PROJET INFORMATIQUE INDIVIDUELLE

ENSC – Semestre 8

Perceptron multi-couches

Plan du cours

1. Théorie de la rétro-propagation du gradient

1.1. Problème

Soit le perceptron multi-couches dont les couches sont liées entre elles par des relations de type :

$$f_B(B * W) = A \Leftrightarrow f_B\left(\begin{bmatrix} B_1 & \cdots & B_{n_B} \end{bmatrix} * \begin{bmatrix} W_{1,1} & \cdots & W_{1,n_A} \\ \vdots & & \vdots \\ W_{n_B,1} & \cdots & W_{n_B,n_A} \end{bmatrix}\right) = \begin{bmatrix} A_1 & \cdots & A_{n_A} \end{bmatrix}$$

avec

- n_B le nombre d'entrées dans la couche B
- n_A le nombre de sorties dans la couche A
- f_B la fonction d'activation de la couche B ($f_B : \mathbb{R} \to [-1, 1]$)
- $B \in \mathcal{M}_{1,n_B}$ la matrice contenant les valeurs d'entrée
- $\{B_b, \forall\, b \in [\![1, n_B]\!]\}$ les valeurs des entrées
- $A \in \mathcal{M}_{1,n_A}$ la matrice contenant les valeurs de sortie
- $\left\{A_j, \forall\, a \in [\![1, n_A]\!]\right\}$ les valeurs des sorties
- $W \in \mathcal{M}_{n_B,n_A}$ la matrice contenant les poids
- $\{W_{b,a}, \forall\, (b, a) \in [\![1, n_B]\!] \times [\![1, n_A]\!]\}$ les poids entre les B_b et les A_a

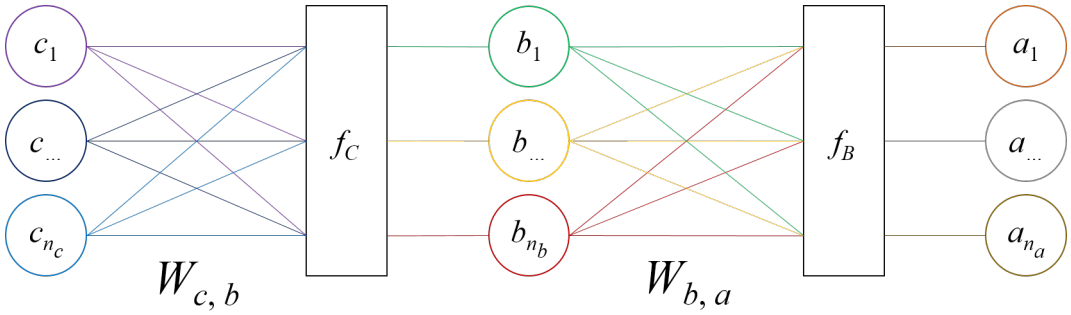

FIGURE 1 – Représentation des relations entre quelques couches de neurones

Dans le reste du document, $*$ désignera la multiplication matricielle et $\times$ la multiplication terme à terme.
On notera ${}^{\mathbf{t}}M$ la transposée de la matrice M.

1.2. Correction des poids de la dernière couche

Supposons que pour tout $a \in [\![1, n_A]\!]$ la sortie théorique soit α_a et non A_a.

L'erreur quadratique $\mathcal{E}$ commise sur un tel réseau est : $\mathcal{E} = \frac{1}{2} \sum_{\ell=1}^{n_A} (\alpha_\ell - A_\ell)^2$.

On s'intéresse aux variations qu'apporteraient les modifications de $W_{b,a}$ ($(b,a) \in [\![1, n_B]\!] \times [\![1, n_A]\!]$) sur l'erreur quadratique.

Il s'agit donc de faire varier les $W_{b,a}$ pour chaque couple $(b,a) \in [\![1, n_B]\!] \times [\![1, n_A]\!]$ de telle façon à ce que réduire au maximum $\mathcal{E}$.

On va donc calculer pour tout $(b,a) \in [\![1, n_B]\!] \times [\![1, n_A]\!]$ les dérivées partielles $\varepsilon_{b,a}$ de $\mathcal{E}$ par rapport aux $W_{b,a}$.

