Rapport de TIPE
Étude des réseaux de neurones artificiels

I Notions théoriques

Les notions développées ici sont issues de la théorie générale des réseaux de neurones artificiels. La première approche de notre TIPE fut d'appréhender cette théorie et de nous l'approprier. Il a ainsi fallu définir formellement les objets que nous allions étudier et mettre au point un lexique. Une grande part de ces définitions sont évidemment inspirées de travaux antérieurs de chercheurs, mais nous nous sommes permis de choisir nos propres notations, de restreindre certaines définitions aux seuls cas que nous allons étudier et de formaliser par nous-mêmes quand la théorie antérieure n'offrait aucune formalisation.

Même si l'aspect expérimental doit prévaloir au sein d'un TIPE, nous avons décidé de rendre compte dans la première partie de ce rapport de cette étude théorique. En effet, elle est indispensable pour mettre en place le vocabulaire que nous utiliserons pour le rédiger et pour fixer le cadre de notre étude expérimentale. De plus, les classes que nous avons implémentées en Python sont grandement inspirées de cette théorie et l'ont beaucoup inspirée, l'aspect mathématique et l'aspect informatique ayant été pensés en même temps. Il nous parait donc pertinent d'associer l'un à l'autre dans ce document.

(A) Neurone formel

1- Définition mathématique

On note ici $(\cdot|\cdot)$ le produit scalaire canonique de le $\mathbb{R}$-espace vectoriel $\mathbb{R}^n$.

On a : $\forall (x_1,\dots,x_n),(y_1,\dots,y_n)\in\mathbb{R}^n,\ \big((x_1,\dots,x_n)\big|(y_1,\dots,y_n)\big)=\sum_{i=1}^{n} x_i y_i$.

Définition

Soient $n\in\mathbb{N}^*$ et $E\in\mathscr{P}(\mathbb{R})\setminus\{\varnothing\}$.
On considère $\phi\in\mathbb{R}^{\mathbb{R}}$, $b\in\mathbb{R}$ et $\vec{w}=(w_1,\dots,w_n)\in\mathbb{R}^n$.
Un n-**neurone formel** est une application $f : E^n \longrightarrow \mathbb{R}$ définie par :

$$\forall \vec{x}=(x_1,\dots,x_n)\in E^n\ ;\ f(\vec{x})=\phi\big(b+(\vec{x}|\vec{w})\big)$$

- n est **l'ordre du neurone**, noté ordre(f).
- ϕ est appelée **fonction d'activation**.
- $\vec{w}$ est **le vecteur des poids synaptiques**.
- b est **le biais**.

Remarque

Tout n-neurone est entièrement déterminé par la donnée du quadruplet $(E,b,\vec{w},\phi)$, l'ordre étant donné par le nombre de composantes de $\vec{w}$. On notera par la suite $f=\text{neur}(E,b,\vec{w},\phi)$.

Il n'y a pas unicité du quadruplet associé à un neurone. On verra plus tard des exemples de neurones égaux (mêmes ensemble de départ et mêmes graphes) sans que leurs quadruplets associés le soient.

Exemples

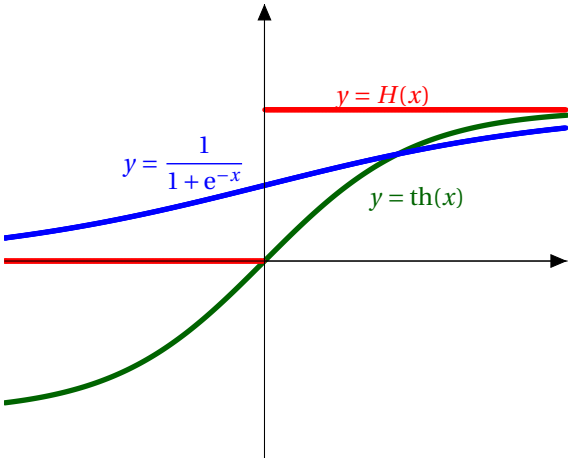
- Les espaces de travails classiques sont : $E=\text{Im}(f)=\mathbb{R}$; $E=\text{Im}(f)=\{0;1\}$ (utile lorsque l'on travaille avec des booléens) ; $E=\text{Im}(f)=[0;1]$.
- Les fonctions d'activations classiques sont :
  - la fonction indicatrice d'un intervalle $J=[a,+\infty[$ où $a\in\mathbb{R}$: $\mathbb{1}_J$;
  - la fonction th ;

– la fonction d'Heaviside $H : \left| \begin{array}{ccc} \mathbb{R} & \longrightarrow & \{0;1\} \\ x & \longmapsto & \begin{cases} 0 \text{ si } x < 0 \\ 1 \text{ si } x \geqslant 0 \end{cases} \end{array} \right.$;

– la fonction sigmoïde $\left| \begin{array}{ccc} \mathbb{R} & \longrightarrow & [0;1] \\ x & \longmapsto & \dfrac{1}{1+\mathrm{e}^{-x}} \end{array} \right.$.

Voici leurs graphes :

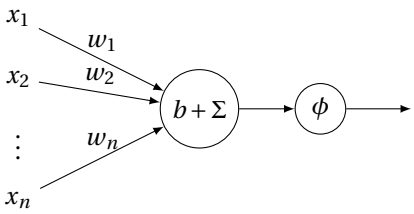
FIGURE 1 – Représentation graphique des fonctions d'activation usuelle



2- Représentation graphique

Soit $f = \text{neur}(E, F, b, \vec{w}, \phi)$ un n-neurone formel. On représente graphiquement ce neurone de la façon suivante :

FIGURE 2 – Représentation graphique d'un neurone formel



B Perceptron et séparabilité linéaire

1- Définition du perceptron

Définition

Soient $n \in \mathbb{N}^*$, $E \in \mathscr{P}(\mathbb{R}) \setminus \{\varnothing\}$, $\theta \in \mathbb{R}$ et $\vec{w} \in \mathbb{R}^n$. On note $H = \mathbb{1}_{\mathbb{R}_+}$ la fonction d'Heaviside.
On appelle **n-perceptron** tout neurone formel de la forme $P = \text{neur}\left(E, -\theta, \vec{w}, H\right)$.
θ est appelé **seuil d'activation**.

