

13/15 points

Exercise Sheet 4: Neural Mass Models & Networks

```
In [1]: import numpy as np
from matplotlib import pyplot as plt
from scipy import spatial
from scipy import signal

def logistic(h,a=1):
    return 1/(1+np.exp(-a*h))
def dlogistic(h,a=1):
    return a*(logistic(h,a)-np.power(logistic(h,a),2))

def gaussRBF(h,a=1):
    return np.exp(-(a*h)**2)

def dgaussRBF(h,a=1):
    return -2*a**2*h*np.exp(-(a*h)**2)
```

Code given: Gradient Descent on a single neuron and an MLP

The neuron class below from sheet 1 is extended and includes the Gradient Descent training algorithm as a method. It uses the sum of squares error to measure the output error. A method calculating the last local error (δ) of the neuron and one for training the neuron using the derivative of the output error $\frac{dE}{dy} = y - o$ is included.

The train function updates the weights by the gradient descent weight update rule $w(t+1) = w(t) - \eta \delta(t) \cdot x(t)$ and the bias by $b(t+1) = b(t) - \eta \delta(t)$ with η being the learning rate. As input arguments it should get the derivative of the output error $\frac{dE}{dy} = y - o$ and the learning rate.

The class "MLP" includes a list of layers of type "MLPlayer" called "MLP.layers". Also, there is a method "MLP.out(x)" that returns the outputs of the whole network of the input vector "x".

The size of the weight vector is set to the number of inputs for the first layer and the number of inputs for the following layers corresponds to the number of neurons in the preceding layer. The number of outputs equals the number of neurons in the last layer.

The backpropagation training algorithm is included as a method "MLP.train()" into the class. The passed arguments should consist of the number of iterations (no stopping criteria in this case), the training input and the training output - both as function pointers - as well as the learning rate. It iterates over the layers, which themselves iterate over their neurons. Deltas and W of the following layer will always be the input to the previous.

The function pointer argument x_train has to link to a function that produce a random array of inputs of size [NolInputs,] and o_train has to produce the corresponding target function output for a given input. x_train should work without an argument passed and the target training output o

should be calculated using only that vector x .

The output of the method *train()* consists of the sum-of-squares error within each iteration.

```

In [2]: class neuron:
    lasth=0
    lastout=0
    lastin=0

    def __init__(self,w,b,trafunc=logistic,dtrafunc=dlogistic):
        self.w=np.array(w)
        self.b=np.array(b)
        self.trafunc=trafunc
        self.dtrafunc=dtrafunc

    def out(self, x):
        self.lastin=x;
        self.lasth=np.dot(self.w,x)-self.b
        self.lastout=self.trafunc(self.lasth)
        return self.lastout

    def delta(self,deltanext,weightsnext):
        df=self.dtrafunc(self.lasth)
        self.lastdelta=np.dot(deltanext,weightsnext)*df
        return self.lastdelta

    def train(self,deltanext,weightsnext,learnrate=0.1):
        self.delta(deltanext,weightsnext)
        self.w=self.w-learnrate*self.lastdelta*self.lastin
        self.b=self.b+learnrate*self.lastdelta
        return self.lastdelta

class MLPlayer:
    def __init__(self,NodeNo,WeightNo,weightinit=np.random.randn,biasinit=n
        self.nodes=[neuron(weightinit(WeightNo),biasinit(1),trafunc,dtrafun
    def out(self,x):
        return np.ravel([n.out(x) for n in self.nodes])
    def train(self,deltanext,W,learnrate=0.1):
        Wo=np.array([n.w for n in self.nodes]).T
        deltas=np.array([ n.train(deltanext,W[ineur],learnrate) for ineur,n
        return deltas, Wo

class MLP:
    def __init__(self,InputNo,NodeNos,weightinit=np.random.randn,biasinit=n
        self.NodeNos=np.append(InputNo,NodeNos)
        self.layers=[MLPlayer(self.NodeNos[i+1],self.NodeNos[i],weightinit,
    def out(self,x):
        for i in range(len(self.layers)):
            x=self.layers[i].out(x)
        self.lastout=x
        return x
    def train(self,NumIt,x_train,o_train,learnrate=0.1):
        errors=np.zeros(NumIt)
        for i in range(NumIt):
            if callable(x_train):
                x=x_train()
            else:
                x=x_train[i]
            if callable(o_train):
                o=o_train(x)