Pour un b et un a donnés :

$$
\begin{aligned}
\varepsilon_{b,a} &= \frac{\mathrm{d}}{\mathrm{d}W_{b,a}} \mathcal{E} \\
&= \frac{\mathrm{d}}{\mathrm{d}W_{b,a}} \frac{1}{2} \sum_{\ell=1}^{n_A} (\alpha_\ell - A_\ell)^2 \\
&= \frac{\mathrm{d}}{\mathrm{d}W_{b,a}} \frac{1}{2} \sum_{\ell=1}^{n_A} \left(\alpha_\ell - f_B \left(\sum_{k=1}^{n_B} B_k W_{k,\ell} \right) \right)^2 \\
&= \frac{1}{2} \sum_{\ell=1}^{n_A} \left[\frac{\mathrm{d}}{\mathrm{d}W_{b,a}} \left(\alpha_\ell - f_B \left(\sum_{k=1}^{n_B} B_k W_{k,\ell} \right) \right)^2 \right] \\
&= \sum_{\ell=1}^{n_A} \left[\left(\alpha_\ell - f_B \left(\sum_{j=1}^{n_B} B_j W_{j,\ell} \right) \right) \frac{\mathrm{d}}{\mathrm{d}W_{b,a}} \left(\alpha_\ell - f_B \left(\sum_{k=1}^{n_B} B_k W_{k,\ell} \right) \right) \right] \\
&= -\sum_{\ell=1}^{n_A} \left[\left(\alpha_\ell - f_B \left(\sum_{j=1}^{n_B} B_j W_{j,\ell} \right) \right) \frac{\mathrm{d}}{\mathrm{d}W_{b,a}} f_B \left(\sum_{k=1}^{n_B} B_k W_{k,\ell} \right) \right] \\
&= -\sum_{\ell=1}^{n_A} \left[\left(\alpha_\ell - f_B \left(\sum_{j=1}^{n_B} B_j W_{j,\ell} \right) \right) f_B' \left(\sum_{i=1}^{n_B} B_i W_{i,\ell} \right) \frac{\mathrm{d}}{\mathrm{d}W_{b,a}} \sum_{k=1}^{n_B} B_k W_{k,\ell} \right] \\
&= -\sum_{\ell=1}^{n_A} \left[\left(\alpha_\ell - f_B \left(\sum_{j=1}^{n_B} B_j W_{j,\ell} \right) \right) f_B' \left(\sum_{i=1}^{n_B} B_i W_{i,\ell} \right) \sum_{k=1}^{n_B} \frac{\mathrm{d}}{\mathrm{d}W_{b,a}} B_k W_{k,\ell} \right] \\
&= -\left(\alpha_a - f_B \left(\sum_{j=1}^{n_B} B_j W_{j,a} \right) \right) f_B' \left(\sum_{i=1}^{n_B} B_i W_{i,a} \right) \frac{\mathrm{d}}{\mathrm{d}W_{b,a}} B_b W_{b,a} \\
&= -\left(\alpha_a - f_B \left(\sum_{j=1}^{n_B} B_j W_{j,a} \right) \right) f_B' \left(\sum_{i=1}^{n_B} B_i W_{i,a} \right) B_b \\
&= -(\alpha_a - A_a) f_B' \left(\sum_{i=1}^{n_B} B_i W_{i,a} \right) B_b
\end{aligned}
$$

En notant $\varepsilon_a = \begin{bmatrix} \varepsilon_{1,a} & \cdots & \varepsilon_{n_B,a} \end{bmatrix} \in \mathcal{M}_{1,n_B}$, on a donc :

$$
\varepsilon_a = -(\alpha_a - A_a) f_B' \left(\sum_{i=1}^{n_B} B_i W_{i,a} \right) \begin{bmatrix} B_1 & \cdots & B_{n_B} \end{bmatrix} = -(\alpha_a - A_a) f_B' \left(\sum_{i=1}^{n_B} B_i W_{i,a} \right) B
$$

Puis en notant $\varepsilon = \begin{bmatrix} \varepsilon_{1,1} & \cdots & \varepsilon_{1,n_A} \\ \vdots & & \vdots \\ \varepsilon_{n_B,1} & \cdots & \varepsilon_{n_B,n_A} \end{bmatrix} = \begin{bmatrix} {}^{\mathbf{t}}\varepsilon_1 & \cdots & {}^{\mathbf{t}}\varepsilon_{n_B} \end{bmatrix} \in \mathcal{M}_{n_B,n_A}$, on en déduit que :

$$\begin{aligned}
\varepsilon &= \left[-(\alpha_1 - A_1) f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,1}\right) {}^{\mathbf{t}}B \quad \cdots \quad -(\alpha_{n_A} - A_{n_A}) f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,n_A}\right) {}^{\mathbf{t}}B \right] \\
&= -{}^{\mathbf{t}}B * \left[(\alpha_1 - A_1) f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,1}\right) \quad \cdots \quad (\alpha_{n_A} - A_{n_A}) f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,n_A}\right) \right] \\
&= -{}^{\mathbf{t}}B * \left(\left[(\alpha_1 - A_1) \quad \cdots \quad (\alpha_{n_A} - A_{n_A}) \right] \times \left[f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,1}\right) \quad \cdots \quad f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,n_A}\right) \right] \right) \\
&= -{}^{\mathbf{t}}B * \left[\left(\begin{bmatrix} \alpha_1 & \cdots & \alpha_{n_A} \end{bmatrix} - \begin{bmatrix} A_1 & \cdots & A_{n_A} \end{bmatrix} \right) \times f'_B\left(\begin{bmatrix} \sum_{i=1}^{n_B} B_i W_{i,1} & \cdots & \sum_{i=1}^{n_B} B_i W_{i,n_A} \end{bmatrix} \right) \right] \\
&= -{}^{\mathbf{t}}B * \left[(\alpha - A) \times f'_B (B * W) \right]
\end{aligned}$$

On posant $\delta_B = \left[(\alpha - A) \times f'_B (B * W) \right]$, on a $\varepsilon = -{}^{\mathbf{t}}B * \delta_B$.