Remarques

i. Le seuil d'activation est l'opposé du biais dans la définition générale du neurone formel.

ii. Nous avons ici bouleversé les notations car le perceptron peut être considéré avec une vision totalement dif-

férente de celle qu'on porte sur les autres neurones. On peut comprendre son fonctionnement de la manière suivante.

Appliquer le perceptron P à un vecteur $\vec{x} \in E^n$ revient à comparer le produit scalaire $(\vec{x}|\vec{w})$ à un seuil θ. On a :

$$P(\vec{x}) = \begin{cases} 0 \text{ si } (\vec{x}|\vec{w}) < \theta \\ 1 \text{ si } (\vec{x}|\vec{w}) \geqslant \theta \end{cases}$$

Ce nouveau point de vue est adapté à la compréhension du phénomène de séparabilité linéaire.

iii. Un perceptron est entièrement déterminé par la donnée de E, θ et $\vec{w}$. On notera $P = \text{perc}(E, \theta, \vec{w})$.

2- Notion de séparabilité linéaire

Définition

Soit $n \in \mathbb{N}^*$. On considère B une partie non-vide de $\mathbb{R}^n$ et $\{a, b\}$ un ensemble à deux éléments. Soit l'application $f : B \longrightarrow \{a; b\}$.

B est **linéairement séparable** par f si et seulement s'il existe un hyperplan affine $\mathscr{H}$ de $\mathbb{R}^n$, d'équation $(\vec{w}|\vec{x}) = \theta$ ($\vec{w} \in \mathbb{R}^n$ et $\theta \in \mathbb{R}$ fixés), tel que $f^{-1}(\{a\})$ et $f^{-1}(\{b\})$ soient situés de part et d'autre de $\mathscr{H}$:

$$\begin{cases} \forall \vec{x} \in f^{-1}(\{a\}) \; ; \; (\vec{w}|\vec{x}) < \theta \\ \forall \vec{x} \in f^{-1}(\{b\}) \; ; \; (\vec{w}|\vec{x}) \geqslant \theta \end{cases}$$

On dit que f est un **sépérateur linéaire** de B et $\mathscr{H}$ un **hyperplan de séparation**.

Remarques

i. B peut représenter concrètement une base de données codée sous forme de tuples que la fonction f classe en deux catégories a et b, le plus souvent 0 et 1.

ii. *A priori*, l'hyperplan de séparation n'est pas unique.

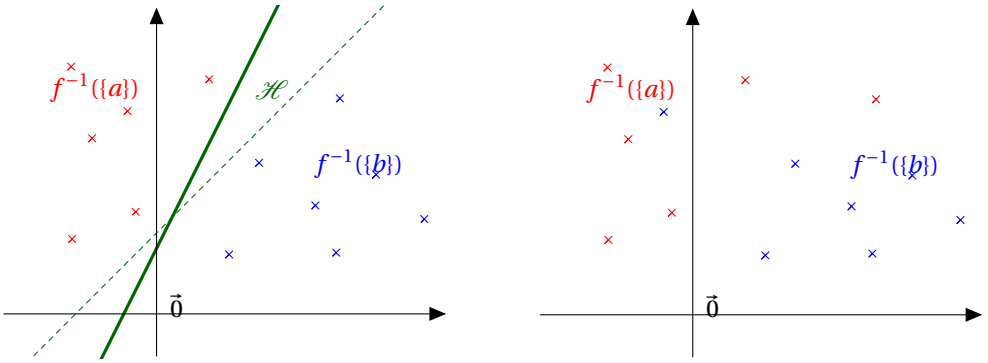
iii. On peut définir f ainsi : $\forall \vec{x} \in B \; ; \; f(\vec{x}) = \begin{cases} a \text{ si } (\vec{w}|\vec{x}) < \theta \\ b \text{ si } (\vec{w}|\vec{x}) \geqslant \theta \end{cases}$

Exemples dans $\mathbb{R}^2$

i. On représente dans le plan un nuage de points répartis en deux catégories linéairement séparables : a (colorés en rouge) et b (colorés en bleu). On a représenté en vert l'hyperplan $\mathscr{H}$ de séparation (ici une droite) et en pointillés un autre hyperplan possible.

ii. On représente dans le plan un nuage de points répartis en deux catégories qui ne sont pas linéairement séparables : a (colorés en rouge) et b (colorés en bleu). On "voit" bien que les deux nuages ne sont pas séparables par une droite.

FIGURE 3 – Séparabilité et non-séparabilité linéaire



3- Lien avec le perceptron

Propriété

Soient E une partie non-vide de $\mathbb{R}$ et $n \in \mathbb{N}^*$. On pose $\theta \in \mathbb{R}$ et $\vec{w} \in \mathbb{R}^n$.

- Un perceptron $P = \text{perc}(E, \theta, \vec{w})$ d'ordre n est un séparateur linéaire de E^n.
- Réciproquement, si E^n est linéairement séparable pour $f : E^n \longrightarrow \{0;1\}$ alors f est un perceptron.

Démonstration

- Par définition du perceptron, $P^{-1}(\{0\})$ et $P^{-1}(\{1\})$ sont de part et d'autre de l'hyperplan d'équation $(\vec{w}|\vec{x}) = \theta$.
- Si l'équation de l'hyperplan séparateur est $(\vec{\omega}|\vec{x}) = \tau$, alors on a $f = \text{perc}(E, \tau, \vec{\omega})$.

■

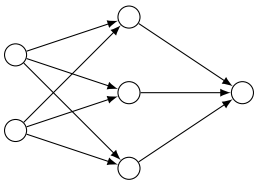
Remarque

Cette propriété signifie que toute fonction correspondant à un séparateur linéaire pourra être construite sous la forme d'un perceptron. Il sera donc impossible pour un perceptron d'être paramétré pour modéliser une fonction qui n'est pas un séparateur linéaire.