```

```

else:
    o=o_train[i]
y=self.out(x)
if len(y)==1:
    o=np.array([o])
    y=np.array([y])
deltas=y-o
W=np.eye(y.shape[0])
for il, l in enumerate(reversed(self.layers)):
    deltas, W=l.train(deltas,W,learnrate)
errors[i]=0.5*np.sum(np.power(y-o,2))
return errors

```

Task 1: logical functions (2 points - programming)

1.5/2

Train a single neuron on the logical functions below for a two-dimensional input x . Use instances of the neuron class above to build the equivalents to logical "or", "and" and "xor"-functions and test them for 2-dimensional input vectors x resembling all possibilities of combinations $([0,0], [1,0], [0,1], [1,1])$. Do 10.000 iterations and plot the evolution of the error (the error over the iteration number). You don't need to implement a stopping criterion.

Set the **learning rate to** $\eta = 1$ and initialize the weight w and the bias b randomly with normal distribution (`np.random.randn`). Use the logistic function.

-0.5

In the next cell you find an exemplary random number generator and the corresponding functions you can use for sample creation in every single iteration. In every iteration use the random input x , the neuron output $y = \text{neuron.out}(x)$ and the training data $o = \text{targetfunction}(x)$

```

In [3]: def train_and_display(NumIt,neuron1,inputfunc,targetfunc):
        errors=np.zeros(NumIt)
        for i in range(NumIt):
            x=inputfunc()
            neuron1.train(neuron1.out(x)-targetfunc(x),1)
            errors[i]=0.5*np.sum(np.power(neuron1.out(x)-targetfunc(x),2))
        plt.plot(errors)
        return neuron1

```

```

In [4]: # neuron out: x -> w*x-b
        # targetfunc: x -> and(x),or(x),xor(x)

```

```

In [5]: # x:
        random_input = lambda :np.random.randint(2,size=(2,))

        # w:
        w = np.random.randn(2)

        # b:
        b = np.random.randn(1)

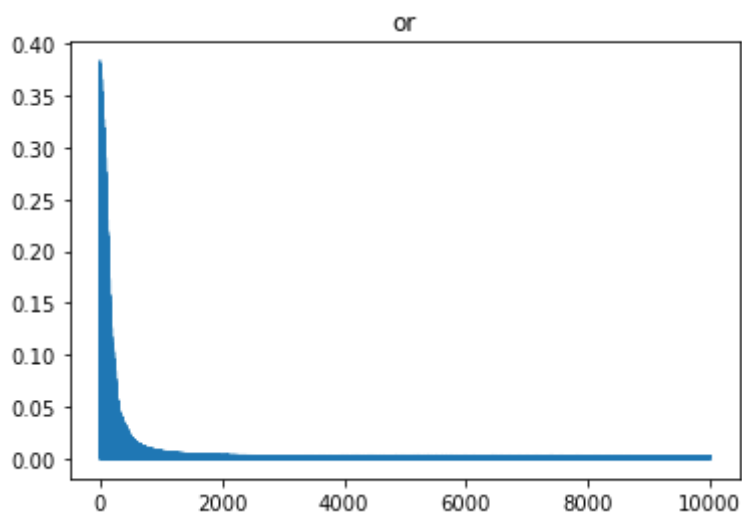
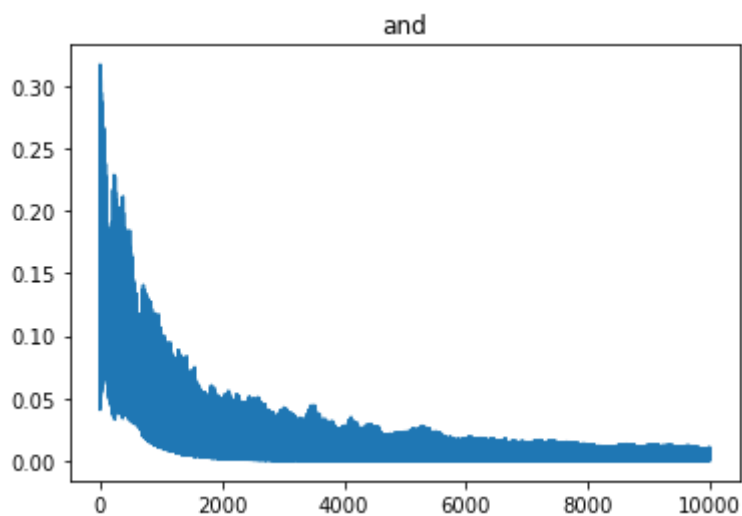
```