1.3. Correction des poids de l'avant-dernière précédentes

Maintenant qu'on a calculé les variations à appliquer sur les poids entre la dernière et l'avant-dernière couche, il s'agit désormais de faire de même pour les poids entre les couches précédentes.

L'erreur quadratique $\mathcal{E}$ commise est toujours : $\mathcal{E} = \frac{1}{2}\sum_{\ell=1}^{n_A}(\alpha_\ell - A_\ell)^2$.

On s'intéresse maintenant aux variations qu'apporteraient les modifications de $W'_{c,b}$ ($(c,b) \in [\![1, n_C]\!] \times [\![1, n_B]\!]$) sur l'erreur quadratique.

Il s'agit donc de faire varier les $W'_{c,b}$ pour chaque couple $(c,b) \in [\![1, n_C]\!] \times [\![1, n_B]\!]$ de telle façon à ce que réduire au maximum $\mathcal{E}$.

On va donc calculer pour tout $(c,b) \in [\![1, n_C]\!] \times [\![1, n_B]\!]$ les dérivées partielles $\varepsilon'_{c,b}$ de $\mathcal{E}$ par rapport aux $W'_{c,b}$.

Pour un c et un b donnés :

$$
\begin{aligned}
\varepsilon'_{c,b} &= \frac{\mathrm{d}}{\mathrm{d}W'_{c,b}}\mathcal{E} \\
&= \frac{\mathrm{d}}{\mathrm{d}W'_{c,b}}\frac{1}{2}\sum_{\ell=1}^{n_A}(\alpha_\ell - A_\ell)^2 \\
&= \frac{\mathrm{d}}{\mathrm{d}W'_{c,b}}\frac{1}{2}\sum_{\ell=1}^{n_A}\left(\alpha_\ell - f_B\left(\sum_{k=1}^{n_B} B_k W_{k,\ell}\right)\right)^2 \\
&= \frac{\mathrm{d}}{\mathrm{d}W'_{c,b}}\frac{1}{2}\sum_{\ell=1}^{n_A}\left(\alpha_\ell - f_B\left(\sum_{k=1}^{n_B} f_C\left(\sum_{j=1}^{n_c} C_j W'_{j,k}\right) W_{k,\ell}\right)\right)^2 \\
&= \frac{1}{2}\sum_{\ell=1}^{n_A}\left[\frac{\mathrm{d}}{\mathrm{d}W'_{c,b}}\left(\alpha_\ell - f_B\left(\sum_{k=1}^{n_B} f_C\left(\sum_{j=1}^{n_c} C_j W'_{j,k}\right) W_{k,\ell}\right)\right)^2\right] \\
&= \sum_{\ell=1}^{n_A}\left[\left(\alpha_\ell - f_B\left(\sum_{i=1}^{n_B} f_C\left(\sum_{h=1}^{n_c} C_h W'_{h,i}\right) W_{i,\ell}\right)\right)\frac{\mathrm{d}}{\mathrm{d}W'_{c,b}}\left(\alpha_\ell - f_B\left(\sum_{k=1}^{n_B} f_C\left(\sum_{j=1}^{n_c} C_j W'_{j,k}\right) W_{k,\ell}\right)\right)\right] \\
&= \sum_{\ell=1}^{n_A}\left[(\alpha_\ell - A_\ell)\frac{\mathrm{d}}{\mathrm{d}W'_{c,b}}\left(\alpha_\ell - f_B\left(\sum_{k=1}^{n_B} f_C\left(\sum_{j=1}^{n_c} C_j W'_{j,k}\right) W_{k,\ell}\right)\right)\right] \\
&= -\sum_{\ell=1}^{n_A}\left[(\alpha_\ell - A_\ell)\frac{\mathrm{d}}{\mathrm{d}W'_{c,b}} f_B\left(\sum_{k=1}^{n_B} f_C\left(\sum_{j=1}^{n_c} C_j W'_{j,k}\right) W_{k,\ell}\right)\right] \\
&= -\sum_{\ell=1}^{n_A}\left[(\alpha_\ell - A_\ell) f'_B\left(\sum_{i=1}^{n_B} f_C\left(\sum_{h=1}^{n_c} C_h W'_{h,i}\right) W_{i,\ell}\right)\frac{\mathrm{d}}{\mathrm{d}W'_{c,b}}\left(\sum_{k=1}^{n_B} f_C\left(\sum_{j=1}^{n_c} C_j W'_{j,k}\right) W_{k,\ell}\right)\right] \\
&= -\sum_{\ell=1}^{n_A}\left[(\alpha_\ell - A_\ell) f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,\ell}\right)\frac{\mathrm{d}}{\mathrm{d}W'_{c,b}}\left(\sum_{k=1}^{n_B} f_C\left(\sum_{j=1}^{n_c} C_j W'_{j,k}\right) W_{k,\ell}\right)\right] \\
&= -\sum_{\ell=1}^{n_A}\left[(\alpha_\ell - A_\ell) f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,\ell}\right)\sum_{k=1}^{n_B}\left(W_{k,\ell}\frac{\mathrm{d}}{\mathrm{d}W'_{c,b}} f_C\left(\sum_{j=1}^{n_c} C_j W'_{j,k}\right)\right)\right] \\
&= -\sum_{\ell=1}^{n_A}\left[(\alpha_\ell - A_\ell) f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,\ell}\right)\sum_{k=1}^{n_B}\left(W_{k,\ell} f'_C\left(\sum_{h=1}^{n_c} C_h W'_{h,k}\right)\frac{\mathrm{d}}{\mathrm{d}W'_{c,b}}\left(\sum_{j=1}^{n_c} C_j W'_{j,k}\right)\right)\right]
\end{aligned}
$$

