch 58/100
1000/1000 [==============================] - 0s 49us/step - loss: 0.0446 - acc: 0.9977
Epoch 59/100
1000/1000 [==============================] - 0s 48us/step - loss: 0.0425 - acc: 0.9997
Epoch 60/100
1000/1000 [==============================] - 0s 52us/step - loss: 0.0405 - acc: 1.0000
Epoch 61/100
1000/1000 [==============================] - 0s 63us/step - loss: 0.0386 - acc: 1.0000
```

This encouraging result means that our network is now trained enough to predict 100% of the final boxes of one evolution. If we translate this to our problem, we are now able to predict with an accuracy of 100% the evolution of an epidemic from one day to another.

Note that the training data were lists of 1st days of epidemics associated to the corresponding 2nd days, which means that it was easy for the neural network to understand the rules of the game for evolving to one state to the following one.

D filter weights and interpretation:

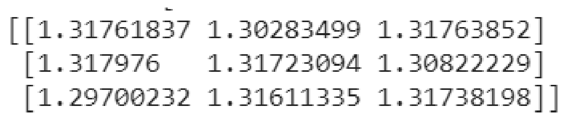
We recover the filter weights of our convolution layer with the following function:



```
Weight=np.zeros((3,3)) # we get the filter weights of our convolution
for i in range (3):
  for j in range (3):
    Weight [i][j]=float (Depay.get_weights()[0][i][j][0])
print (Weight)
```

figure 14.

The result is very encouraging since all values are relatively equialent (about 1.31):

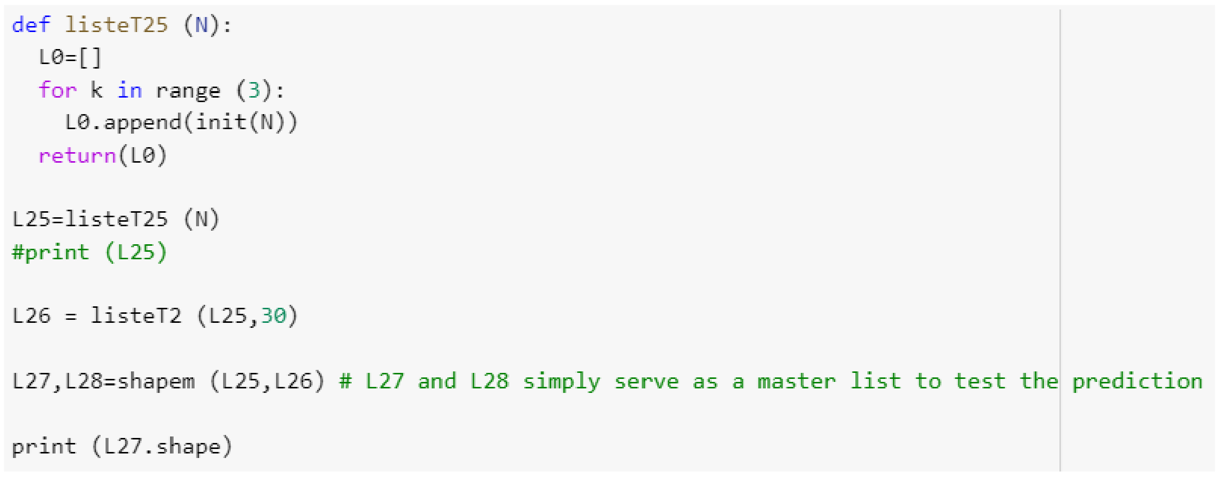


```
[[1.31761837 1.30283499 1.31763852]
 [1.317976   1.31723094 1.30822229]
 [1.29700232 1.31611335 1.31738198]]
```

E Testing of the model:

We test our model with randomized new data to check if the network really has an accuracy of 100%.

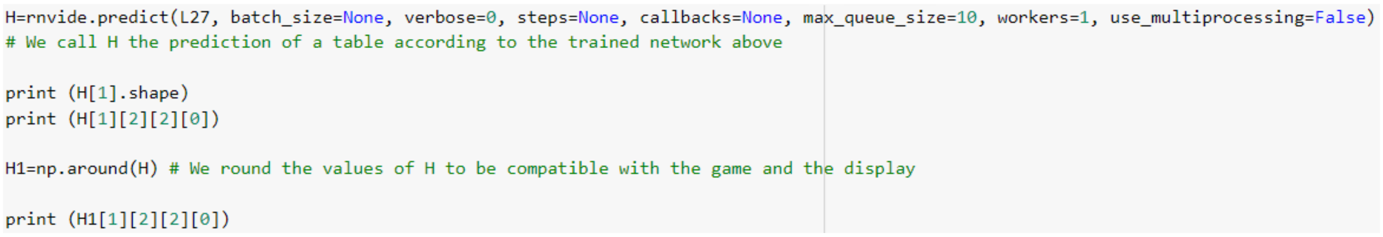
First we create a starting list and an advanced list of 3 tables:



```
def listeT25 (N):
  L0=[]
  for k in range (3):
    L0.append(init(N))
  return(L0)

L25=listeT25 (N)
#print (L25)

L26 = listeT2 (L25,30)

L27,L28=shapem (L25,L26) # L27 and L28 simply serve as a master list to test the prediction

print (L27.shape)
```

figure 15.

Then, we use our network to predict the following day of the epidemic:

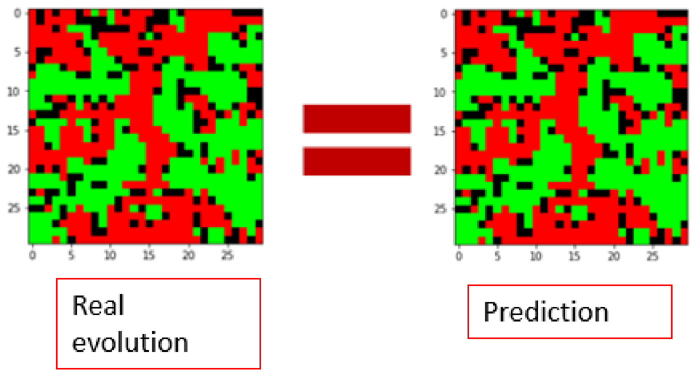


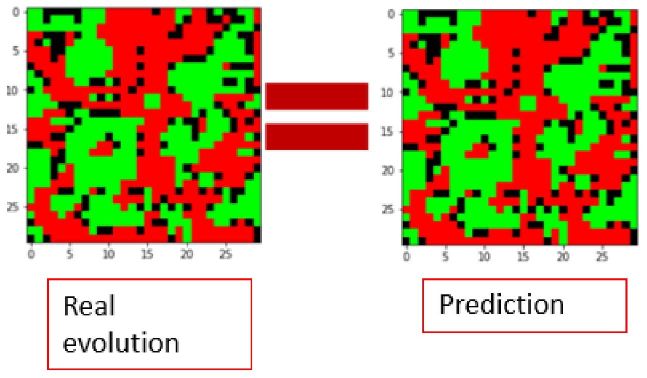
```
H=rnvide.predict(L27, batch_size=None, verbose=0, steps=None, callbacks=None, max_queue_size=10, workers=1, use_multiprocessing=False)
# We call H the prediction of a table according to the trained network above

print (H[1].shape)
print (H[1][2][2][0])