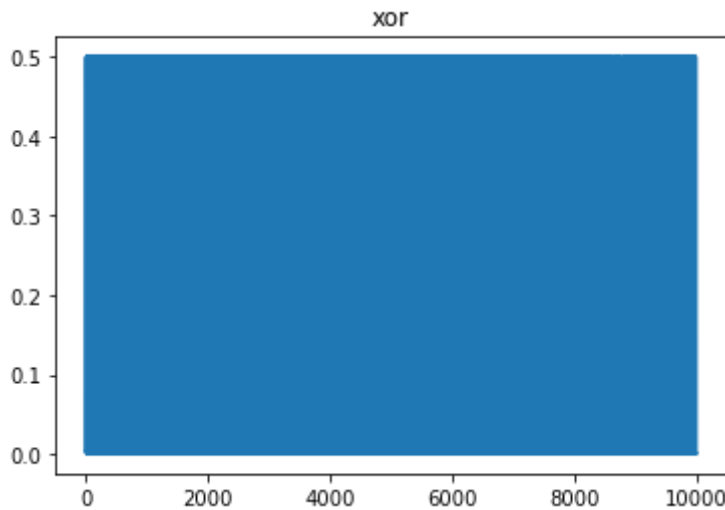
```
In [6]: neuron1 = neuron(w, b)

# and
train_and_display(10000, neuron1, random_input, lambda x:x[0] & x[1])
plt.title('and')
plt.show()

# or
train_and_display(10000, neuron1, random_input, lambda x:x[0] | x[1])
plt.title('or')
plt.show()

# xor
train_and_display(10000, neuron1, random_input, lambda x:x[0] ^ x[1])
plt.title('xor')
plt.show()
```





Task 2 Train an MLP on the XOR (2 points - programming)

2/2

Train a Multi-Layer-Perceptron on the logical "xor"-function. Do the same training on an XOR as in exercise sheet 2 task 5, using an architecture of 2 and then 3 neurons in the first layer and one output neuron in the second (you may use the sample solution code). Do 10.000 iterations and plot the evolution of the error. You don't need to implement a stopping criterion. Use the logistic transfer function.

Set the learning rate to $\eta = 1$.

Investigate the following steps:

- The network should consist of two layers, where the first has the two input neurons and the second only one output neuron. Does it always converge?
- The network should consist of two layers, where the first has the three input neurons and the second only one output neuron. Does it now always converge?

What can we learn from this?

The MLP is already implemented above. The syntax of the contained MLP class is:

NeuralNetwork=MLP(NoInputs,ListNoNeuronsPerLayer) for initialization and *errors=NeuralNetwork.train(NoIterations,x,o,learnrate)* for training. x and o can either be function pointers as defined or arrays of samples. If they are functions, x has to produce a random array of inputs of size $[NoInputs,]$ and $o(x)$ has to produce the corresponding target function output. If they are samples they have to have the shape $x.shape=[NoIterations,NoInputs]$ and $o.shape=[NoIterations,NoOutputs]$.