$$\begin{aligned}
\varepsilon'_{c,b} &= -\sum_{\ell=1}^{n_A}\left[(\alpha_\ell - A_\ell)\, f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,\ell}\right) W_{b,\ell}\; f'_C\left(\sum_{h=1}^{n_c} C_h W'_{h,b}\right)\frac{\mathrm{d}}{\mathrm{d}W'_{c,b}}\left(C_c W'_{c,b}\right)\right] \\
&= -\sum_{\ell=1}^{n_A}\left[(\alpha_\ell - A_\ell)\, f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,\ell}\right) W_{b,\ell}\; f'_C\left(\sum_{h=1}^{n_c} C_h W'_{h,b}\right) C_c\right] \\
&= -\sum_{\ell=1}^{n_A}\left[(\alpha_\ell - A_\ell)\, f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,\ell}\right) W_{b,\ell}\right] f'_C\left(\sum_{h=1}^{n_c} C_h W'_{h,b}\right) C_c
\end{aligned}$$

En notant $\varepsilon'_b = \begin{bmatrix}\varepsilon_{1,b} & \cdots & \varepsilon_{n_C,b}\end{bmatrix} \in \mathcal{M}_{1,n_C}$, on a donc :

$$\begin{aligned}
\varepsilon'_b &= -\sum_{\ell=1}^{n_A}\left[(\alpha_\ell - A_\ell)\, f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,\ell}\right) W_{b,\ell}\right] f'_C\left(\sum_{h=1}^{n_c} C_h W'_{h,b}\right)\begin{bmatrix}C_1 & \cdots & C_{n_C}\end{bmatrix} \\
&= -\sum_{\ell=1}^{n_A}\left[(\alpha_\ell - A_\ell)\, f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,\ell}\right) W_{b,\ell}\right] f'_C\left(\sum_{h=1}^{n_c} C_h W'_{h,b}\right) C
\end{aligned}$$

Puis en notant $\varepsilon' = \begin{bmatrix}\varepsilon'_{1,1} & \cdots & \varepsilon'_{1,n_B} \\ \vdots & & \vdots \\ \varepsilon'_{n_C,1} & \cdots & \varepsilon'_{n_C,n_B}\end{bmatrix} = \begin{bmatrix}{}^{\mathrm{t}}\varepsilon'_1 & \cdots & {}^{\mathrm{t}}\varepsilon'_b\end{bmatrix} \in \mathcal{M}_{n_C,n_B}$, on en déduit que :