H1=np.around(H) # We round the values of H to be compatible with the game and the display

print (H1[1][2][2][0])
```

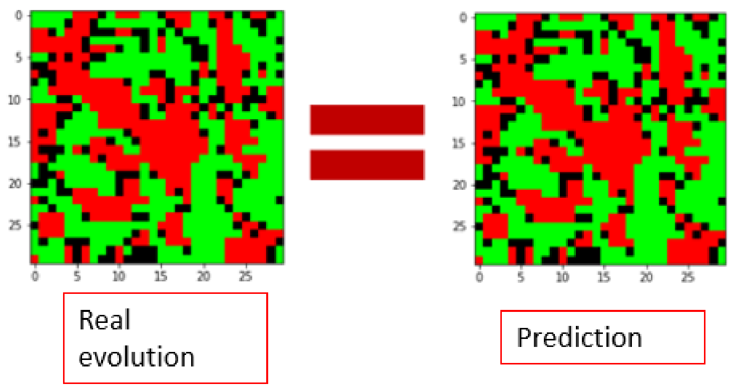
figure 16.

Note that the function np.around rounds up the values to make sure that the predicted array is mad of -1, 0, 1. Also note that the rounded values were originally very close from one of those 3 accepted values (for example, np.around rounds up 0.9875464 to 1).

We then display the evolved grids and the predicted ones in order to compare them:







<u>Conclusion:</u> The created network has a 100% accuracy because it was trained to compare 1000 tables to their direct evolutions and thus understand the rules of the game which from one state to the next are relatively easy to understand.

It remains to be seen if the accuracy would have been the same by comparing the 1000 arrays to the 1000 evolved, finalized, corresponding arrays.

If we translate this conclusion to our problem: we are perfectly able to predict the next day of an epidemic if we have enough data showing the evolution of two following days. However, nothing can confirm yet that the result would be the same if the data available was only the 1st and last day of an epidemic.

III) Second network (several convolutions to a filter), comparison of the first with last states:

The goal here is to train a network this time by comparing starting grids to their evolved (=final) and stable versions. This step is necessary since we want our AI to predict outcome of an epidemic, not only the following day.

Note that we will use the functions presented in I.D, in particular the function called "function" which allows to obtain these final states.

A Testing of our existing neural network:

So we start by creating the list of final grids corresponding to the outcome of the starting list L0:

```
def listeT20 (L0,N):  # we create an array list in the final state from L0
  L20=[]
  for k in L0:
    L20.append(fonction(k,N))
  return (L20)

L20= listeT20 (L0, N) # L20 corresponds to the end list of L0

L1,L5=shapem (L0,L20) # L1 and L5 are the compatible versions of L0 and L20 with Conv2D
```

Figure 17.

Note that if we translate this code to our problem, it represents the final state of every corresponding epidemic in the list L0.

We then re-use our existing neural network (presented in II.C), and here are the results:

```
Epoch 98/100
1000/1000 [==============================] - 0s 48us/step - loss: 0.6331 - acc: 0.4788
Epoch 99/100
1000/1000 [==============================] - 0s 57us/step - loss: 0.6331 - acc: 0.4787
Epoch 100/100
1000/1000 [==============================] - 0s 49us/step - loss: 0.6331 - acc: 0.4788
```

The accuracy stagnates at about 47.88% of these predicted boxes, which **is not enough to predict outcomes.**

B Adapted number of convolutions:

Because of the disappointing obtained with our first neural network with one convolution, we decide to add a certain number of them.

The number of evolution that a grid undergoes before reaching its stable form is variable. So we decide to **calculate the average X of the number of evolutions undergone** by these grids before stabilizing them. Logically, by adding X convolutions, this should leave about one convolution per evolution of the grids: we would then come back to a network close to the one created in part II but adapted to compare the starting grids to their final versions (not only their evolutions).

We start by creating a function that returns the number of evolutions before stabilization, then take the mean of such a list to obtain the average numbers of iterations before stabilization:

```
def listeT1000 (L0,N): # This function returns the list of the devolution number before the stabilization of the starting tables
  L1000=[]
  for k in L0:
    L1000.append(fonction2(k,N))
  return (L1000) # This list will allow us to deduce an optimal convolution number

L1000=listeT1000 (L0,N)
```

```
z = max(L1000)
y= mean (L1000)
print (z)
print (y)
x= int(y) # x corresponds to the average of the number of table evolutions
print (x)
```

The nimber of convolutions we are going to add

figure 18.

Now we can create a neural network with an adapted number of convolutions.

C Neural network with an adapted number of convolutions:

Now that we know the number of convolutions to add we can create a more efficient neural network:

```
rnvide2= Sequential() # We create a new neural network
rnvide2.add(Conv2D(1, (3, 3), input_shape=(N, N, 1) , padding='same', activation='tanh')) # First convolution
for k in range (x): # We add x convolutions
  rnvide2.add(Conv2D(1, (3, 3), padding='same', activation='tanh'))

# we train the neural network with arrays in their initial and final states

rnvide2.compile(loss='mse', optimizer='adadelta', metrics=['accuracy'])
rnvide2.fit(x=L1, y=L5, batch_size=64, epochs=300, verbose=1, callbacks=None, validation_split=0.0, validation_data=None,
```

figure 19.

The accuracy result of this network is as follows:

```
Epoch 298/300
1000/1000 [==============================] - 0s 175us/step - loss: 0.0820 - acc: 0.9555
Epoch 299/300
1000/1000 [==============================] - 0s 177us/step - loss: 0.0821 - acc: 0.9556
Epoch 300/300
1000/1000 [==============================] - 0s 186us/step - loss: 0.0817 - acc: 0.9560
```

We obtain an accuracy of a little more than 95%, which is very encouraging since following this idea means that this network is theoretically capable of predicting at least 95% of the cells.

It is still possible to improve this result by greatly increasing the number of epochs or convolutions, but then at the expense of the training time of the network.

D Testing of our new model:

Having a network with an accuracy of 95%, we then test it with a set of test data:

```
H2=rnvide2.predict(L1, batch_size=None, verbose=0, steps=None, callbacks=None, max_queue_size=10, workers=1, use_multiprocessing=False)