C Réseau de neurones formels

Les neurones formels et leur théorie s'inspirent des cerveaux réels. Ainsi, de manière intuitive, un réseau de neurones est un ensemble de neurones connectés entre-eux et parcourus par un flux d'informations. On se le représente aisément sous la forme d'un graphe :

FIGURE 4 – Vision intuitive d'un réseau de neurones formels



Les réseaux de neurones que nous considérerons seront disposés par colonnes, qui sont appelés couches et disposées dans un certain ordre, de façon à ce que chaque neurone d'une couche donnée soient reliés à tous les neurones de la couche suivante. De plus, on n'a pas de connexion possible avec des neurones d'une couche antérieure.

Ces restrictions ne sont pas imposées par la théorie générale des réseaux de neurones mais elles nous permettrons de simplifier notre étude théorique et nos expérimentations. Nous allons à présent définir formellement les réseaux de neurones que nous allons utiliser, avec les restrictions précédemment évoquées.

1- Couche de neurones

Définition

Soient $p \in \mathbb{N}^*$, $C = \left(N_e = \text{neur}\left(E_e, b_e, \vec{w}_e, \phi_e\right)\right)_{e \in [\![1;p]\!]}$ une famille non-vide de neurones formels.
On dit que C est une **couche de neurones** si C vérifie les propriétés suivantes :

i. **Uniformité de l'ordre**

Tous les neurones de C ont le même ordre : $\forall N_e, N_f \in C,\ \text{ordre}(N_e) = \text{ordre}(N_f)$

ii. **Uniformité des ensembles**

Tous les neurones de C sont définis sur le même ensemble : $\forall N_e, N_f \in C,\ E_e = E_f$

iii. **Uniformité des fonctions d'activation**

Tous les neurones de C ont la même fonction d'activation : $\forall N_e, N_f \in C,\ \phi_e = \phi_f$

Remarques

i. Soit $\mathscr{F} = (x_i)_{i\in I}$ une famille. On note ici abusivement $x \in \mathscr{F}$ s'il existe $i \in I$ tel que $x = x_i$.

ii. De manière plus claire, on peut voir une couche de neurones comme une famille de neurones identiques, aux valeurs des biais et des vecteurs synaptiques près.

2- Réseau de neurones

Définition

Soient $q \in \mathbb{N}^*$ et $\mathscr{R} = (C_i)_{1\leqslant i\leqslant q}$ une famille de couches de neurones.
$\mathscr{R}$ est un **réseau de neurones** si $q = 1$, ou si $q \geqslant 2$ et $\mathscr{R}$ vérifie les propriété :

i. **Compatibilité des couches**

L'ensemble d'arrivé d'une neurone donné est contenu dans l'ensemble de départ des neurones de la couche suivante :

$\forall i \in [\![1; q-1]\!],\ \forall N_i \in C_i,\ \forall N_{i+1} \in C_{i+1}\ ;\ \mathrm{Im}(N_i) \subset E_{i+1}$

ii. **Uniformité des fonctions d'activation**

Tous les neurones du réseau ont la même fonction d'activation :

$\forall i \in [\![1; q-1]\!],\ \forall N_i \in C_i,\ \forall N_{i+1} \in C_{i+1}\ ;\ \phi_i = \phi_{i+1}$

iii. **Lien entre taille et ordre**

La taille d'une couche est l'ordre des neurones de la couche suivante :

$\forall i \in [\![1; q-1]\!],\ \forall N_{i+1} \in C_{i+1}\ ;\ \mathrm{taille}(C_i) = \mathrm{ordre}(N_{i+1})$

où $N_i = \mathrm{neur}(E_i, b_i, \vec{w}_i, \phi_i)$ et $N_{i+1} = \mathrm{neur}(E_{i+1}, b_{i+1}, \vec{w}_{i+1}, \phi_{i+1})$.

Remarques

i. Si $\mathscr{F} = (x_i)_{i\in I}$ est une famille finie, on pose $\mathrm{taille}(\mathscr{F}) = \mathrm{card}(I)$. De fait, la taille d'une couche est le nombre de neurones (éventuellement non-tous distincts) qu'elle contient.

ii. La propriété (ii) signifie qu'un cerveau formel est constitué de neurones ayant tous la même fonction d'activation.

iii. Les propriétés (i) et (iii) servent à composer les différents neurones en tant qu'applications. Considérons la couche C_i. Chaque neurone donne en sortie un scalaire. Si on concatène tous ces scalaires, on obtient un $\big(\mathrm{taille}(C_i)\big)$-uplet : la propriété (iii) dit que ce vecteur aura la bonne taille pour être mis en entrée des neurones de la couche C_{i+1} ; la propriété (i) que le vecteur est bien dans le domaine de définition des neurones de la couche C_{i+1}.

D Cerveau formel

1- Fonction induite par une couche

Définition

Soient $p \in \mathbb{N}^*$ et $C = \big(N_e = \mathrm{neur}\big(E, F, b_e, \vec{w}_e, \phi\big)\big)_{e\in[\![1;p]\!]}$ une couche de neurones formels d'ordre n.
On appelle fonction induite par la couche C la fonction vectorielle définie par :

$$\widetilde{C} : \left| \begin{array}{ccc} E^n & \longrightarrow & F^p \\ \vec{x} & \longmapsto & \left(N_1(\vec{x}), \dots, N_p(\vec{x})\right) \end{array} \right.$$

2- Cerveau formel

Définition

Soit $\mathscr{R} = (C_i)_{i \in [\![1;q]\!]}$ un réseau de neurones, de fonctions induites $\left(\widetilde{C_i}\right)_{i \in [\![1;q]\!]}$.

