Documentation pour la classe Neuron obtenable avec help(Neuron) :

```
class Neuron(builtins.object)
 |  fields:
 |     input_size: int
 |     beta: list of the coefficitents used in the linear combination of the
 |        outputs of the previous layer
 |
 |  Methods defined here:
 |
 |  __init__(self, input_size)
 |     Creates a Neuron object with random coefficients
 |     ----
 |     input:
 |        input_size: int -> n_c where c is the layer of the neuron
 |     ----
 |     output:
 |        void
 |
 |  comb_lin(self, Zc)
 |     Returns the sum of beta * Z
 |     ----
 |     Input:
 |        Zc: array of length n_c -> outputs of the previous layer
 |     ----
 |     output:
 |        out: float -> sum of beta * Z
 |
 |  compute_output(self, Zc)
 |     Given the outputs of the previous layer, computes the output of
 |     this neuron
 |     ----
 |     input:
 |        Zc : array of length n_c -> outputs of the previous layer
 |     ----
 |     output:
 |        out : float -> output of the neuron
```

Documentation pour la classe NeuralNetwork obtenable avec help(NeuralNetwork) :

```
class NeuralNetwork(builtins.object)
 |  fields:
 |     format: array of integers of size C -> (n_c)_c
 |     neuron_layers: array of arrays ofNeurons array length C. The c th layer has length n_c
 |        -> a line represents a layer of neurons
 |     Z_layers: float array array -> stores the current output of each Neurons
 |     learning_Rate: float -> learning Rate used for backpropagation
 |     current_input: float array of size p -> the input being computed
 |     errors: float array of size n_C -> the error of each line
 |     derivatives: float array -> derivatives[c][k][j] = dR / dbeta_(j, k)^c
 |
 |  Methods defined here:
 |
 |  __init__(self, format, p)
 |     Creates a NeuralNetwork object with random coefficients and a given
 |     size for each layer
 |     ----
 |     input:
 |        format: array of integer of size C -> (n_c)_c
 |        p: int -> the numberof column used as input of the network
 |     ----
 |     output:
 |        void
 |
 |  compute_all(self, database, outputs)
 |     Make the every line go through the network, storing the errors of each one
 |     ----
 |     input:
 |        database: array of shape (N, p) -> the training database
 |        outputs: array of shape (N, n_C) -> the expected outputs (y)
 |     ----
 |     output:
 |        void
 |
 |  compute_derivatives(self, expected_output)
 |     Adds the derivative of R_i with respect of every coefficient to the
 |     derivative matrix
 |     ----
 |     input:
 |        expected_output: float array of length n_C -> the outputs expected
 |           for the current inputs
 |     ----
 |     output:
 |        void
 |
 |  compute_error(self, expected_output)
```

```
|     Returns the current error
|     ----
|     input:
|       expected_output: float array of size n_C -> y
|     ----
|     output:
|       res: float -> R_i(theta)
|
|  compute_one(self, input)
|     Make the input go through the network and stores the outputs of each layer
|     ----
|     input:
|       input: array of floats of size p -> (x_i)_i
|     ----
|     output:
|       void
|
|  deriv_Z(self, m, cz, j, k, cb)
|     Returns the derivative of Z_m^cz with respect to beta_(j, k)^cb
|     ----
|     input:
|       m, cz, j, k, cb: int
|     ----
|     output:
|       res: float -> the derivative
|
|  deriv_error_i(self, j, k, c, expected_output)
|     Returns the derivative of R_i with respect to beta_(j, k)^c
|     ----
|     input:
|       j, k, c: int
|       expected_output: float array of length n_C -> the outputs expected
|         for the current inputs
|     ----
|     output:
|       res: float -> the derivative
|
|  predict(self, database)
|     Predicts the outputs for each line of the database
|     ---
|     input:
|       database: float array of shape (N, p)
|     ----
|     output:
|       prediction: float array of shape (N, n_C)
|
|  total_error(self)
|     Returns the error of one line
```

```
|      ----
|      input:
|          void
|      ----
|      output:
|          res: float -> R = \sum_i R_i
|
|  train(self, database, outputs, n)
|      Trains the network on the database
|      ----
|      input:
|          database: float array of shape (N, p)
|          outputs: float array of shape (N, n_C)
|          n: int -> how many times the database will go through the network
|      ----
|      output:
|          error_list: float array of length n -> the error after each turn
|
|  update_coeff(self)
|      Update every coefficient of the network using backpropagation
|      ----
|      input:
|          void
|      ----
|      output:
|          void
```

Bon fonctionnement :

```
[[array([-0.08982536, -0.89274098,  0.77577876]),
  array([-0.80687683,  0.77447995, -0.504361  ]),
  array([-0.39831706,  0.1897549 , -0.44296642]),
  array([-0.32492599,  0.89831503, -0.89571099]),
  array([-0.00400449,  0.43951317, -0.54238366])],
 [array([ 0.57707236,  0.57091261,  0.8467288 , -0.9330265 , -0.07194982]),
  array([-0.12016059,  0.41060086,  0.81954115,  0.16439526,  0.64751315]),
  array([-0.27295292,  0.38408992, -0.14306856,  0.97542059, -0.59960544])],
 [array([-0.74240986, -0.88103109, -0.84235402]),
  array([ 0.69122103, -0.25549439, -0.92839153])]]
```