H3=np.around(H2) # H3 corresponds to the prediction of L1
```

figure 20.

Here is one example of the performances of our network:

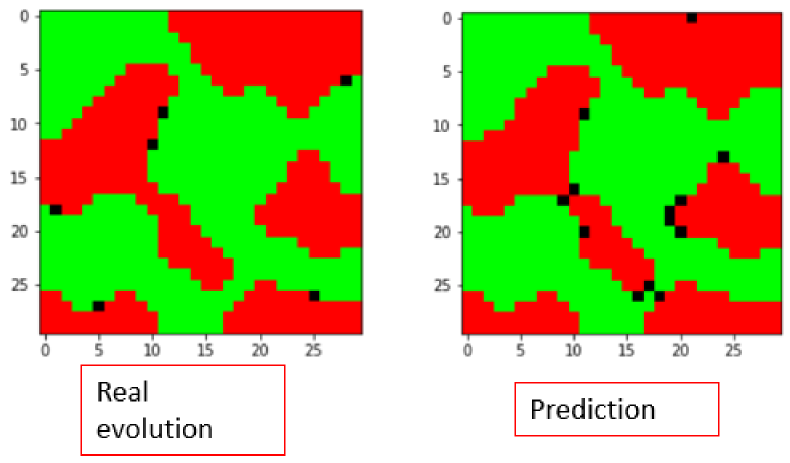


figure 21.

We can clearly see with this example that our model is able to be close to the actual result since out of the 900 boxes, less than 20 boxes are different (which makes an accuracy of 97,7% in that specific case).

If we translate this to our problem, we can now assess that **this neural network is able to predict the outcome of an epidemic, with a decent accuracy of 95%.** Therefore, if we make the assumption that an epidemic follows strictly the rules presented in the instructions,with only one input being the state of an epidemic at a given t time, this neural network can predict the stabilization state.

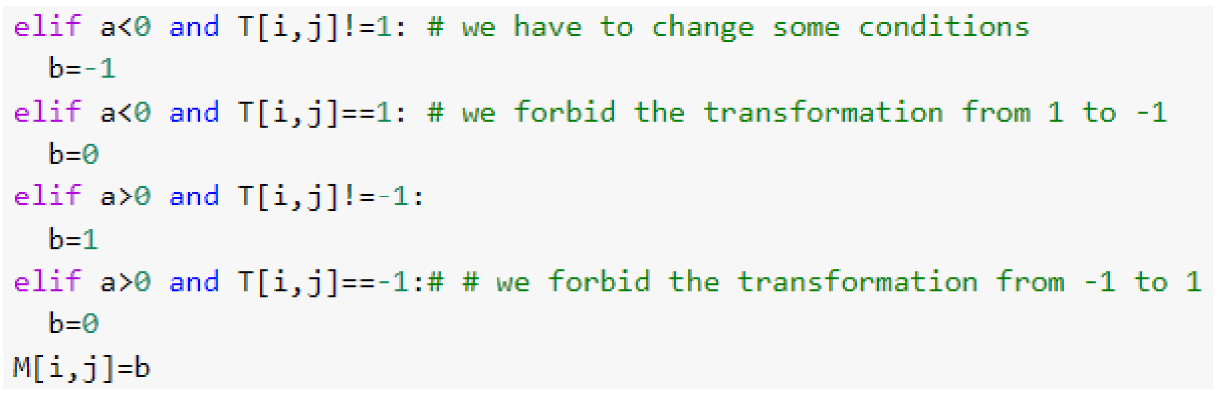
IV) Third network (several convolutions with several filters), comparison of the first with last states:

Having now a network able to predict the final grids rather well, we wonder if it is possible to increase the performances even more. Having successfully increased the number of convolutions, increasing the number of filters seems to be the next step. We therefore propose to try to create a new network of several layers of convolutions with several filters.

We also propose to **implement the optional rule**: -1 cannot evolve into +1 in the next turn, and vice versa. To change to 1, a -1 must go through the intermediate state 0. If we translate this rule to our problem, that means that someone healthy cannot get infected just like that and must go through a state where this person is “at risk” (=0). This also means that to heal from a disease, someone must go through a period of being at risk (for example still infected but without symptoms).

A New rule:

We begin by adding this new rule and for that we must modify the evolution function presented in part I).C by adding conditions prohibiting the transformation of -1 into 1 and vice versa:



```
elif a<0 and T[i,j]!=1: # we have to change some conditions
  b=-1
elif a<0 and T[i,j]==1: # we forbid the transformation from 1 to -1
  b=0
elif a>0 and T[i,j]!=-1:
  b=1
elif a>0 and T[i,j]==-1:# # we forbid the transformation from -1 to 1
  b=0
M[i,j]=b
```

figure 22.

We then decide to compare the difference between the two rules by displaying two evolutions: one according to the old rule and one according to the new one:

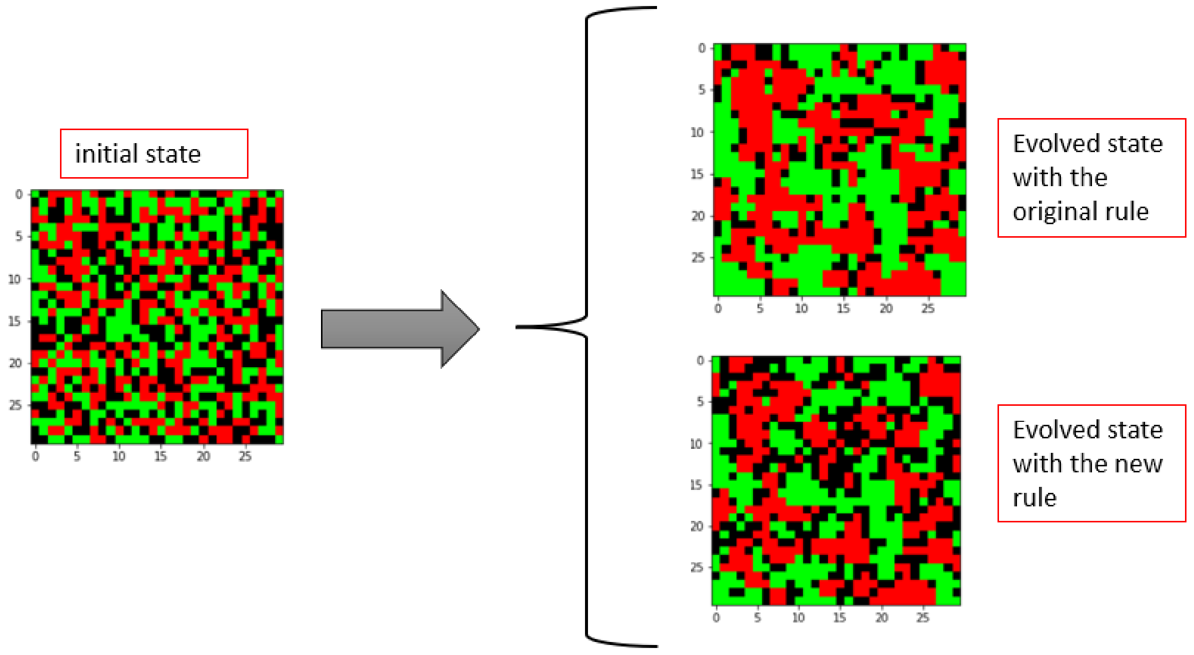
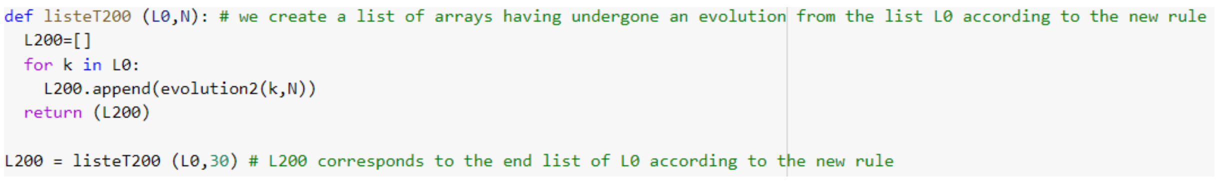


figure 23.

The two rules give different evolutions, and it goes without saying that **the new one is more complicated**. It will be more difficult for the future networks created to understand the rule and predict an outcome.

B New data after only one evolution:

Since we have new rules, **we need to update our data.** To use the same exact code but with the new evolution function. We create the list of original grids and the list of final grids:



```
def listeT200 (L0,N): # we create a list of arrays having undergone an evolution from the list L0 according to the new rule
  L200=[]
  for k in L0:
    L200.append(evolution2(k,N))
  return (L200)

L200 = listeT200 (L0,30) # L200 corresponds to the end list of L0 according to the new rule
```

figure 24.

C Training and testing of our first neural network:

We can then re-create our first network with this new rule. It will have a convolution layer with a filter, and it will be trained with the initial grids associated to their following state.