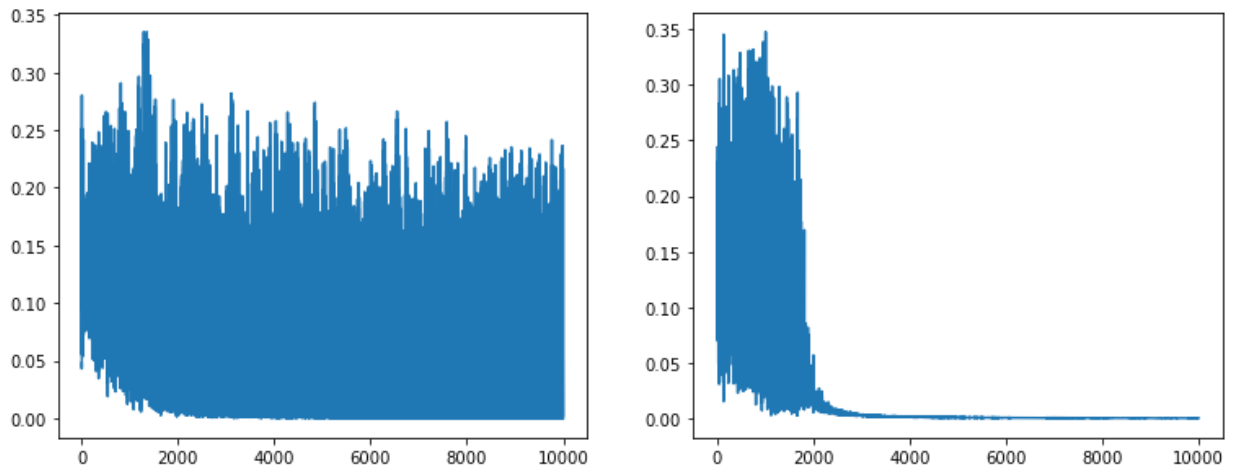
See descriptions above for further explanations.

a)two input and one output: it doesn't always converge.

```
In [11]: neuralnet1_1 = MLP(2,[2,1])
neuralnet1_2 = MLP(2,[2,1])
errors1_1 = neuralnet1_1.train(10000, random_input,lambda x:x[0] ^ x[1],1)
errors1_2 = neuralnet1_2.train(10000, random_input,lambda x:x[0] ^ x[1],1)

fig = plt.figure(figsize=[12.8, 4.8])
ax = plt.subplot(121)
plt.plot(errors1_1)
ax = plt.subplot(122)
plt.plot(errors1_2)
```

Out[11]: [

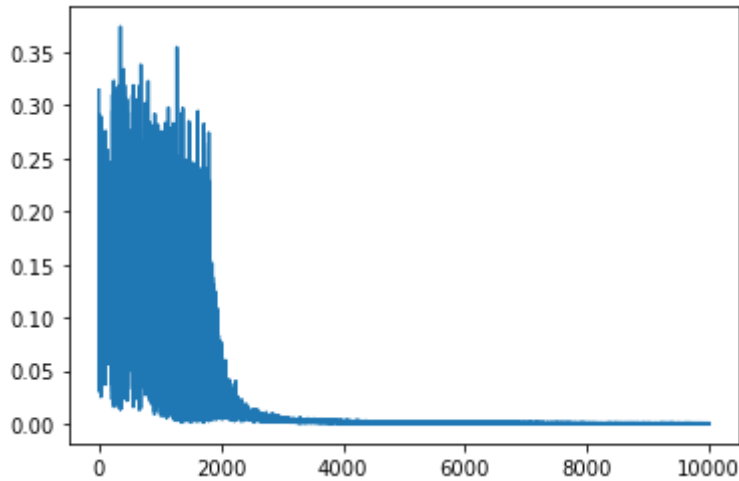


```
In [12]: # It is not always converged after 10,000 iteration according to the plots.
```

b)three input and one output: it always converges.

```
In [13]: neuralnet2 = MLP(2,[3,1])
errors2 = neuralnet2.train(10000, random_input,lambda x:x[0] ^ x[1],1)
plt.plot(errors2)
```

```
Out[13]: [<matplotlib.lines.Line2D at 0x7ff5e2c89100>]
```



```
In [14]: # It isconverged after 10,000 iteration according to the plots.
```

what can we learn from this?

Type *Markdown* and LaTeX: α^2

```
In [15]: # More layers, the neuron network converges more, we get a more solide resu
```

Task 3: A neural network described by a connectivity matrix (5 points - programming)

5/5

We will now translate the network described by objects into a connectivity matrix and then iterate over points in time in order to propagate the input to the output.

The connectivity is only supposed to be close to instantaneous in this task, we will introduce delay between different areas in the next task. The connectivity is hence a matrix W of size (NoNeurons, NoNeurons).

