Un réseau de neurones simple

- La partie 1 présente l'exemple choisi.
- La partie 2 propose de programmer un réseau de neurones à une couche de manière complètement basique.
- Dans 3, on aborde la même chose d'un point de vue algèbre linéaire, qui fait apparaître clairement les limitations d'un réseau à une seule couche.
- Pour ceux qui veulent aller plus loin, la partie 4 explique le principe de base des réseaux de neurones multicouches. Je donne les calculs jusqu'au point où je les ai moi-même poussés (c'est-à-dire pas si loin que ça...).

Le fichier `bibNeurones.py` contient un minimum de données pour faire des tests. Le fichier `neuronesÀRemplir.py` contient en outre une proposition de fonction intermédiaires avec leurs spécifications.

1 Présentation du contexte

On propose ici un réseau de neurones à une seule couche, appelé « perceptron ». Ce type de réseau de neurones a été introduit par Frank Rosenblatt en 1957.

Le but sera de reconnaître des images : le programme final prendra en entrée une image et en sortie indiquera ce qu'elle représente. Nos images seront en noir et blanc (ou éventuellement niveau de gris à partir de 2.2). Donc chaque pixel contient un nombre. Par exemple :

```
un = [ [0, 1, 0],
       [1, 1, 0],
       [0, 1, 0],
       [0, 1, 0],
       [0, 1, 0]]
```

La matrice `un` représente une image 5×3 pixels qui représente le chiffre 1.

Soit $\mathcal{C}$ l'ensemble des « catégories » à reconnaître. Chaque image est supposée appartenir à une des catégories. Par exemple si nous voulons reconnaître des chiffres, $\mathcal{C} = [\![0, 9]\!]$.

Un perceptron a un neurone d'entrée pour chaque pixel de l'image lue, et un neurone de sortie pour chaque catégorie. Les différents neurones sont reliés entre eux par des «synapses».

Mettons qu'un neurone d'entrée est activé lorsque le pixel correspondant vaut 1. Nous allons choisir une formule pour déterminer quels neurones de sortie s'activent.

Notations :

- n le nombre de lignes et p le nombre de colonnes des images.
- $N_e = n \times p$ le nombre de pixels des images. Ce sera aussi le nombre de neurones d'entrées de nos réseaux. Pour toute image `Im` et tout $(i, j) \in [\![0, n[\![\times [\![0, p[\![$, nous dirons que le neurone d'entrée (i, j) est activé en lisant `Im` lorsque `Im[i][j]==1`.
- $N_s = \text{Card}(\mathcal{C})$: c'est le nombre de catégories à reconnaître. Ce sera aussi le nombre de neurones de sorties de nos réseaux.
- L'ensemble des catégories sera enregistré dans un tableau `catégories` de longueur N_s. Par exemple `catégories = ["0", "1", "2","+", "-", "a",...]`.

2 Réseau une couche, version élémentaire

Dans ce premier réseau, nous relions chaque neurone d'entrée à chaque neurone de sortie par une synapse, ce qui nous fait $N_e \times N_s$ synapses. Pour tout $(i, j, k) \in [\![0, n[\![\times [\![0, p[\![\times [\![0, N_s[\![$, nous noterons $P^k_{i,j}$ un nombre qui représente la «connectivité» de la synapse qui relie l'entrée liée au pixel (i, j) à la sortie liée à la catégorie k. Dans les programmes, tous ces coefficients seront rangés dans une matrice `P` de format (N_s, n, p) de telle sorte que $P^k_{i,j}$ s'obtiendra par `P[k][i][j]`.

Pour toute image `Im`, et pour tout $k \in [\![0, N_s[\![$, nous posons :

$$\mathcal{A}(k, \texttt{Im}, \texttt{P}) = \sum_{i=0}^{n-1} \sum_{j=0}^{p-1} \texttt{P[k][i][j]} \times \texttt{Im[i][j]}.$$

Interprétation : Pour tout (i,j), `Im[i][j]` est le signal émis par le neurone d'entrée (i,j), puis `P[k][i][j]` $\times$ `Im[i][j]` est ce qui est transmis au neurone de sortie k.

Ainsi, $\mathcal{A}(k, \texttt{Im}, \texttt{P})$ est la quantité totale de signal reçue par le neurone de sortie k. Nous dirons que le neurone de sortie k est « activé » lors de la lecture de `Im` lorsque $\mathcal{A}(k, \texttt{Im}, \texttt{P}) \geqslant 1$.

Programmation : Programmons la fonction $\mathcal{A}$, ainsi qu'une fonction renvoyant le tableau des synapses activées.

Initialement, nous pouvons remplir `P` de 0, ou de nombres aléatoires. Vient ensuite la phase d'apprentissage. Nous présentons au réseau de nombreuses images dont nous connaissons la catégorie, et à chaque erreur faite nous corrigeons `P`.