```
rnvide3 = Sequential() # We recreate our new neural network
Depay=Conv2D(1, (3, 3), input_shape=(N, N, 1) , padding='same', activation='tanh')
rnvide3.add(Depay)


rnvide3.compile(loss='mse', optimizer='adadelta', metrics=['accuracy'])
rnvide3.fit(x=L1, y=L40, batch_size=64, epochs=68, verbose=1, callbacks=None, validation_split=0.0,
```

figure 25.

The result is very mixed with an accuracy of only 85%. We recall that with the same number of epochs and similar network we had obtained an accuracy of 100% with the old rule:

```
Epoch 66/68
1000/1000 [==============================] - 0s 54us/step - loss: 0.1031 - acc: 0.8543
Epoch 67/68
1000/1000 [==============================] - 0s 60us/step - loss: 0.1030 - acc: 0.8542
Epoch 68/68
1000/1000 [==============================] - 0s 50us/step - loss: 0.1030 - acc: 0.8531
```

This is due to the higher difficulty of the new rule: it is more difficult for the network to understand.

D New data with stabilized states:

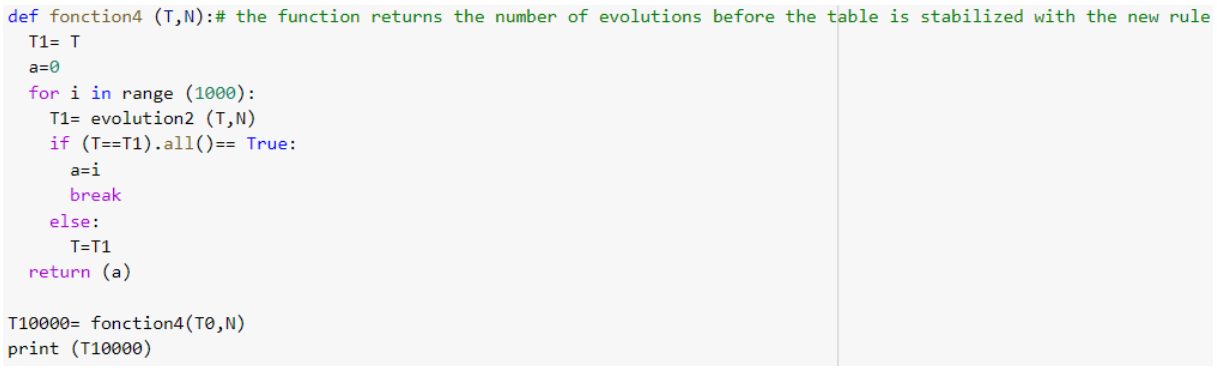
Now, according to part III), we need to create a network capable of predicting the final state of the grids. Therefore we need to create the new data for the final grids. To do so, we use the same codes but with the new evolution function:

First we code the complete evolution of a grid with the new rule:

```
def fonction3 (T,N): # This function makes the table evolve to the final state with the new rule
  T1= T
  a=0
  for i in range (10000):
    T1= evolution2 (T,N)
    if (T==T1).all()== True:
      a=i
      break
    else:
      T=T1
  return (T1)
```

figure 26.

Then the list of grids in the final state:

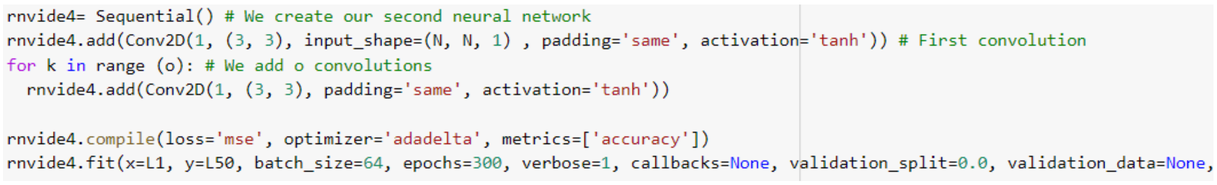


```
def fonction4 (T,N):# the function returns the number of evolutions before the table is stabilized with the new rule
  T1= T
  a=0
  for i in range (1000):
    T1= evolution2 (T,N)
    if (T==T1).all()== True:
      a=i
      break
    else:
      T=T1
  return (a)

T10000= fonction4(T0,N)
print (T10000)
```

figure 27.

E Training and testing of our second neural network:

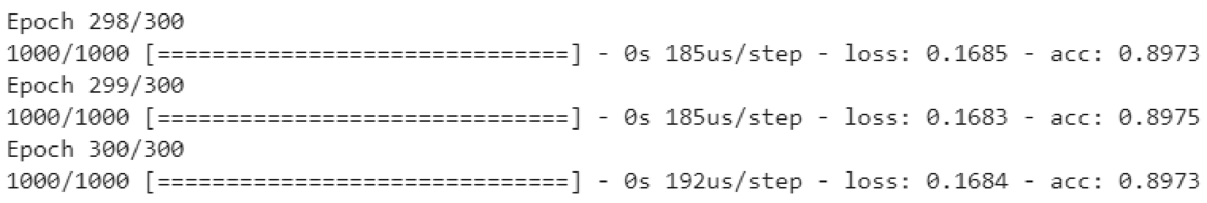
Before recreating our second neural network, we need to compute the new average number of iterations before the problem stabilizes itself. To do so we use the same functions as in part III.b, to be able to recreate our model. Only then we can train it with the new data sets.



```
rnvide4= Sequential() # We create our second neural network
rnvide4.add(Conv2D(1, (3, 3), input_shape=(N, N, 1) , padding='same', activation='tanh')) # First convolution
for k in range (o): # We add o convolutions
  rnvide4.add(Conv2D(1, (3, 3), padding='same', activation='tanh'))

rnvide4.compile(loss='mse', optimizer='adadelta', metrics=['accuracy'])
rnvide4.fit(x=L1, y=L50, batch_size=64, epochs=300, verbose=1, callbacks=None, validation_split=0.0, validation_data=None,
```

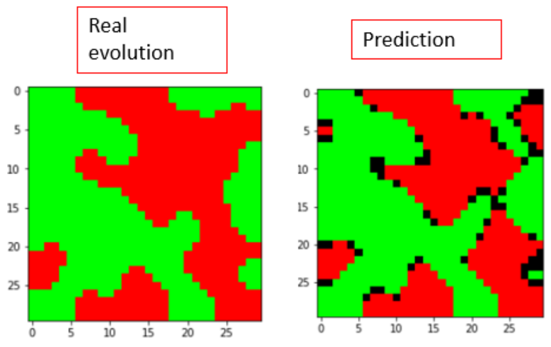
figure 28.

We obtain an accuracy of 89%. We recall that for the first rule with an equivalent network and the same number of epochs we obtained an accuracy of more than 95%. This result confirms the additional difficulty that the new rule brings as proven by this difference in accuracy:



```
Epoch 298/300
1000/1000 [==============================] - 0s 185us/step - loss: 0.1685 - acc: 0.8973
Epoch 299/300
1000/1000 [==============================] - 0s 185us/step - loss: 0.1683 - acc: 0.8975
Epoch 300/300
1000/1000 [==============================] - 0s 192us/step - loss: 0.1684 - acc: 0.8973
```

We can visualize these results by using this model to run some validation data, in a similar way than in part III.D, and here is one of the examples:

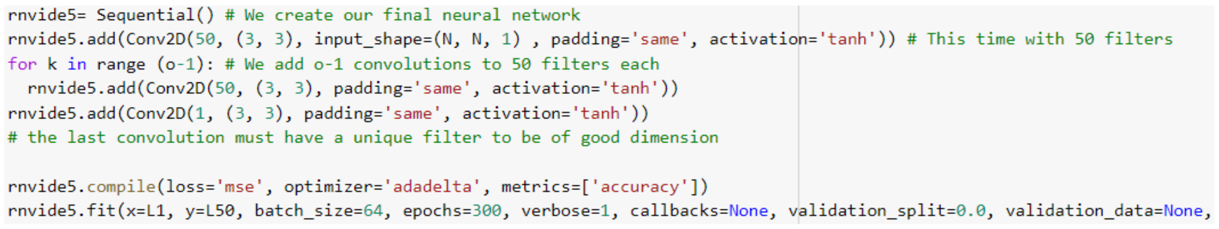


We can clearly observe that even if our prediction is correct, it is still far from perfection. This is why we need to improve our model.

E Final neural network:

As explained at the beginning of the game, we decide to create a last network, with as many convolution layers but this time more than a single filter.

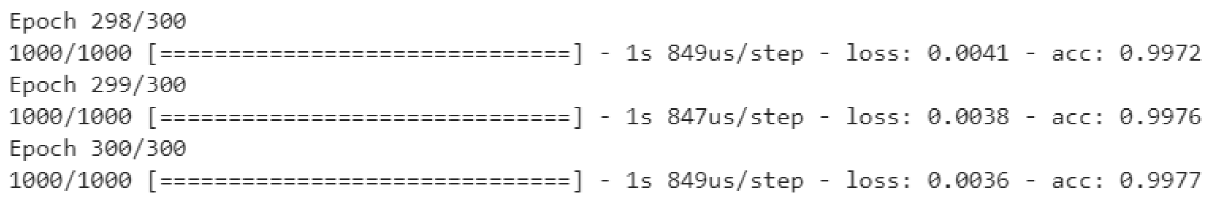
We choose arbitrarily a large number of filters (we choose 50) in order to have a good precision:



```
rnvide5= Sequential() # We create our final neural network
rnvide5.add(Conv2D(50, (3, 3), input_shape=(N, N, 1) , padding='same', activation='tanh')) # This time with 50 filters
for k in range (o-1): # We add o-1 convolutions to 50 filters each
  rnvide5.add(Conv2D(50, (3, 3), padding='same', activation='tanh'))
rnvide5.add(Conv2D(1, (3, 3), padding='same', activation='tanh'))
# the last convolution must have a unique filter to be of good dimension

rnvide5.compile(loss='mse', optimizer='adadelta', metrics=['accuracy'])
rnvide5.fit(x=L1, y=L50, batch_size=64, epochs=300, verbose=1, callbacks=None, validation_split=0.0, validation_data=None,
```

figure 29.

And the result exceeds all expectations since with this network of several convolution layers with 50 filters we obtain an accuracy of almost 100% (99.77% to be precise).



```
Epoch 298/300
1000/1000 [==============================] - 1s 849us/step - loss: 0.0041 - acc: 0.9972
Epoch 299/300
1000/1000 [==============================] - 1s 847us/step - loss: 0.0038 - acc: 0.9976
Epoch 300/300
1000/1000 [==============================] - 1s 849us/step - loss: 0.0036 - acc: 0.9977
```

We can visualize these results by using this model to run some validation data, and here is one of the examples:

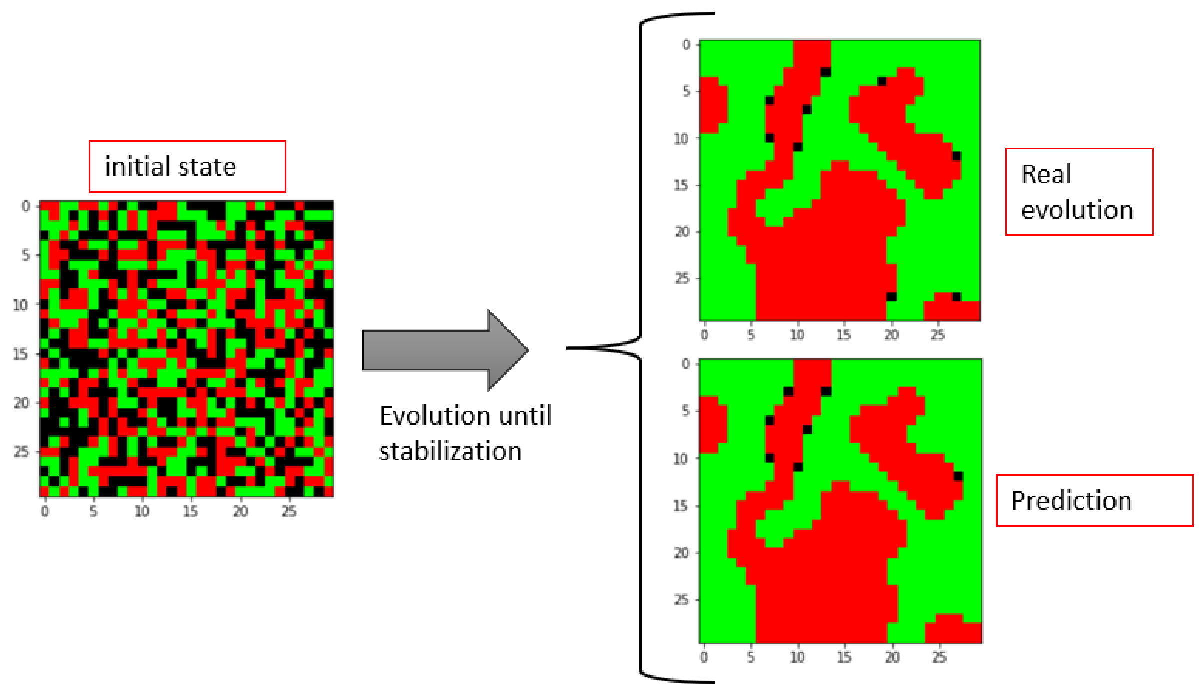


figure 29.

In the case of this particular example, only 3 of the 900 cells were not correctly predicted, which can be translated into an accuracy of 99.67%.

Apart from this case, the validation set and the accuracy of the results obtained prove that this neural network is the best performing of the three networks presented in this study.

If we translate this to our problem, we can now assess that this neural network is able to predict the outcome of an epidemic, with a very hight accuracy of 99.77%. Therefore, if we make the assumption that an epidemic follows strictly the rules presented in the instructions (in addition to the added rule), with only one input being the state of an epidemic at a given t time, this neural network can predict the stabilization state: the outcome of the give epidemic.

Conclusion :

Interpretation of results

As the questions and tests progress, we can testify to the following evolution:

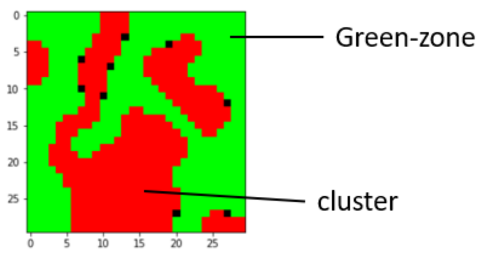
- The more complicated the rule is, the more difficult it is for the network to understand it.
- The more evolutions of a grid there are, the more difficult it is for the network to retrace it and thus to understand how the game works.
- By adding a number of convolutions, we can predict with a fairly high accuracy a complete evolution of grids.

- By adding several layers of convolutions to several filters one can predict almost perfectly the complete evolution of a grid despite a more complicated rule.

A larger number of convolution layers and filters generally increases the accuracy of a network and allows it to understand more complex mechanics.

Answer to our problem

We tried to simulate the evolution of an epidemic by dividing the population into only 3 categories: infected, at risk and healthy. The close-to-close contamination that was imposed allowed us to compare the real evolution of the epidemic with the predictions made along the way of this project. The results are quite consistent with what happens during a real epidemic: clear separation of the contaminated foci (cluster) with the so-called green foci. Therefore we can deduce that the evolution game represents accurately the evolution of an epidemic, and that the neural netork is actually able to predict quite accurately to predict the outcome.



Of course this problem is much more complex that is seems, since contamination is not solely close-to-close and that some imuned people stay imuned for the entire life etc… This model can be even more precise and take into consideration even more rules of epidemiology. This can be an improvement to do for a next study.

Appendix:

```
import matplotlib.pyplot as plt
import numpy as np
import math as ma
import numpy.random as rd
import sklearn.neural_network as nn #not used
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, SeparableConv2D
from keras.layers import Flatten, Dense #not used
from keras.applications.vgg16 import VGG16 #not used
from keras.preprocessing.image import load_img, img_to_array
from keras.applications.vgg16 import preprocess_input #not used
from numpy import zeros, newaxis
import statistics
from statistics import mean


#import of the values
N=30
```

```
def init (N): # we initialize an array randomly filled with -1; 0 and 1
  a=0
  T=np.zeros ((N,N)) # creation of a table of dimension 2 and size NxN
  for i in range (N):
    for j in range (N):
      a= rd.randint (-1,2) # we fill this table with a random value -
1, 0 or 1
      T[i,j]=a
  return(T)

T0 =init(30)
#print(T0)

def listeT0 (N): # The purpose of this function is to create a list of
K start tables.
  L0=[]
  for k in range (1000): # Here the list will be composed of K = 1000 t
ables
    L0.append(init(N))  # These tables will conform to the previous ini
t function
  return(L0)

L0= listeT0 (N) # We call this list of starting tables L0 (important fo
r the continuation)
print(len(L0))


def evolution (T,N): # This function is the function corresponding to t
he rule and which makes a table evolve to the following stage
  M=np.zeros ((N,N)) # creation of a table of dimension 2 and size N.N
  a=0
  b=0
  for i in range (N): # we go through the table with distinction of the
 cases to modify the boxes to carry out...
    for j in range (N): #... an evolution
      if i>0 and i<N-1 and j>0 and j<N-1: # center boxes
        a=T[i,j]+T[i+1,j]+T[i-1,j]+T[i,j+1]+T[i,j-
1]+T[i+1,j+1]+T[i+1,j-1]+T[i-1,j+1]+T[i-1,j-1]
      if i==0 and j!=0 and j!=N-1: # first line
        a=T[i,j]+T[i+1,j]+T[i,j+1]+T[i,j-1]+T[i+1,j+1]+T[i+1,j-1]
      if i==N-1 and j!=0 and j!=N-1: # last line
        a=T[i,j]+T[i-1,j]+T[i,j+1]+T[i,j-1]+T[i-1,j+1]+T[i-1,j-1]
      if j==0 and i!=0 and i!=N-1: # first column
        a=T[i,j]+T[i+1,j]+T[i-1,j]+T[i,j+1]+T[i+1,j+1]+T[i-1,j+1]
      if j==N-1 and i!=0 and i!=N-1: # last column
        a=T[i,j]+T[i+1,j]+T[i-1,j]+T[i,j-1]+T[i+1,j-1]+T[i-1,j-1]
      if i==0 and j==0: # angle 1 (top left)
        a=T[i,j]+T[i+1,j]+T[i,j+1]+T[i+1,j+1]
      if i==0 and j==N-1: # angle 2 (top right)
```

```
        a=T[i,j]+T[i+1,j]+T[i,j-1]+T[i+1,j-1]
      if i==N-1 and j==0: # angle 3 (bottom left)
        a=T[i,j]+T[i-1,j]+T[i,j+1]+T[i-1,j+1]
      if i==N-1 and j==N-1: # angle 4 (bottom right)
        a=T[i,j]+T[i-1,j]+T[i,j-1]+T[i-1,j-1]
      if a==0: # simple ...
        b=0
      elif a<0:            # ... representaion ...
        b=-1
      elif a>0:                                  # ... of the instruct
ions
        b=1
      M[i,j]=b   # we modify the value of the cell according to the val
ue of the sum
  return (M)

M0= evolution (T0,30)
print (T0)
print (M0)

def listeT2 (L0,N): # we create the list of tables having undergone a s
ingle evolution from the list L0.
  L2=[]
  for k in L0:
    L2.append(evolution(k,N))
  return (L2)

L2 = listeT2 (L0,30) # On appelle L2 cette liste de tableau de dépar (i
mportant pour la suite)
#print (L2)


def fonction (T,N): # We make the table evolve until the final state
  T1= T
  a=0
  for i in range (10000): # the final state is reached each time well b
efore 10000 evolutions
    T1= evolution (T,N)
    if (T==T1).all()== True: # we get out of the loop as soon as 2 evol
utions in a row are identical (= the array will not evolve anymore)
      a=i
      break
    else:
      T=T1
  return (T1) # we return the final table (we see that the result conve
rges all the time)