The vector of network state variables \mathbf{x}_t at time point t shall include the input in the first variables. The new state \mathbf{x}_{t+1} for the next time step is calculated by: $\mathbf{x}_{t+1} = f(\mathbf{h}) = f(W \cdot \mathbf{x}_t - \mathbf{b})$

Tasks:

a) Calculate the connectivity matrix W and the bias vector \mathbf{b} out of the MLP from task 2 b) (if you have not succeeded with training the MLP simply use a feedforward MLP of size (3,1) with arbitrary wiring). To this extent, translate layerwise indexed neurons into a unique index across layers and then connect neurons from each layer only with the preceeding layer/the input, respectively. Print the matrix to command line.

b) Simulate a propagation of a random logic vector of size 2 (as in task 2) until it reaches the output layer. This means iteration over time points, where in each time point the output $f(\mathbf{h})$ from the preceeding time point t is used as the input for the next time point $t + 1$. Print the state variables x_t at every step and the output of the network at the end to the command line.

a)M & b

```
In [16]: W = np.zeros((6, 6))
```

```
In [17]: W[2][0] = neuralnet2.layers[0].nodes[0].w[0] # if we use the notation in cl
W[2][1] = neuralnet2.layers[0].nodes[0].w[1] # x2 -> node3
W[3][0] = neuralnet2.layers[0].nodes[1].w[0] # x1 -> node4
W[3][1] = neuralnet2.layers[0].nodes[1].w[1] # x2 -> node4
W[4][0] = neuralnet2.layers[0].nodes[2].w[0] # x1 -> node5
W[4][1] = neuralnet2.layers[0].nodes[2].w[1] # x2 -> node5
W[5][2] = neuralnet2.layers[1].nodes[0].w[0] # node3 -> node6
W[5][3] = neuralnet2.layers[1].nodes[0].w[1] # node4 -> node6
W[5][4] = neuralnet2.layers[1].nodes[0].w[2] # node5 -> node6
```

```
In [18]: np.round(W,4)
```

```
Out[18]: array([[ 0.      ,  0.      ,  0.      ,  0.      ,  0.      ,  0.      ],
 [ 0.      ,  0.      ,  0.      ,  0.      ,  0.      ,  0.      ],
 [ 6.3244, -4.2505,  0.      ,  0.      ,  0.      ,  0.      ],
 [ 6.2787,  5.4037,  0.      ,  0.      ,  0.      ,  0.      ],
 [ 2.4426, -5.8812,  0.      ,  0.      ,  0.      ,  0.      ],
 [ 0.      ,  0.      , -8.1261,  6.6025,  8.117 ,  0.      ]])
```

```
In [19]: b = np.array([0,
                       0,
                       neuralnet2.layers[0].nodes[0].b[0],
                       neuralnet2.layers[0].nodes[1].b[0],
                       neuralnet2.layers[0].nodes[2].b[0],
                       neuralnet2.layers[1].nodes[0].b[0]]).reshape(6,1)
```

```
In [20]: np.round(b,4)
```

```
Out[20]: array([[ 0.      ],
 [ 0.      ],
 [-1.7509],
 [ 1.7661],
 [ 0.095 ],
 [ 2.2128]])
```

b)simulate

```
In [21]: # the input x should be x=[x1,x2] as we used before
# the input to the Wx-b should be xt= [x1, x2, node00, node01, node02, node03]
f = logistic
x0 = np.append(random_input(), [0, 0, 0, 0]).reshape(6,1)
x1 = f(np.dot(W,x0)-b)
x2 = f(np.dot(W,x1)-b)
y = np.round(x2[-1])
```

```
In [22]: np.round(x0,4) # the input [x1,x2,nodes...]
```

```
Out[22]: array([[1],
               [0],
               [0],
               [0],
               [0],
               [0]])
```

```
In [23]: np.round(x1,4) # the input has gone throught the fisrt layer.
```

```
Out[23]: array([[0.5   ],
               [0.5   ],
               [0.9997],
               [0.9891],
               [0.9127],
               [0.0986]])
```

```
In [24]: np.round(x2,4) # the result of first layer went throught the second layer
```

```
Out[24]: array([[0.5   ],
               [0.5   ],
               [0.942  ],
               [0.9833],
               [0.1401],
               [0.9735]])
```

```
In [25]: print('x1= %d, x2= %d,\ny=(x1)xor(x2)= %d'%(x0[0],x0[1],y))
```

```
x1= 1, x2= 0,
y=(x1)xor(x2)= 1
```

Task 6 : Neural Mass Model (6 points - programming)

4.5/6

We will now look at a neural mass model based on the source positions from our head model (gridpos.npy). You will find the connected equations again on the last slides of the current lecture.