On a appelle **cerveau formel**, noté $\mathscr{C} = \text{cerv}(\mathscr{R})$, la fonction définie par :

$$\text{cerv}(\mathscr{R}) = \widetilde{C_p} \circ \cdots \circ \widetilde{C_1}$$

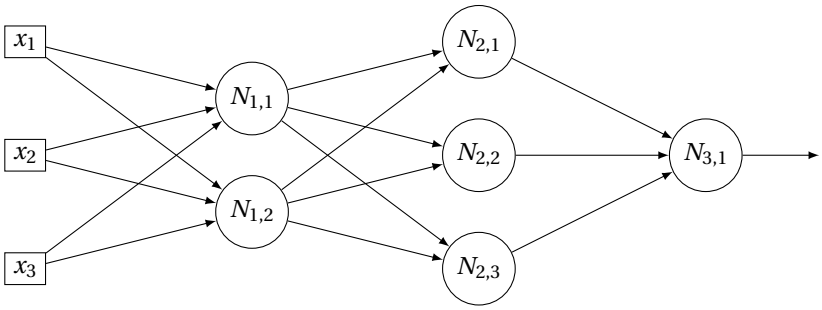
Remarque

La fonction est bien définie d'après les axiomes définissant les couches de neurones et les réseaux de neurones formels.

3- Représentation graphique

Pour représenter graphiquement un cerveau formel $\mathscr{C}$, On représente d'abord les entrées par des carrés disposées sur une colonne. On représente alors chaque neurone par un cercle. On aligne les cercles en colonne, couche par couche. Les couches sont quant à elles disposées dans l'ordre de leur indexation. On représente les entrées par des carrés.

FIGURE 5 – Représentation graphique d'un cerveau formel



II Apprentissage des réseaux de neurones

L'utilité des réseaux de neurones formels réside dans l'apprentissage. Il s'agit de paramétrer de manière automatisée le réseau pour qu'il réponde à une exigence fixée par l'utilisateur. Pour cela, on entre dans la mémoire de la machine un certain nombre d'exemples de données qui peuvent être mises en entrée du réseau et des sorties qui sont alors attendues. L'objectif est donc de trouver une méthode pour mettre à profit ces exemples pour paramétrer le réseau de manière optimale.

A Lexique

1- Modèle d'apprentissage

Définition

Soient $E \in \mathscr{P}(\mathbb{R})$, $n \in \mathbb{N}^*$ et $\mathscr{C}_P : E^n \longrightarrow \mathbb{R}^p$ un cerveau formel où P représente les paramètres du cerveau (biais et vecteurs synaptiques de chaque neurone).

On appelle **modèle** ou **base d'apprentissage** de $\mathscr{C}_P$ toute application $M : B \longrightarrow \mathbb{R}^p$ où B est une partie finie non-vide de E^n.

Remarques

i. M représente le modèle que l'on fournit au programme pour qu'il se paramètre de manière à répondre aux exigences du problème posé. L'idéal est *a priori* d'avoir : $\mathscr{C}_{P|B} = M$. Néanmoins nous verrons que dans la pratique, ceci peut être dû à phénomène de sur-apprentissage qu'il faut éviter.

ii. Un modèle $M : B \to F^p$ est implémenté comme une liste de couples $(x, M(x))$ où x parcourt B.

2- Fonction de coût

Définition

Avec les notations précédentes, on appelle **fonction de coût** du cerveau formel la fonction :

$$\delta : P \longmapsto \frac{1}{2} \sum_{\vec{x} \in B} \|\mathscr{C}_P(\vec{x}) - M(\vec{x})\|_2{}^2$$

Remarques

i. $\|\cdot\|_2$ désigne la norme euclidienne canonique sur $\mathbb{R}^n$: $\forall \vec{x} \in \mathbb{R}^n$, $\|\vec{x}\|_2 = \sqrt{(\vec{x}|\vec{x})}$. On choisit cette norme car il nous sera nécessaire dans la suite de pouvoir dériver δ, et les autres normes usuelles présentent des problèmes de dérivabilité qui n'existent pas avec $\|\cdot\|_2{}^2$.

ii. Le coefficient $\frac{1}{2}$ sert à simplifier le calcul des dérivées vu que dériver $x \mapsto x^2$ fait apparaitre des facteurs 2.

iii. Cette fonction prend une valeur d'autant plus faible que $\mathscr{C}_P$ et M ont des valeurs proches sur B. C'est en ça que δ va nous permettre de mesurer l'erreur commise par le cerveau.

3- Algorithme d'apprentissage

Définition

On appelle **algorithme d'apprentissage** tout algorithme prenant en entrée un cerveau formel $\mathscr{C}_P$ et un modèle d'apprentissage M, et modifiant sur le place le cerveau, i.e. en modifiant P, de façon à **minimiser l'écart entre les réponses données par le cerveau et les réponses attendues**, notamment en minimisant la fonction de coût.

Remarque

L'idéal est d'avoir à la fin de l'apprentissage $\delta(P) = \min \delta$, s'il existe. Mais dans la pratique, on ne peut trouver qu'un minimum local.

B Algorithme de WIDROW-HOFF

1- Principe de l'algorithme

Ce premier algorithme d'apprentissage est exclusivement réservé à la modification d'un perceptron $\mathscr{P}$. Son principe est relativement simple. On donne en entrée de l'algorithme un neurone, un modèle d'apprentissage et un certain coefficient α. Un compteur parcours l'indexation du modèle.

Pour chaque élément du modèle $(\vec{x}, M(\vec{x}))$, on va ensuite modifier un à un les composantes du vecteur des poids synaptiques de la façon suivante : si w_i est l'ancienne valeur, alors la nouvelle valeur sera $w_i + \alpha \times x \times (M(\vec{x}) - P(\vec{x}))$ où x_i est la composante de l'entrée $\vec{x}$ correspondant à ce synapse. Quand au biais b, il prend la valeur $b + \alpha \times (M(\vec{x}) - P(\vec{x}))$.

Nous verrons que le coefficient α, dont la valeur est typiquement comprise sur le segment $[0\ ;\ 1]$, influence la vitesse de convergence de l'algorithme mais aussi son bon fonctionnement.