T2= fonction (T0,30)
#print (T0)
```

```
#print (T2)

def MtoIm(M,N):
  im=np.zeros ((N,N,3))
  for x in range (N):
    for y in range (N):
      if M[x][y]==-1:
        im[x,y,:]=[1,0,0]
      elif M[x,y]==0:
        im[x,y,:]= [0,0,0]
      elif M[x,y]==1:
        im[x,y,:]=[0,1,0]
  return (im)

im = MtoIm(T0,30)    # we display the tables with the color code (initi
al state/end state)
plt.imshow(im)
plt.show()

im2 = MtoIm(T2,30)
plt.imshow(im2)
plt.show()

def shapem(L0,L2):# The purpose of this function is to transform the st
arting list and the evolved list into arrays of compatible dimensions
  L1=[]
  L4=[]
  for k in L0:
    k=k.reshape((N,N,1)) # We transform all the tables, then of dimensi
on 2, of the two lists into tables of dimension 3
    L1.append(k)
  for i in L2:
    i=i.reshape((N,N,1))
    L4.append(i)
  L1=np.stack(L1) # with the stack function we have L1.shape =(1000, 30
, 30, 1)
  L4=np.stack(L4)
  return (L1,L4)

L1,L4=shapem (L0,L2) # L1 and L4 are the compatible versions of L0 and
L2 with Conv2D
print (L1.shape)
print (L1[24].shape)


rnvide = Sequential() # We create our neural network
Depay=Conv2D(1, (3, 3), input_shape=(N, N, 1) , padding='same', activat
ion='tanh') # a convolution with 1 filter of dimension 3.3
rnvide.add(Depay)
```

```
#we train our neural network with tables having undergone 1 evolution

rnvide.compile(loss='mse', optimizer='adadelta', metrics=['accuracy'])
rnvide.fit(x=L1, y=L4, batch_size=64, epochs=100, verbose=1, callbacks=
None, validation_split=0.0, validation_data=None, shuffle=True, class_w
eight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=None,
validation_steps=None, validation_freq=1, max_queue_size=10, workers=1,
 use_multiprocessing=False)


Weight=np.zeros((3,3)) # we get the filter weights of our convolution
for i in range (3):
  for j in range (3):
    Weight [i][j]=float (Depay.get_weights()[0][i][j][0])
print (Weight)


def listeT25 (N):
  L0=[]
  for k in range (3):
    L0.append(init(N))
  return(L0)

L25=listeT25 (N)
#print (L25)

L26 = listeT2 (L25,30)

L27,L28=shapem (L25,L26) # L27 and L28 simply serve as a master list to
 test the prediction

print (L27.shape)


H=rnvide.predict(L27, batch_size=None, verbose=0, steps=None, callbacks
=None, max_queue_size=10, workers=1, use_multiprocessing=False)
# We call H the prediction of a table according to the trained network
above

print (H[1].shape)
print (H[1][2][2][0])

H1=np.around(H) # We round the values of H to be compatible with the ga
me and the display

print (H1[1][2][2][0])
```

```
for k in range (3): # we test the neural network by visually comparing
the prediction of the evolution
  im30 = MtoIm(L28[k],30)  # and the real evolution
  plt.imshow(im30)
  plt.show()

  im31 = MtoIm(H1[k],30)
  plt.imshow(im31)
  plt.show()
print (H1[1][0])



def fonction (T,N): # We make the table evolve until the final state
  T1= T
  a=0
  for i in range (10000): # the final state is reached each time well b
efore 10000 evolutions
    T1= evolution (T,N)
    if (T==T1).all()== True: # we get out of the loop as soon as 2 evol
utions in a row are identical (= the array will not evolve anymore)
      a=i
      break
    else:
      T=T1
  return (T1) # we send back the final table

def listeT20 (L0,N):  # we create an array list in the final state from
 L0
  L20=[]
  for k in L0:
    L20.append(fonction(k,N))
  return (L20)

L20= listeT20 (L0, N) # L20 corresponds to the end list of L0

L1,L5=shapem (L0,L20) # L1 and L5 are the compatible versions of L0 and
 L20 with Conv2D



rnvideraté = Sequential() # We create our first neural network
Depay=Conv2D(1, (3, 3), input_shape=(N, N, 1) , padding='same', activat
ion='tanh') # a convolution with 1 filter of dimension 3.3
rnvideraté.add(Depay)

# we train our neural network with tables having undergone 1 evolution
```

```
rnvideraté.compile(loss='mse', optimizer='adadelta', metrics=['accuracy
'])
rnvideraté.fit(x=L1, y=L5, batch_size=64, epochs=100, verbose=1, callba
cks=None, validation_split=0.0, validation_data=None, shuffle=True, cla
ss_weight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=No
ne, validation_steps=None, validation_freq=1, max_queue_size=10, worker
s=1, use_multiprocessing=False)


def fonction2 (T,N): # the function returns the number of changes befor
e the table is stabilized
  T1= T
  a=0
  for i in range (1000):
    T1= evolution (T,N)
    if (T==T1).all()== True:
      a=i
      break
    else:
      T=T1
  return (a)

T1000= fonction2(T0,N)
print (T1000)

def listeT1000 (L0,N): # This function returns the list of the devoluti
on number before the stabilization of the starting tables
  L1000=[]
  for k in L0:
    L1000.append(fonction2(k,N))
  return (L1000) # This list will allow us to deduce an optimal convolu
tion number

L1000=listeT1000 (L0,N)


z = max(L1000)
y= mean (L1000)
print (z)
print (y)
x= int(y) # x corresponds to the average of the number of table evoluti
ons
print (x)


T20=L0[20]
T21=L20[20]

im3 = MtoIm(T20,30) # we compare a starting table and an ending table
plt.imshow(im3)
plt.show()

im4 = MtoIm(T21,30)
plt.imshow(im4)
plt.show()



rnvide2= Sequential() # We create a new neural network
rnvide2.add(Conv2D(1, (3, 3), input_shape=(N, N, 1) , padding='same', a
ctivation='tanh')) # First convolution
for k in range (x): # We add x convolutions
  rnvide2.add(Conv2D(1, (3, 3), padding='same', activation='tanh'))

# we train the neural network with arrays in their initial and final st
ates

rnvide2.compile(loss='mse', optimizer='adadelta', metrics=['accuracy'])
rnvide2.fit(x=L1, y=L5, batch_size=64, epochs=300, verbose=1, callbacks
=None, validation_split=0.0, validation_data=None, shuffle=True, class_
weight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=None,
 validation_steps=None, validation_freq=1, max_queue_size=10, workers=1
, use_multiprocessing=False)



H2=rnvide2.predict(L1, batch_size=None, verbose=0, steps=None, callback
s=None, max_queue_size=10, workers=1, use_multiprocessing=False)