**** Tasks: ****

a) Derive the distance matrix **D** from the source positions in gridpos using `scipy.spatial.distance_matrix(...)`.

b) Calculate the matrix of weights **W₀** with dimensionality (NoSources,NoSources) by distance without delay using the equation $\mathbf{W} = a_0 e^{b_0 \mathbf{D}} + a_1 e^{b_1 \mathbf{D}}$. Use $a_0 = 0.03$, $b_0 = 10$ for replicating excitatory connections and $a_1 = -0.25$ and $b_1 = 50$ for inhibitory. Plot the matrix **W₀** using

-0.5

imshow.

c) Calculate the delay $\Delta t = D/c + \Delta t_0$ for each connection by using the transmission speed $c = 0.5 \frac{m}{s}$ and the synaptic delay $\Delta t_0 = 2ms$ as a matrix corresponding to the dimensions of \mathbf{D} . Then introduce a sampling rate of $fs = 1000Hz$ and discretize the delay into time samples. Calculate the maximal time delay T .

d) Combine the results of b) and c) into an array of size (NoSources,NoSources,MaxDelay) by expanding W_0 into the time domain/delay. We will call this array W .

e) Simulate the multi-variate time course for a random starting point of the network over 1s with $fs = 1000Hz$ by creating a random array X of size (NoSources,MaxDelay) with uniform distribution (`np.random.rand`) and then propagate activity over time. To this extent, use the current state history array \mathbf{X} of the network to calculate the source activity vector for the next time step by $\mathbf{y} = f(\mathbf{W} \odot \mathbf{X})$. \odot can be calculated using `np.tensordot(W,X,axes=2)`. Then \mathbf{X} can be shifted like a circular buffer so that the oldest source activity vector is discarded while the newest is set to \mathbf{y} .

```
In [26]: gridpos=np.load('/Users/changmincheng/Desktop/BCI/Lecture 3/Exercise3_headm
Nsource=gridpos.shape[0]
```

```
In [27]: Nsource
```

```
Out[27]: 3990
```

a) distance matrix D

```
In [28]: D = spatial.distance_matrix(gridpos,gridpos) # Dij =the distance between so
np.round(D,4) # diagonal = 0, distance from a point to itself is 0.
```

```
Out[28]: array([[0.      , 0.0038, 0.0038, ..., 0.1585, 0.1591, 0.1607],
 [0.0038, 0.      , 0.0042, ..., 0.1567, 0.1573, 0.1584],
 [0.0038, 0.0042, 0.      , ..., 0.1558, 0.1569, 0.1585],
 ...,
 [0.1585, 0.1567, 0.1558, ..., 0.      , 0.0242, 0.0359],
 [0.1591, 0.1573, 0.1569, ..., 0.0242, 0.      , 0.0267],
 [0.1607, 0.1584, 0.1585, ..., 0.0359, 0.0267, 0.      ]])
```

b) the first weight matrix W0

```
In [29]: a0 = 0.03
b0 = 10
a1 = -0.25
b1 = 50
#w0ij=a0*e^{b0*dij}+a1*e^{b1*dij}
W0 = np.zeros((Nsource, Nsource))
for i in range(Nsource):
    for j in range(Nsource):
        W0[i][j] = a0 * np.exp(-1 * b0 * D[i][j]) + a1 * np.exp(-1 * b1 *
```

```
In [30]: np.round(W0,4) # to check if the calculation is right: diagonal = a0*e^0+a1
Out[30]: array([[ -0.22   , -0.1782, -0.1776, ...,  0.0061,  0.006 ,  0.0059],
 [ -0.1782, -0.22   , -0.1738, ...,  0.0062,  0.0061,  0.0061],
 [ -0.1776, -0.1738, -0.22   , ...,  0.0062,  0.0061,  0.0061],
 ...,
 [  0.0061,  0.0062,  0.0062, ..., -0.22   , -0.0509, -0.0206],
 [  0.006 ,  0.0061,  0.0061, ..., -0.0509, -0.22   , -0.043 ],
 [  0.0059,  0.0061,  0.0061, ..., -0.0206, -0.043 , -0.22   ]])
```