2- Description de l'algorithme

Algorithme 1 : WIDROW-HOFF

Entrées :

- $N = \text{perc}(E, -b, \vec{w})$ un perceptron ;
- $M : B \longrightarrow \text{Im}(M)$ un modèle où B est un ensemble fini non-vide de tuples.
- $\alpha > 0$ le coefficient pondérant l'apprentissage.

début
  pour $\vec{x} \in B$ **faire**
    $s \leftarrow N(\vec{x})$
    $\Delta \leftarrow \alpha \times (M(\vec{x}) - s)$
    $b \leftarrow b + \Delta$
    pour $i = 1$ **à** taille$(\vec{w})$ **faire**
      $[\vec{w}]_i \leftarrow [\vec{w}]_i + \Delta \times [\vec{x}]_i$
    fin
  fin
fin

C Algorithme de rétro-propagation du gradient

1- Notion de descente du gradient

On considère, pour $n \in \mathbb{N}^*$, l'espace vectoriel $\mathbb{R}^n$. On se donne $\mathscr{D}$ un domaine ouvert non-vide de $\mathbb{R}^n$. On veut étudier la fonction $f : \mathscr{D} \longrightarrow \mathbb{R}$, choisie de façon à être de classe $\mathscr{C}^1$. On notera $\overrightarrow{\text{grad}}\, f$ le gradient de f et pour $i \in [\![1\,;\,n]\!]$, $\dfrac{\partial f}{\partial x_i}$ la dérivée partielle de f par rapport à $i^{\text{ème}}$ composante dans la base canonique de $\mathbb{R}^n$.

Problème : Trouver, s'il existe, un point où f atteint son minimum.

On suppose que f est une fonction trop compliquée pour déterminer formellement où se situe son minimum. On va donc se servir d'une méthode de recherche numérique qui se sert d'une propriété du gradient. On se contentera ici d'exprimer cette propriété de manière informelle.

Propriété

Soit $\vec{x}_0 \in \mathscr{D}$. Le vecteur $\overrightarrow{\text{grad}}\, f(\vec{x}_0)$ est tel que sa direction et son sens soient ceux à suivre pour obtenir la plus forte augmentation de f à partir de sa valeur en x_0.

Donc pour trouver le minimum de f quand on est placé en $\vec{x}_0$, on va se placer en un point situé sur la demi-droite affine $\vec{x}_0 - \mathbb{R}_+ \overrightarrow{\text{grad}}\, f(\vec{x}_0)$, c'est à dire à l'opposé de cette forte augmentation. En pratique, on se placera en le point $\vec{x}_1 = \vec{x}_0 - \alpha \overrightarrow{\text{grad}}\, f(\vec{x}_0)$, α étant un paramètre strictement positif déterminé empiriquement. On peut, en itérant ce calcul, progresser suivant une suite de points $(\vec{x}_i)_i$ pour se rapprocher du minimum, avec $\overrightarrow{x_{i+1}} = \vec{x}_i - \alpha \overrightarrow{\text{grad}}\, f(\vec{x}_i)$.

Remarques

i. Un critère d'arrêt classique pour la fonction est d'attendre que la norme du gradient passe en-dessous d'un certain seuil $\epsilon > 0$, mais ici nous n'en aurons pas besoin.

ii. Cette méthode présente un défaut important car elle peut tout aussi bien permettre de déterminer un minium local que global, sans possibilité de s'en rendre compte *a priori*.

iii. Le choix de α est compliqué car s'il est trop grand, on risque de ne pas pour pouvoir situer précisément le minimum et s'il est trop petit, l'algorithme sera excessivement lent à l'exécution. Une solution consisterait à faire évoluer le pas en fonction de l'avancée de la recherche du minimum, mais nous ne le ferons pas ici.

2- Principe de l'algorithme

Avec les notations précédemment utilisées, on se donne un modèle $M : B \longleftarrow \mathbb{R}^p$ et un cerveau formel $\mathscr{C}_P : E^n \longrightarrow \mathbb{R}^p$ tel que la fonction d'activation des neurones de $\mathscr{C}_p$ soit l'application définie sur $\mathbb{R}$ par :

$$\phi : \left| \begin{array}{ccc} \mathbb{R} & \longrightarrow & \mathbb{R} \\ x & \longmapsto & \dfrac{1}{1+\mathrm{e}^{-x}} \end{array} \right. .$$

Son graphe a été représenté dans la section I.

Propriété

On pose $\mathscr{N}$ la somme du nombre total de composantes des vecteurs de poids synaptique et du nombre total de biais des neurones du cerveau, c'est-à-dire le taille du paramétrage P (qui correspond au nombre de degrés de libertés du réseau).

La fonction de coût δ associée au cerveau $\mathscr{C}_P$ est de classe $\mathscr{C}^1$ sur $\mathbb{R}^{\mathscr{N}}$.

Démonstration

Montrons que pour $\vec{x} \in B$ fixé, $P \longmapsto \|\mathscr{C}_P(\vec{x}) - M(\vec{x})\|_2{}^2$ est de classe $\mathscr{C}^1$. Par addition, δ le sera aussi.

Pour un poids $\vec{w}$ et un biais b fixés, la fonction $(\vec{w}, b) \mapsto (\vec{x}|\vec{w}) + b$ est de classe $\mathscr{C}^1$. Donc par composition par ϕ elle aussi de classe $\mathscr{C}^1$, chaque neurone est une fonction de classe $\mathscr{C}^1$ en ses paramètres. Toujours par composition, on a que $\mathscr{C}_P$ est une fonction de classe $\mathscr{C}^1$ par rapport à P. Vu que $\|\cdot\|_2{}^2$ est aussi de classe $\mathscr{C}^1$, on en déduit facilement que $P \longmapsto \|\mathscr{C}_P(\vec{x}) - M(\vec{x})\|_2{}^2$ est de classe $\mathscr{C}^1$.

■

Nous allons donc pouvoir appliquer l'algorithme du gradient à δ pour chercher son point d'atteinte du minimum et ainsi trouver un paramétrage optimal pour le cerveau $\mathscr{C}_P$.

III Implémentation en Python

Ⓐ Création des classes et des méthodes

1- Classe `Genome`

Lorsque l'on considère notre étude théorique des cerveaux formels et les cadres auxquels nous nous sommes limités, nous constatons la nécessité d'obliger l'utilisateur à créer des neurones homogènes pour leurs fonctions d'activation. Pour cela, nous implémentons une classe appelée `Genome` qui servira à implémenter les neurones et les cerveaux en fixant la fonction d'activation qu'ils utilisent.