Coefficients initiaux, générés aléatoirement dans $[-1,1]$, pour un réseau avec des entrées de taille 3, une couche de 5 neurones, une de 3 neurones puis deux sorties.
Les rectangles sont pour la suite.

```
[array([ 0.44848654,  0.3689421 ,  0.34264516,  0.42010998,  0.47330666]),
 array([ 0.58262103,  0.68003784,  0.52404548]),
 array([-1.47310989, -0.25754533])]
```

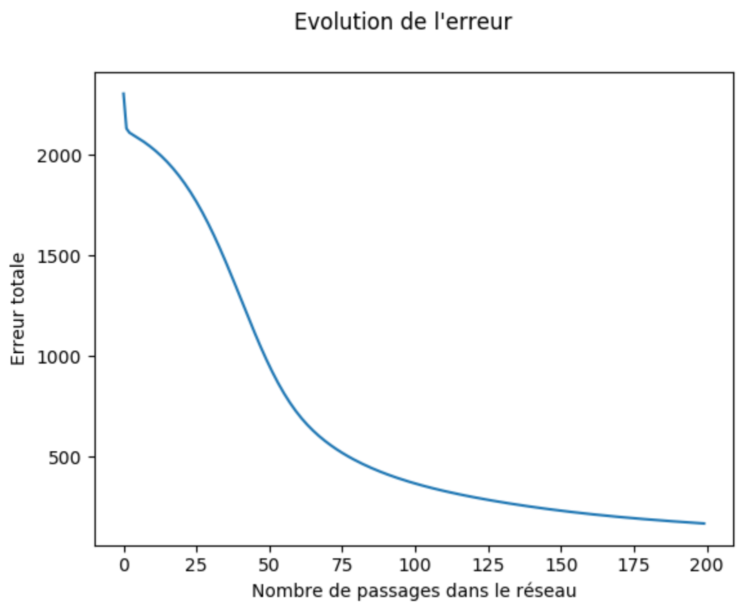
On fait passer l'entrée $(1,1,1)$ dans le réseau.
Ligne 1 et 2 : sortie des neurones des couches 1 et 2.
Ligne 3 : sortie du réseau de neurone (on applique pas sigma).
Rectangles pour ce qui suit

```
[[[-0.08647486268756685, -0.08647486268756685, -0.08647486268756685],
  [0.1893021138049199, 0.1893021138049199, 0.1893021138049199],
  [0.11216638830268734, 0.11216638830268734, 0.11216638830268734],
  [0.28890016443275446, 0.28890016443275446, 0.28890016443275446],
  [-0.06780056444161039, -0.06780056444161039, -0.06780056444161039]],
 [[0.048948279303379555,
   0.04026671826385422З,
   0.03739664252962862б,
   0.04585123215129355З,
   0.05165717247192816 4],
  [0.3160083536839233З,
   0.2599605037788610 4,
   0.24143139671670139,
   0.29601392720477626,
   0.33349687182350918],
  [0.538812742587959З,
   0.44324787738281113,
   0.4116546651228149,
   0.50472107494021712,
   0.56863168981748302]],
 [[-1.7165296069021385, -2.003540945223663, -1.5439531521823546],
  [-1.4653447097980814, -1.7103568229421868, -1.3180218824244663]]]
```

Les dérivées par rapport à chaque coefficient (première couche en haut, dernière en bas).
Par exemple, on a que $\frac{\partial R_i}{\partial \beta_{1,1}^2} = -2(y_{i,1} - \hat{y}_{i,1})Z_1^2$ par application de la formule. On a $y_{i,1} = 0, \hat{y}_{i,1} = -1.47\ldots, Z_1^2 = 0.58\ldots$ (termes encadrés en rouge avant), on trouve bien le même résultat.

De même, on a $\frac{\partial R_i}{\partial \beta_{1,1}^1} = -2Z_1^2(1 - Z_1^2)Z_1^1\left((y_{i,1} - \hat{y}_{-i}, 1)\beta_{1,1}^2 + (y_{i,2} - \hat{y}_{i,2})\beta_{1,2}^2\right)$

$$= -2 * 0.58262103 * (1 - 0.58262103) * 0.44848654 * \left((0 + 1.47310989) * -0.74240986 + (1 + 0.25754533) * 0.69122103\right) = 0.489\ldots$$



Évolution de l’erreur sur un échantillon de 4500 lignes de la base de données de spam (après mise à l’échelle), pour 200 passages dans un réseau $[5, 3, 2]$, avec un learning rate constant de $\frac{0.5}{4500}$.
Le procédé a pris plusieurs heures, et la prédiction sur les 101 lignes restant a montré un taux de réussite de 97%.