H3=np.around(H2) # H3 correspond à la prédiction de L1

# we test the neural network by visually comparing the prediction and t
he real final state

t=rd.randint (0,1001)
print(t)
im34 = MtoIm(L5[t],30) # we visually compare the arrival list and the p
rediction according to the neural network
plt.imshow(im34)
plt.show()

im35 = MtoIm(H3[t],30)
plt.imshow(im35)
plt.show()


def evolution2 (T,N): # we rewrite the evolution with the new rules
  M=np.zeros ((N,N))
  a=0
  b=0
  for i in range (N):
    for j in range (N):
      if i>0 and i<N-1 and j>0 and j<N-1:
        a=T[i,j]+T[i+1,j]+T[i-1,j]+T[i,j+1]+T[i,j-
1]+T[i+1,j+1]+T[i+1,j-1]+T[i-1,j+1]+T[i-1,j-1]
      if i==0 and j!=0 and j!=N-1:
        a=T[i,j]+T[i+1,j]+T[i,j+1]+T[i,j-1]+T[i+1,j+1]+T[i+1,j-1]
      if i==N-1 and j!=0 and j!=N-1:
        a=T[i,j]+T[i-1,j]+T[i,j+1]+T[i,j-1]+T[i-1,j+1]+T[i-1,j-1]
      if j==0 and i!=0 and i!=N-1:
        a=T[i,j]+T[i+1,j]+T[i-1,j]+T[i,j+1]+T[i+1,j+1]+T[i-1,j+1]
      if j==N-1 and i!=0 and i!=N-1:
        a=T[i,j]+T[i+1,j]+T[i-1,j]+T[i,j-1]+T[i+1,j-1]+T[i-1,j-1]
      if i==0 and j==0:
        a=T[i,j]+T[i+1,j]+T[i,j+1]+T[i+1,j+1]
      if i==0 and j==N-1:
        a=T[i,j]+T[i+1,j]+T[i,j-1]+T[i+1,j-1]
      if i==N-1 and j==0:
        a=T[i,j]+T[i-1,j]+T[i,j+1]+T[i-1,j+1]
      if i==N-1 and j==N-1:
        a=T[i,j]+T[i-1,j]+T[i,j-1]+T[i-1,j-1]
      if a==0:
        b=0
      elif a<0 and T[i,j]!=1: # we have to change some conditions
        b=-1
      elif a<0 and T[i,j]==1: # we forbid the transformation from 1 to
-1
        b=0
      elif a>0 and T[i,j]!=-1:
        b=1
      elif a>0 and T[i,j]==-1:# # we forbid the transformation from -
1 to 1
        b=0
      M[i,j]=b
  return (M)

M16= evolution2 (T0,30)
print (T0)
print (M16)



im = MtoIm(T0,30) # we visually compare the differences between the two
 rules
plt.imshow(im)
plt.show()
```

```
im11 = MtoIm(M0,30)
plt.imshow(im11)
plt.show()

im16 = MtoIm(M16,30)
plt.imshow(im16)
plt.show()



L0= listeT0 (30)

def listeT200 (L0,N): # we create a list of arrays having undergone an
evolution from the list L0 according to the new rule
  L200=[]
  for k in L0:
    L200.append(evolution2(k,N))
  return (L200)

L200 = listeT200 (L0,30) # L200 corresponds to the end list of L0 accor
ding to the new rule




L1,L40=shapem (L0,L200) # L1 et L40 sont les versions compatibles de L0
 et L200 avec Conv2D

print (L1.shape)
print(L40.shape)





rnvide3 = Sequential() # We recreate our new neural network
Depay=Conv2D(1, (3, 3), input_shape=(N, N, 1) , padding='same', activat
ion='tanh')
rnvide3.add(Depay)


rnvide3.compile(loss='mse', optimizer='adadelta', metrics=['accuracy'])

rnvide3.fit(x=L1, y=L40, batch_size=64, epochs=68, verbose=1, callbacks
=None, validation_split=0.0, validation_data=None, shuffle=True, class_
weight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=None,
 validation_steps=None, validation_freq=1, max_queue_size=10, workers=1
, use_multiprocessing=False)
```

```
def fonction3 (T,N): # This function makes the table evolve to the fina
l state with the new rule
  T1= T
  a=0
  for i in range (10000):
    T1= evolution2 (T,N)
    if (T==T1).all()== True:
      a=i
      break
    else:
      T=T1
  return (T1)

def fonction4 (T,N):# the function returns the number of evolutions bef
ore the table is stabilized with the new rule
  T1= T
  a=0
  for i in range (1000):
    T1= evolution2 (T,N)
    if (T==T1).all()== True:
      a=i
      break
    else:
      T=T1
  return (a)

T10000= fonction4(T0,N)
print (T10000)

def listeT10000 (L0,N): # This function returns the list of the evoluti
on number of each array before stabilization
  L10000=[]
  for k in L0:
    L10000.append(fonction4(k,N))
  return (L10000)

L10000=listeT10000 (L0,N)

m = max(L10000)
n= mean (L10000)
print (m)
print (n)
o= int(n) # o is the average number of evolutions
print (o)


def listeT2000 (L0,N):
  L2000=[]
  for k in L0:
    L2000.append(fonction3(k,N))
  return (L2000)

L2000= listeT2000 (L0, N) # L2000 is the table list in the final state
with the new rule

L1,L50=shapem (L0,L2000) # L1 and L50 are the compatible versions of L0
 and L2000 with Conv2D



rnvide4= Sequential() # We create our second neural network
rnvide4.add(Conv2D(1, (3, 3), input_shape=(N, N, 1) , padding='same', a
ctivation='tanh')) # First convolution
for k in range (o): # We add o convolutions
  rnvide4.add(Conv2D(1, (3, 3), padding='same', activation='tanh'))

rnvide4.compile(loss='mse', optimizer='adadelta', metrics=['accuracy'])
rnvide4.fit(x=L1, y=L50, batch_size=64, epochs=300, verbose=1, callback
s=None, validation_split=0.0, validation_data=None, shuffle=True, class
_weight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=None
, validation_steps=None, validation_freq=1, max_queue_size=10, workers=
1, use_multiprocessing=False)



H4=rnvide4.predict(L1, batch_size=None, verbose=0, steps=None, callback
s=None, max_queue_size=10, workers=1, use_multiprocessing=False)

H5=np.around(H4) # H5 corresponds to the prediction of the evolution of
 L1 according to the new rule

t=rd.randint (0,1001)
print(t)

im330 = MtoIm(L1[t],30)
plt.imshow(im330)
plt.show()

im340 = MtoIm(L50[t],30)
plt.imshow(im340)
plt.show()

im3500 = MtoIm(H5[t],30)
plt.imshow(im3500)
plt.show()
```

```
rnvide5= Sequential() # We create our final neural network
rnvide5.add(Conv2D(50, (3, 3), input_shape=(N, N, 1) , padding='same',
activation='tanh')) # This time with 50 filters
for k in range (o-1): # We add o-1 convolutions to 50 filters each
  rnvide5.add(Conv2D(50, (3, 3), padding='same', activation='tanh'))
rnvide5.add(Conv2D(1, (3, 3), padding='same', activation='tanh'))
# the last convolution must have a unique filter to be of good dimensio
n

rnvide5.compile(loss='mse', optimizer='adadelta', metrics=['accuracy'])
rnvide5.fit(x=L1, y=L50, batch_size=64, epochs=300, verbose=1, callback
s=None, validation_split=0.0, validation_data=None, shuffle=True, class
_weight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=None
, validation_steps=None, validation_freq=1, max_queue_size=10, workers=
1, use_multiprocessing=False)



H6=rnvide5.predict(L1, batch_size=None, verbose=0, steps=None, callback
s=None, max_queue_size=10, workers=1, use_multiprocessing=False)

H7=np.around(H6) # work

t=rd.randint (0,1001)
print(t)

im3300 = MtoIm(L1[t],30) # We visually compare the precondition and the
 final result
plt.imshow(im3300)
plt.show()

im3400 = MtoIm(L50[t],30)
plt.imshow(im3400)
plt.show()

im35000 = MtoIm(H7[t],30)
plt.imshow(im35000)
plt.show()
```