Programme

```
class Genome(object):
  """ Classe correspondant a la fonction d'activation, permettant de verifier
      que tous les neurones ont la meme fonction d'activation. """
  def __init__(self, nom_genome ="Heaviside",
               activation =lambda x: 0 if (x < 0) else 1):
    self.nom_genome = nom_genome
    self.activation = activation

  def __str__(self):
    return self.nom_genome
```

Données :
La fonction influx.app. qui à un réseau $\mathscr{R} = (C_i)_{1\leqslant i\leqslant q}$ et un vecteur $\vec{x}$ associe la liste $\left[\widetilde{C_1}(\vec{x})\ ;\ \widetilde{C_2}\circ\widetilde{C_1}(\vec{x})\ ;\ \ldots\ ;\ \widetilde{C_q}\circ\cdots\circ\widetilde{C_1}(\vec{x})\right]$.
Entrées :

- $\mathscr{R} = \left((N_{i,j})_{1\leqslant j\leqslant p_i}\right)_{1\leqslant i\leqslant q}$ un réseau neurone tel que pour $(i,j)\in[\![1\ ;q]\!]\times[\![1\ ;p_j]\!]$, $\overrightarrow{w_{i,j}}$ est le vecteur des poids synaptiques de $N_{i,j}$ et $b_{i,j}$ le biais ;
- $M : B \longrightarrow \mathrm{Im}(M)$ un modèle d'apprentissage où B est un ensemble fini non-vide de tuples.
- $\alpha > 0$ le coefficient pondérant l'apprentissage.

début
  $\Delta \leftarrow \left((0)_{1\leqslant j\leqslant p_i}\right)_{1\leqslant i\leqslant q}$
  pour $\vec{x}\in B$ **faire**
    matrice $\leftarrow \mathscr{R}$.influx_app($\vec{x}$)
    pour $i\in[\![1\ ;\ p_q]\!]$ **faire**
      $[\Delta]_{q,i} \leftarrow [\text{matrice}]_{q,i}\times(1-[\text{matrice}]_{q,i})\times([M(\vec{x})]_i - [\text{matrice}]_{q,i})$
    fin
    pour $k = q-1$ **à** 1 **faire**
      pour $i\in[\![1\ ;\ p_k]\!]$ **faire**
        $s\leftarrow 0$
        pour $j\in[\![1\ \ p_{k+1}]\!]$ **faire**
          $s \leftarrow s + [\Delta]_{k+1,j}\times\left[\overrightarrow{w_{k+1,j}}\right]_i$
        fin
        $[\Delta]_{k,i} \leftarrow [\text{matrice}]_{k,i}\times(1-[\text{matrice}]_{k,i})\times s$
      fin
    fin
    pour $i\in[\![1\ ;\ q]\!]$ **faire**
      pour $j\in[\![1\ ;\ p_i]\!]$ **faire**
        $b_{i,j}\leftarrow b_{i,j}+\alpha\times[\Delta]_{i,j}$
        pour $k\in\left[\!\left[1\ ;\ \text{ordre}\left(N_{i,j}\right)\right]\!\right]$ **faire**
          si $i=0$ **alors**
            $e\leftarrow[\vec{x}]_k$
          fin
          sinon
            $e\leftarrow[\text{matrice}]_{i-1,j}$
          fin
          $\left[\overrightarrow{w_{i,j}}\right]_k = \left[\overrightarrow{w_{i,j}}\right]_k + \alpha\times\Delta_{i,j}\times e$
        fin
      fin
    fin
  fin
fin

2- Classe `Neurone`

Programme

```
class Neurone(object):
  """ Classe d'un Neurone. """
  def __init__(self, nom_neurone ="default", ordre =1, Adn =None):
    self.nom_neurone = nom_neurone
    self.Adn = Adn
    self.ordre = ordre
    self.activation = Adn.activation
    self.vect_w = np.array([np.random.random_sample()-np.random.random_sample()
                            for i in range(self.ordre)])
    self.biais = np.random.random_sample()-np.random.random_sample()

  def __str__(self) :
    return self.nom_neurone

  def prod_scal(self, x, y):
    return np.dot(x,y)

  def influx(self, vect_x):
    ''' Calcule l'image d'un vecteur par un neurone. '''
    self.sortie_tmp = self.activation( self.prod_scal(vect_x, self.vect_w)
                                       + self.biais)
    return self.sortie_tmp
```

3- Classe `Couche`

Programme

```
class Couche(object):
  """ Classe Couche."""
  def __init__(self, nom_couche ="defaut", ordre =1, Adn =None, couche =[]):
    self.nom_couche = nom_couche
    self.Adn = Adn
    self.ordre = ordre
    self.activation = Adn.activation
    self.couche = couche[:] # liste vide de neurones
    self.verifie_couche()

  def __str__(self):
    return self.nom_couche

  def verifie_couche(self):
    ''' Verifie que la couche initialisee par l'utilisateur est viable. '''
    for neur in self.couche :
      if (self.Adn != neur.Adn) :
        raise Warning ("Genome non respecte : "+str(self)+" -> "+str(neur))
      if (self.ordre != neur.ordre) :
        raise Warning ("Ordre non respecte : "+str(self)+" -> "+str(neur))

  def ajout_neurone(self, neurone =None):
    ''' Verifie que le neurone ajoute par l'utilisateur est viable. '''
    if neurone == None:
      self.couche.append(Neurone("N"+str(len(self.couche)), self.ordre, Adn =self.Adn))
    else:
```

```
      if neurone.Adn == self.Adn and neurone.ordre == self.ordre:
        self.couche.append(neurone)
      else:
        raise Warning ("Genome non respecte!")

  def influx_couche(self, vect_x):
    ''' Pour une couche [Ni] de neurones et un vecteur x, on renvoit la liste
        [Ni(x)]. '''
    sortie = []
    for neurone in self.couche:
      sortie.append(neurone.influx(vect_x))
    return np.array(sortie)
```