$$\begin{aligned}
\varepsilon' &= \left[-\sum_{\ell=1}^{n_A}\left[(\alpha_\ell - A_\ell)\, f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,\ell}\right) W_{1,\ell}\right] f'_C\left(\sum_{h=1}^{n_c} C_h W'_{h,1}\right){}^{\mathrm{t}}C \quad \cdots \quad -\sum_{\ell=1}^{n_A}\left[(\alpha_\ell - A_\ell)\, f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,\ell}\right) W_{n_B,\ell}\right] f'_C\left(\sum_{h=1}^{n_c} C_h W'_{h,n_B}\right){}^{\mathrm{t}}C\right] \\
&= -{}^{\mathrm{t}}C * \left[\sum_{\ell=1}^{n_A}\left[(\alpha_\ell - A_\ell)\, f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,\ell}\right) W_{1,\ell}\right] f'_C\left(\sum_{h=1}^{n_c} C_h W'_{h,1}\right) \quad \cdots \quad \sum_{\ell=1}^{n_A}\left[(\alpha_\ell - A_\ell)\, f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,\ell}\right) W_{n_B,\ell}\right] f'_C\left(\sum_{h=1}^{n_c} C_h W'_{h,n_B}\right)\right] \\
&= -{}^{\mathrm{t}}C * \left(\left[\sum_{\ell=1}^{n_A}\left[(\alpha_\ell - A_\ell)\, f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,\ell}\right) W_{1,\ell}\right] \quad \cdots \quad \sum_{\ell=1}^{n_A}\left[(\alpha_\ell - A_\ell)\, f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,\ell}\right) W_{n_B,\ell}\right]\right] \times \left[f'_C\left(\sum_{h=1}^{n_c} C_h W'_{h,1}\right) \quad \cdots \quad f'_C\left(\sum_{h=1}^{n_c} C_h W'_{h,n_B}\right)\right]\right) \\
&= -{}^{\mathrm{t}}C * \left[\left(\left[(\alpha_1 - A_1)\, f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,1}\right) \quad \cdots \quad \left(\alpha_{n_A} - A_{n_A}\right) f'_B\left(\sum_{i=1}^{n_B} B_i W_{i,n_A}\right)\right] * \begin{bmatrix} W_{1,1} & \cdots & W_{n_B,1} \\ \vdots & & \vdots \\ W_{1,n_A} & \cdots & W_{n_B,n_A}\end{bmatrix}\right) \times f'_C\left(\left[\sum_{h=1}^{n_c} C_h W'_{h,1} \quad \cdots \quad \sum_{h=1}^{n_c} C_h W'_{h,n_B}\right]\right)\right] \\
&= -{}^{\mathrm{t}}C * \left[\left(\left[(\alpha - A) \times f'_B\,(B * W)\right] * {}^{\mathrm{t}}W\right) \times f'_C\,(C * W')\right] \\
&= -{}^{\mathrm{t}}C * \left[\left(\delta_B * {}^{\mathrm{t}}W\right) \times f'_C\,(C * W')\right]
\end{aligned}$$

On posant $\delta_C = \left[\left(\delta_B * {}^{\mathrm{t}}W\right) \times f'_C\,(C * W')\right]$, on a $\varepsilon' = -{}^{\mathrm{t}}C * \delta_C$.

1.4. Correction des poids des précédentes couches

Pour une couche D ayant n_D neurones, la fonction d'activation f_D et les poids $W''_{d,c}$ $((d,c) \in [\![1, n_D]\!] \times [\![1, n_C]\!])$ entre les couches D et C, on peut montrer que :

$$\varepsilon'' = -\,{}^{\mathbf{t}}D * \left[\left(\delta_C * {}^{\mathbf{t}}W'\right) \times f'_D\left(D * W''\right)\right]$$

On posant $\delta_D = \left[\left(\delta_C * {}^{\mathbf{t}}W'\right) \times f'_D\left(D * W''\right)\right]$, on a $\varepsilon'' = -\,{}^{\mathbf{t}}D * \delta_D$.
De même, pour une couche E ayant n_E neurones, la fonction d'activation f_E et les poids $W'''_{e,d}$ $((e,d) \in [\![1, n_E]\!] \times [\![1, n_D]\!])$ entre les couches E et D, on peut montrer que :

$$\varepsilon''' = -\,{}^{\mathbf{t}}E * \left[\left(\delta_D * {}^{\mathbf{t}}W''\right) \times f'_E\left(E * W'''\right)\right]$$

On posant $\delta_E = \left[\left(\delta_D * {}^{\mathbf{t}}W''\right) \times f'_E\left(E * W'''\right)\right]$, on a $\varepsilon''' = -\,{}^{\mathbf{t}}E * \delta_E$.

2. Codage du perceptron multi-couches

2.1. Codage d'une couche

La classe « Layer » est construite en prenant en argument 2 chaînes de caractères (correspondant au nom de la couche et au nom de la fonction d'activation de la couche), un entier (correspondant au nombre de neurones de la couche) et un flottant (correspondant à l'amplitude initiale des poids synaptiques en sortie des neurones).

La fonction d'activation ne peut être que `sigmoid` ou `relu`, le nombre de neurones de la couche ne peut pas être nul et l'amplitude initiale des poids synaptiques doit être comprise dans l'intervalle]0, 1[.

La seule méthode de cette classe est la méthode `get()` qui permet de récupérer la couche sous forme de dictionnaire.