c) maximal delay T

```
In [31]: fs=100 # it takes too long to calculate fs=1000, here chose 100
c=0.5
wde=10
wdi=50
t0 = 0.002
```

```
In [32]: delta_t = ( D/c + t0) * fs # delta_t ij = the delay between source i and so
np.round(delta_t,4)
```

```
Out[32]: array([[ 0.2    ,  0.9525,  0.9655, ..., 31.9003, 32.0294, 32.3483],
 [ 0.9525,  0.2    ,  1.0407, ..., 31.5306, 31.6576, 31.8725],
 [ 0.9655,  1.0407,  0.2    , ..., 31.363 , 31.5814, 31.8954],
 ...,
 [31.9003, 31.5306, 31.363 , ...,  0.2    ,  5.0443,  7.3749],
 [32.0294, 31.6576, 31.5814, ...,  5.0443,  0.2    ,  5.5306],
 [32.3483, 31.8725, 31.8954, ...,  7.3749,  5.5306,  0.2    ]])
```

```
In [33]: T = np.floor(delta_t.max()) # the maximum delay
T
```

```
Out[33]: 34.0
```

d) weight matrix W = [W0, W1, ... W_T]

```
In [34]: # at current time, all transmissions are done, means we have all weights, s
# at time delay 1, because of the longest distance and speef of transmissio
### there were some neurons not 'activated'. they didn't work, their weight
### means if delta_t(ij) > T, w1(ij) =0; else wij = w0ij
# at time delay T-1, the transmissions are only finished in the shortest di

W=[W0]
delta_t_t = delta_t.copy()
for t in range(int(T)):
    W_t = W[-1].copy()
    delay_t = T - t
    W_t[ delta_t_t > delay_t] = 0
    delta_t_t[delta_t_t >= delay_t] = 0
    W.append(W_t)
```

```
In [35]: len(W)
```

```
Out[35]: 35
```

```
In [36]: np.round(W[-1],4) # check  $W_T$  the element in  $W_T$  has a delay < 1
```

```
Out[36]: array([[ -0.22   , -0.1782, -0.1776, ...,  0.       ,  0.       ,  0.       ],
                [ -0.1782, -0.22   ,  0.       , ...,  0.       ,  0.       ,  0.       ],
                [ -0.1776,  0.       , -0.22   , ...,  0.       ,  0.       ,  0.       ],
                ...,
                [  0.       ,  0.       ,  0.       , ..., -0.22   ,  0.       ,  0.       ],
                [  0.       ,  0.       ,  0.       , ...,  0.       , -0.22   ,  0.       ],
                [  0.       ,  0.       ,  0.       , ...,  0.       ,  0.       , -0.22   ]])
```

```
In [37]: W = np.array(W)
```

```
In [38]: W.swapaxes(2,0).shape
```

```
Out[38]: (3990, 3990, 35)
```

e) Simulation

```
In [39]: X = np.random.randn(Nsource, (int(T)+1))
```

```
In [40]: X.shape
```

```
Out[40]: (3990, 35)
```

```
In [41]: #Weight goes from 0 to T, T means Delay, so we remove x[35] and move every
#add new one at x[0]
for i in range(int(T)):
    y = logistic(np.tensordot(W.swapaxes(2,0),X,axes=2))
    X = np.vstack((y,X[:,0:34].swapaxes(1,0))).swapaxes(1,0)
```

```
In [44]: y
```

```
Out[44]: array([6.62749355e-25, 2.68171746e-28, 1.85383265e-28, ...,
                6.94151962e-60, 1.12309930e-59, 1.64598012e-53])
```

```
In [45]: y.shape[0]
```

```
Out[45]: 3990
```

```
In [ ]:
```