4- Classe `Cerveau`

Programme

```
class Cerveau(object):
  def __init__(self, nom_cerveau ="default", Adn =None, reseau =[]):
    self.nom_cerveau = nom_cerveau
    self.Adn = Adn
    self.reseau = reseau[:]

  def __str__(self):
    return self.nom_cerveau

  def influx_cerveau(self, vect_x):
    sortie = vect_x
    for couche in self.reseau:
      sortie = couche.influx_couche(sortie)
    return sortie

  def ajout_couche(self, couche =None):
    taille_cerveau = len(self.reseau)
    if (couche == None):
    #Ajout d'une couche avec parametres initialisees par defaut
      if (taille_cerveau == 0):
      #Ajout d'une couche dans le cas ou le cerveau est vide
        self.reseau.append(Couche(nom_couche ="c0", Adn = self.Adn))
      else: #Cas ou le cerveau est non vide, Ajout en queue
        self.reseau.append(Couche(nom_couche ="c"
                                  +str(len(self.reseau)),
                                  ordre =len(self.reseau[-1].couche),
                                  Adn =self.Adn))
    else: #Ajout d'une couche specifique
      if (couche.Adn == self.Adn):
      #On verifie la compatibilite du genome
        if (taille_cerveau == 0):
        #Si le cerveau est vide on ne verifie pas l'ordre
          self.reseau.append(couche)
        elif (len(self.reseau[-1].couche) == couche.ordre):
        #Cerveau non vide, on verifie l'ordre
          self.reseau.append(couche)
        else:
          raise Warning ("Ordre non respecte!")
      else:
        raise Warning ("Genome non respecte!")
```

```
def ajout_neurone(self, couche, neurone =None):
  cmp = 0
  while (cmp < len(self.reseau)-1) and (couche != self.reseau[cmp]):
    cmp += 1
  if cmp < len(self.reseau):
    couche.ajout_neurone(neurone)
  if cmp < len(self.reseau)-1:
    self.reseau[cmp+1].ordre = len(couche.couche)
    for neurone in self.reseau[cmp+1].couche:
      neurone.ordre = len(couche.couche)
```

IV Expérience sur le perceptron

A Position du problème

1- Nécessité d'une première approche simple

Lorsque nous avons découvert le concept de neurone formel et leurs applications en informatiques, nous sommes restés perplexes devant ce qui semblait relever de la science-fiction : comment un programme informatique pourrait-il apprendre quelque chose et se réajuster en fonction de cet enseignement ? En effet, cela semble *a priori* contredire tout ce que nous croyons savoir de l'automatisme des algorithmes, qui pour nous étaient des entités fixés à leur création et paramétrées uniquement par leurs concepteurs, et en aucun cas destinées à évoluer d'eux-même en fonction d'un apprentissage.

Nous avons donc décidé, plutôt que d'essayer d'attaquer directement la résolution de notre problème, de nous ramener à une situation plus simple pour expérimenter l'implémentation du neurone formel dans un cas concret. Nous pourrons ainsi constater son fonctionnement, ou non-fonctionnement le cas échéant ; nous pourrons aussi voir de quelle façon fonctionne l'apprentissage d'une neurone formel, possibilité qui contredit encore pour l'instant notre intuition ; nous pourrons enfin mettre en évidence certaines conditions nécessaire au bon fonctionnement du neurone.

2- Une sortie au cinéma

Imaginons qu'un samedi après-midi, un étudiant envisage une sortie au cinéma. Nous supposons que cette sortie sera déterminée par les réponses quatre questions :

i. Est-ce que l'étudiant est en bonne santé ?

ii. Est-ce que la météo est bonne ?

iii. Est-ce que la voiture est disponible ?

iv. Est-ce que le film est bon ?

L'étudiant indécis étant incapable de prendre une décision à partir de ces critères, nous décidons d'élaborer un programme répondant à sa place à la question suivante : peut-il aller au cinéma ?

B Élaboration du programme

1- Principe

Nous considérons que pour chaque réponse, "oui" sera traduit en machine par le nombre 1 et "non" par le nombre 0. De même, le programme renverra 1 si l'étudiant peut aller au cinéma et 0 sinon. Nous allons évidemment travailler avec un neurone, pour l'instant unique. L'espace de départ sera $\{0;1\}^4$ et celui d'arrivée sera $\{0;1\}$. Étant donné la dualité de la sortie, la fonction indicatrice de l'intervalle $\mathbb{R}_+$ servira de fonction de seuil. En clair, nous nous servirons d'un perceptron.

Soit $N = \text{neur}\left(\{0;1\}, b, \vec{w}, \mathbb{1}_{\mathbb{R}_+}\right) = \text{perc}(\{0;1\}, -b, \vec{w})$ où $b \in \mathbb{R}$ et $\vec{w} \in \mathbb{R}^4$ sont initialisés au hasard. Ce neurone nous servira de base pour nos premières expérimentations. On se donne ensuite un tableau dont chaque élément est un

couple est tel que la première composante soit le tableau des réponses aux quatre questions et la deuxième composante est la réponse dont on veut qu'elle soit renvoyée par le neurone. Ce tableau fera office de modèle et servira de modèle au neurone pour l'apprentissage.

Programme

```
modele = np.array([ ([0,0,0,0], 0), ([1,0,0,0], 0), ([0,1,0,0], 0),
                    ([0,0,1,0], 0), ([0,0,0,1], 0), ([1,1,0,0], 0),
                    ([1,0,1,0], 0), ([1,0,0,1], 1), ([0,1,1,0], 0),
                    ([0,1,0,1], 1), ([0,0,1,1], 0), ([1,1,1,0], 1),
                    ([1,1,0,1], 1), ([1,0,1,1], 1), ([0,1,1,1], 1),
                    ([1,1,1,1], 1) ])
```

L'objectif du programme va être de paramétrer le réseau de neurones pour qu'il associe à chaque combinaison de critères possible la réponse imposée par le modèle. Pour cela, on utilise notre implémentation de l'algorithme de WIDROW-HOFF.