2.2. Codage d'un réseau

2.2.1. Constructeur

La classe « Network » est construite en prenant en argument une liste de dictionnaire. Chaque dictionnaire de cette liste correspond à une couche du réseau. Cette dictionnaire comprend donc le nombre de neurones de la couche, sa fonction d'activation (généralement la même pour toutes les couches du réseau) et l'amplitude initiale de l'intervalle dans lequel se situent les poids synaptiques à la sortie du réseau.

```
def __init__(self, *layers)
```

Le constructeur prend en argument les différentes couches et construit automatiquement le réseau associé. Voici les différentes étapes de cette construction.

```
self.nbLayers = len(layers)
```

On récupère le nombre de couches du réseau.

```
self.layers = [np.ones(layers[i]["dimension"] + (i == 0)) for i in
                        range(0, self.nbLayers)]
```

On construit une liste contenant les valeurs des neurones de chaque couche et en ajoutant un biais dans la première couche. Il s'agit donc d'une liste de listes (une liste = une couche) qui contient des flottants (un flottant = un neurone). Par défaut, la valeur des neurones est initialisé à 1.

```
self.activations = [layer["activation"] for layer in layers]
```

On construit une liste contenant les fonctions d'activation de chaque couche. Il s'agit donc d'une liste de chaînes de caractères (une chaîne = une fonction d'activation).

```
self.weights = []
    for i in range(self.nbLayers-1):
        matrix = np.random.random((self.layers[i].size, self.layers[i+1].size))
        self.weights.append(2*layers[i]["poids"]*matrix - layers[i]["poids"])
```

On construit une liste `self.weights` contenant les matrices des poids synaptiques. Il s'agit donc d'une liste de matrice (une matrice = une matrice des poids synaptiques entre 2 couches) contenant des flottants (un flottant = un poids synaptique entre 2 couches successives). Chaque poids est initialisé aléatoirement dans un intervalle donné, connu grâce aux dictionnaires passés en argument.

On commence donc par construire une matrice `matrix` dont le nombre de lignes correspond au nombre de neurones de la couche d'entrée et le nombre de colonnes correspond au nombre de neurones de la couche de sortie, en leur attribuant une valeur aléatoire entre 0 et 1.

Ensuite, on ajoute cette matrice à la liste `self.weights`, tout en ramenant les poids dans l'intervalle désiré : si l'amplitude est a, comme ma valeur x est dans l'intervalle $[0, 1]$, pour la ramener dans l'intervalle $[-a, a]$, il suffit de faire $[a - (-a)]\, x + (-a) = 2ax - a$.

2.2.2. Test du réseau

```
def test(self, inputs):
```

Cette méthode permet de tester le réseau, c'est-à-dire de calculer les sorties du réseau pour des entrées `inputs` données. Elle prend donc en argument une liste de valeurs d'entrée et retourne une liste contenant les valeurs de sortie calculées par le réseau en l'état actuel.

```
if len(inputs + [1.0]) != len(self.layers[0]):
    raise ValueError("Le nombre de valeurs d'entrée est
                     incompatible avec le réseau de neurones...")
```

On commence par vérifier que la liste d'entrées contient autant de valeurs qu'il y a de neurones dans la couche d'entrée de notre réseau. Mais, comme on ajoute un neurone pour le biais dans la couche d'entrée, on doit faire comme si on ajoutait un élément à la liste d'entrées pour faire le test. Si le nombre d'entrées ne correspond pas au nombre de neurones de la couche d'entrée, on lève une exception.

```
self.layers[0] = np.array(inputs + [1.0])
    for i in range(1, self.nbLayers):
        if self.activations[i-1] == "sigmoid":
            self.layers[i] = at.sigmoid(np.dot(self.layers[i-1], self.weights[i-1]))
        elif self.activations[i-1] == "relu":
            self.layers[i] = at.relu(np.dot(self.layers[i-1], self.weights[i-1]))
        else:
            raise ValueError("La fonction d'activation n'est pas définie")
```

Si le nombre d'entrées correspond au nombre de neurones de la couche d'entrée, les valeurs des neurones de la première couche prennent les valeurs des entrées passées en argument. Il ne faut pas oublier le biais, initialisé à 1.

Ensuite, on calcule les valeurs des neurones de chaque couche jusqu'à la couche de sortie, c'est-à-dire de la couche 2 à la couche `nbLayers`, avec une boucle `for`.

Pour cela, on récupère la fonction d'activation de la couche précédente. Si c'est la fonction `sigmoid` ou la fonction `relu`, on calcule les valeurs de la couche parcourue en multipliant les valeurs de la couche précédente par la matrice des poids synaptiques entre ces 2 couches, puis en « passant » le résultat dans la fonction d'activation. En effet, on a $A = f_B\,(B * W)$ où A correspond à la couche dont on veut déterminer les valeurs, B la

couche précédente, f_B la fonction de transfert de la couche B, et W la matrice des poids synaptiques entre les couches A et B.
Si la fonction d'activation n'est ni `sigmoid` ni `relu`, on lève une exception.

```
return self.layers[-1]
```

On finit en renvoyant les valeurs de la couche de sortie, calculé par le réseau avec les entrées passées en argument.