2- Tracé de courbes

Pour comprendre le fonctionnement de l'apprentissage de notre neurone, nous nous appuierons sur des résultats graphiques. Chaque graphe représentera en abscisse le nombre d'itérations de la fonction d'apprentissage et en ordonnée le pourcentage de réponses justes renvoyées par le neurone. Ainsi si on nous donne un point (x, y) de la courbe, on saura qu'après x itérations de la courbe d'apprentissage, le neurone renverra une réponse juste dans y % des cas.

Pour le tracé des courbes, on utilisera la bibliothèque `Matplotlib`. On utilise une fonction auxiliaire `bonnes_reponses` qui pour un neurone et un modèle donné, renverra le pourcentage de bonnes réponses (imposées par le modèle) données par le neurone. La forme générale des algorithmes de tracé de courbes sera :

Programme

```
def bonnes_reponses(modele, neurone):
  b_rep = 0
  for entree,sortie in modele:
    b_rep = b_rep + 1 if (neurone.influx(entree) == sortie)
                      else b_rep
  return float(b_rep)/float(len(modele))*float(100)

def courbes(n,iteration): #n est le nombre de courbes
  lx = [i for i in range(iteration)]
  for j in range(n):
    N1 = Neurone(4) #Initialisation du neurone au hasard
    ly = []
    for i in lx:
      apprentissage(N1,0.5)
      ly.append(bonnes_reponses(modele,N1))
    py.plot(lx,ly)
```

C Étude des courbes

1- Principe

Dans un premier temps, nous nous contentons d'appliquer simplement l'algorithme d'apprentissage sur diverses neurones initialisés au hasard, chaque neurone devant correspondre à une courbe. On applique donc l'algorithme précédent. En légende sont indiquée les numéros des courbes, les vecteurs synaptiques et les biais initiaux, les vecteurs synaptiques et les biais finaux : `[numero, w_i, b_i, w_f, b_f]`.

Nous avons en effet tracé plusieurs dizaines de courbes, dont nous avons conservé seulement un vingtaine que nous avons représentées dans la figure 6, page 16.

2- Premières courbes

Nous constatons tout d'abord que l'apprentissage converge quelque soit les conditions initiales. Nous avons en effet tracé plusieurs dizaines de courbes, dont nous avons conservé seulement un vingtaine et à chaque fois, au bout d'un certain nombres d'itérations, le taux de bonnes réponses finit par atteindre 100%. Le neurone peut atteindre un taux de réussite de 100 % entre cinq et trente itérations. De manière générale on constate d'ailleurs que plus le taux de bonnes réponses initial du neurone est proche de 100 %, plus il convergera vite et à l'inverse, si le hasard a été moins favorable il aura tendance à converger moins vite.

Lorsque l'on observe les conditions finales, i.e. les vecteurs synaptiques et les biais obtenus à la fin de l'apprentissage, on voit que toutes les valeurs sont relativement proches, sachant que l'on s'était limité à des initialisations dans $[\![-9\,;\,9]\!]$. Ainsi le biais obtenu est toujours négatif, compris dans l'intervalle $[\![-10\,;\,-5]\!]$. De même les vecteurs synaptiques sont à composantes positives, les valeurs étant en majorité comprises dans l'intervalle $[\![1\,;\,5]\!]$. On constate par ailleurs que des perceptrons construits avec des paramètres différents, mais proches, peuvent représenter la même fonction : cela illustre le fait présenté à la sous-sous-section I-B)2.

Un dernier phénomène remarquable est est celui du dés-apprentissage : en effet les taux de bonnes de réponses ne sont pas des fonctions croissantes de l'itération et on peut constater que presque systématiquement, on a des baisses de ce taux avant qu'il augmente à nouveau, ces bais-ses pouvant atteindre 20 % en cinq itérations. Ce phénomène pourra poser problème dans la suite car il s'oppose à l'efficacité de l'apprentissage des réseaux de neurones. Néanmoins il n'est pas possible d'assister à un dés-apprentissage quand le neurone est paramétré de manière optimale, ce qui avait été prédit lors de l'étude théorique de l'algorithme de WIDROW-HOFF.

3- Restriction du modèle

Cette fois-ci on choisit d'étudier l'effet sur l'apprentissage quand on restreint le modèle. Au lieu de fournir au neurone toutes les réponses possibles pour s'entraîner, on sélectionne cinq éléments au hasard. Une courbe de la figure 5 représente donc un apprentissage pour une restriction sur 100 itérations.

On remarque alors que le neurone peut approximer un apprentissage parfait, ce qui confirme la possibilité d'utiliser cette théorie dans le cas réel de reconnaissances de caractères, où le modèle d'entrainement ne sera jamais complet.

On remarque également que les restrictions n'aboutissent pas toutes au même pourcentage d'approximation ce qui met en évidence la pertinence des éléments sélectionnés pour le modèle.

4- Séparabilité linéaire

Dans cette figure on cherche à mettre en évidence le cas de non séparabilité linéaire du modèle. Pour ceci, on modifie le modèle de façon à avoir $((0,0,0,0),0)$ et $((1,1,1,1),0)$. On fournit donc un modèle non linéairement séparable. D'après le graphique, l'apprentissage ne converge jamais vers 100 % ce qui confirme la propriété sur les perceptrons.

5- Influence de α

Dans cette figure on cherche à évaluer l'influence du coefficient d'apprentissage α. Pour cela on initialise un même neurone (i.e. biais et poids égaux pour chacun) auquel on fait apprendre le modèle pour plusieurs α (ceux de la légende du graphique). On remarque alors que pour le plus grand coefficient (courbe violette, coefficient : 2.01) on a une convergence rapide, inférieur à 10 itérations d'après le graphique. A l'inverse pour le plus petit coefficient(courbe bleu marine, coefficient : 0.01) on a une convergence beaucoup plus lente, de l'ordre de 320 itérations. Néanmoins l'avantage d'un petit coefficient est pour le dés-apprentissage, c'est à dire qu'après un apprentissage, les paramètres changent d'une façon plus fine, ce qui permet de mieux conserver l'apprentissage précédent.

FIGURE 6 – Convergence de l'apprentissage pour des conditions initiales prises au hasard