2.2.3. Entraînement du réseau

```
def train(self, data, iteration = 1000, N = 0.5):
```

Cette méthode permet d'entraîner un certain nombre de fois le réseau à partir des données fournies. Il prend donc en argument `data` correspondant aux données d'apprentissage (entrées et sorties attendues pour de telles entrées), `iteration` correspondant au nombre de fois où le réseau devra apprendre les `data` et `N` qui est le coefficient d'apprentissage. Par défaut, le nombre d'itérations est de 1000 et le coefficient d'apprentissage de 0,5.
Les données se présentent sous la forme d'une liste de listes (une liste = une échantillon de données) contenant 2 listes (une liste pour les données d'entrée et une liste pour les données de sortie).

```
for i in range(iteration):
    error = 0.0
    for d in data:
        inputs = d[0]
        targets = d[1]
        self.test(inputs)
        error = error + self.computation(targets, N)
    if ((i+1) % 100) == 0 :
        print("À l'itération", (i+1), "l'erreur est de : %-.5f" % error)
```

À chaque itération, on calcule l'erreur totale commise `error`, c'est-à-dire la somme des erreurs commises pour toutes les données d'apprentissage.
Pour calculer cette erreur, il suffit donc de parcourir l'ensemble des données d'apprentissage au moyen d'une boucle `for`.
Pour chaque échantillon de données parcouru, on récupère dans `inputs` la première liste correspondant aux entrées et dans `outputs` la seconde liste correspondant aux sorties attendues.
Ensuite, on calcule les sorties véritables obtenus lorsque l'actuel réseau reçoit les `inputs` *via* la méthode `test`.
Cela permet de récupérer l'erreur quadratique $\mathcal{E}$ commise *via* la méthode `computation` qui permet également de modifier les poids synaptiques afin d'améliorer le réseau.
On peut alors ajouter l'erreur quadratique $\mathcal{E}$ à l'erreur totale `error`.
Toutes les 100 itérations, on affiche `error`, afin d'avoir un aperçu de l'évolution d'apprentissage du réseau.

2.2.4. Rétro-propagation du gradient

```
def computation(self, targets, N):
```

Cette méthode permet d'une part de calculer l'erreur commise $\mathcal{E}$ pour des entrées données et mettre en place la rétro-propagation du gradient afin de réajuster les poids synaptiques entre chaque couche du réseau et donc de minimiser $\mathcal{E}$ (*cf.* 1. Théorie).
Elle prend en argument la sortie visée `targets` et le coefficient d'apprentissage `N`, et renvoie l'erreur commise $\mathcal{E}$.

```
if len(targets) != len(self.layers[-1]):
    raise ValueError("Le nombre de valeurs de sortie est
                      incompatible avec le réseau de neurones...")
```

On commence par vérifier que la liste contenant les valeurs de sortie visées a autant de valeurs qu'il y a de neurones dans la couche de sortie de notre réseau. Si le nombre de sorties ne correspond pas au nombre de neurones de la couche de sortie, on lève une exception.

```
deltas = list()
```

On crée une liste `deltas` destinée à contenir les « δ » calculés pour chaque couche.

```
if self.activations[-2] == "sigmoid":
    de = (targets - self.layers[-1]) * at.dsigmoid(self.layers[-1])
elif self.activations[-2] == "relu":
    de = (targets - self.layers[-1]) * at.drelu(self.layers[-1])
deltas.append(de)
```

On commence par calculer le « δ » de l'avant-dernière couche. D'après ce qu'on a vu précédemment dans la partie 1. Théorie, il est égal à $\left[(\alpha - A) \times f'_B(B * W)\right]$ où α est la liste des sorties visées, A la liste des sorties véritables, f'_B la dérivée de la fonction d'activation de l'avant-dernière couche, B les valeurs des neurones de l'avant-dernière couche et W la matrice des poids synaptiques entre la dernière et l'avant-dernière couche.
On a $A = f_B(B * W)$ et on veut $f'_B(B * W)$. Or, comme pour la fonction sigmoïde S, $S'(t) = S(t)[1 - S(t)]$, et que pour la fonction ReLU R, $R'(t) = 1$ si $R(t) > 0$ et 0 sinon, on peut donc bien exprimer $f'_B(B * W)$ en fonction de A.
Ici, α correspond à `targets` la liste des valeurs de sortie visées, A à `self.layers[-1]` la liste des valeurs de sortie calculées par le réseau, et $f'_B(B * W)$ à `dsigmoid(self.layers[-1])` ou `drelu(self.layers[-1])`.
Pour finir, on ajoute le « δ » de l'avant-dernière liste à la liste `deltas`, car comme on l'a vu, les « δ » des couches suivantes s'expriment en fonction des « δ » calculés précédemment.

```
for i in range(self.nbLayers-2, 0, -1):
    if self.activations[i-1] == "sigmoid":
        de = np.dot(deltas[-1], self.weights[i].T) * at.dsigmoid(self.layers[i])
    elif self.activations[i-1] == "relu":
```

```
        de = np.dot(deltas[-1], self.weights[i].T) * at.drelu(self.layers[i])
    deltas.append(de)
```

De même que pour l'avant-dernière couche, on calcule le « δ » de toutes les couches jusqu'à la première. D'après la théorie vu dans la partie 1. pour connaître le « δ_C » d'une couche C sachant le « δ_B » d'une couche B, il suffit de faire $\left[\left(\delta_B * {}^{t}W\right) \times f'_C\left(C * W'\right)\right]$ avec W la matrice des poids synaptiques entre la couche B et la couche suivant B, f'_C la fonction d'activation de la couche C, C la liste contenant les valeurs de la couche C et W' la matrice des poids synaptiques entre les couches B et C.

Et de même que pour l'avant-dernière couche, on peut exprimer $f'_C\left(C * W'\right)$ en fonction de B où $B = f_C\left(C * W'\right)$ la liste des valeurs de la couche B.

Ici, δ_B correspond à `deltas[-1]` le dernier « δ » calculé, W à `self.weights[i]` la matrice des poids synaptiques entre la couche `i` (dans l'exemple précédent, cela correspond à la couche C) et la couche `i+1` (couche B de l'exemple), et $f'_D\left(D * W'\right)$ à `dsigmoid(self.layers[i])` ou `drelu(self.layers[i])` où `self.layers[i]` correspond aux valeurs des neurones de la couche `i` (couche C de l'exemple).

Pour finir, on ajoute le « δ » (δ_C de l'exemple) à la liste `deltas`. Puis on calcule de « δ » de la couche précédente en s'appuyant sur le « δ » qu'on vient de calculer.

```
deltas.reverse()
```

Une fois qu'on a calculé tous les « δ », comme on les a calculé de l'avant-dernière couche à la première couche, il s'agit de retourner la liste afin qu'elle soit dans le bon sens.

```
for i in range(len(self.weights)):
    change = np.dot(np.array([self.layers[i]]).T, np.array([deltas[i]]))
    self.weights[i] += N * change
```

Pour chaque matrice de poids synaptiques, on met à jour les valeurs afin de minimiser $\mathcal{E}$. Comme on l'a vu précédemment, pour une matrice de poids synaptiques W donnée entre 2 couches A et B, la nouvelle matrice de poids est égale à l'ancienne matrice des poids à laquelle on ajoute ${}^{t}B * \delta_B$ multiplié par le coefficient d'apprentissage `N`.

Ici, W correspond à `self.weights[i]`, B correspond `self.layers[i]` et δ_B correspond à `deltas[i]`.

```
error = 0.0
for i in range(len(targets)):
    error = error + 0.5 * (targets[i] - self.layers[-1][i])**2
return error
```

On calcule également l'erreur quadratique $\mathcal{E}$ commise, et on la renvoie.

2.3. Codage des fonctions d'activation

Dans ce projet, on utilise 2 fonctions d'activations : la fonction sigmoïde et la fonction ReLU. Ces 2 fonctions sont définies à part dans une bibliothèque qu'il possible de récupérer. Pour cela, il faut importer la bibliothèque dans le réseau en faisant `import activation as at`.

2.3.1. Fonction sigmoïde

La fonction sigmoïde est définie par $S : t \mapsto \frac{1}{1-\exp(-t)}$.

Elle est définie sur $\mathbb{R}$.

Elle est donc dérivable sur $\mathbb{R}$ et sa dérivée est définie par $S' : t \mapsto \frac{\exp(-t)}{(1-\exp(-t))^2} = S(t)(1 - S(t))$.

Donc si $A = f_B(B * W)$, alors $f'_B(B * W) = A(1 - A)$ où 1 est une matrice de même dimension que A et dont tous les coefficients valent 1.

On peut donc définir `sigmoid` et `dsigmoid` de la sorte :

```
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def dsigmoid(y):
    return y * (1.0 - y)
```

2.3.2. Fonction ReLU

La fonction ReLU est définie par $R : t \mapsto \begin{cases} t & \text{si } t > 0 \\ 0 & \text{si } t \leqslant 0 \end{cases}$.

Elle est définie sur $\mathbb{R}$.

Elle est donc dérivable sur $\mathbb{R}$ et sa dérivée est définie par $R' : t \mapsto \begin{cases} 1 & \text{si } t \geqslant 0 \\ 0 & \text{si } t < 0 \end{cases} = \begin{cases} 1 & \text{si } R(t) > 0 \\ 0 & \text{si } R(t) = 0 \end{cases}$.

On peut donc définir `relu` et `drelu` de la sorte :

```
def relu(x):
    return x * (x > 0)

def drelu(y):
    return 1.0 * (y > 0)
```