2.1 Correction des poids : méthode 1

À chaque image `Im` lue, qui appartient à la catégorie numéro k_0,

- Si le neurone de sortie k_0 n'a pas été activé, nous augmentons `P[k0][i][j]` pour tout $(i,j) \in [\![0,n[\![\times[\![0,p[\![$ tel que `Im[i][j]==1` ;
- Pour tout $k \in [\![0,10[\![$, si $k \neq k_0$ mais que le neurone k s'est activé, nous diminuons `P[k][i][j]` pour tout $(i,j) \in [\![0,n[\![\times[\![0,p[\![$ tel que `Im[i][j]==1`.

On fixera une constante η, qui indiquera de combien on augmente ou diminue les coefficients à chaque étape. La constante η est appelée le « taux d'apprentissage » du réseau.

Remarque : Si η est trop petit, la convergence sera trop lente, s'il est trop grand on risque de « sauter » la valeur optimale des coefficients.

On peut alors proposer les étapes intermédiaires suivantes pour programmer notre réseau :

1. Implémenter la lecture d'une image, avec correction de `P` pour chaque erreur.
2. Restent alors les détails : parcourir une banque d'images suffisamment de fois pour que plus aucune erreur ne se produise, enregistrer la matrice `P` finale, et l'utiliser pour notre fonction finale de détection d'image.

2.2 Correction des poids, méthode 2

Lorsque nous corrigeons un coefficient, nous le corrigeons d'une valeur proportionnelle à l'écart entre la valeur de $\mathcal{A}(k, Im, P)$ souhaitée, et celle obtenue.

En prenant comme valeur souhaitée 2 si k est la catégorie de `Im` et 0 sinon, ceci donne la règle suivante : pour tout $(i,j,k) \in [\![0,n[\![\times[\![0,p[\![\times[\![0,N_s[\![$, lors de la lecture d'une image `Im` appartenant à la catégorie d'indice k_0 :

- Si `Im[i][j]==1` et $k = k_0$, nous augmentons $P[k][i][j]$ de la quantité $\eta \times (2 - \mathcal{A}(k, \texttt{Im}, \texttt{P})$.
- Si `Im[i][j]==1` et $k \neq k_0$, nous augmentons $P[k][i][j]$ de la quantité $-\eta \times \mathcal{A}(k, \texttt{Im}, \texttt{P})$.
- Si `Im[i][j]==0`, le coefficient n'est pas modifié.

On peut rassembler ces trois cas en un seul : dans tous les cas, le coefficient est modifié de

$$\eta \times \big(Im[i][j]\big) \times (\text{Valeur de } \mathcal{A} \text{ souhaitée} - \text{Valeur de } \mathcal{A} \text{ obtenue}) \,.$$

Remarque : Cette formule peut être utilisée pour une image en namces de gris.

3 Une couche, version matricielle. Limitations.

Changement de notations : je numérote désormais les pixels de 0 à $N_e - 1$, via la fonction $(i,j) \mapsto pi + j$, dont la réciproque est $n \mapsto divmod(n,p)$ (`divmod` est la fonction Python qui renvoie le quotient et le reste de la division euclidienne). Ainsi, je considère qu'une image est un tableau simple de N_e cases.

Alors `P` devient une matrice de format $N_s \times N_e$, la formule pour $\mathcal{A}$ devient

$$\forall k \in [\![0,N_s[\![\quad \mathcal{A}(k, Im, P) = \sum_{i=0}^{N_e-1} P[k][i] \cdot Im[i]$$

Si je note $\widehat{\mathcal{A}}(Im, P)$ la colonne contenant $\big(\mathcal{A}(0, Im, P), ..., \mathcal{A}(N_e - 1, Im, P)\big)$, et si j'identifie Im à une matrice colonne, j'obtiens :

$$\widehat{\mathcal{A}}(Im, P) = P \times Im.$$

Remarque : Si nous voulons programmer cette version, le produit matriciel sur une version pas trop vieille de numpy s'obtient par `@`.

Par ailleurs, on remarque que les $\mathcal{A}(k, \cdot, P)$ sont des formes linéaires (de matrice `P[k]`). Donc repérer les images qui sont dans la catégorie k revient à trouver un hyperplan qui sépare ces images des autres. Ce n'est possible que si les enveloppes convexes des deux ensembles sont disjointes (théorème de Hahn Banach (à confirmer)).

Ainsi les situations dans lesquelles un réseau de neurones à une couche sont-elles finalement assez limitées. C'est la raison pour laquelle le domaine a été abandonné dans les année 70.

4 Plusieurs couches

Wikipédia date de 1985 l'introduction des réseaux multicouches, qui relancent l'utilisation des réseaux de neurones.
Remarque : Le terme «deep learning»est employé lorsqu'on utilise un réseau «profond» c'est-à-dire avec beaucoup de couches.

4.1 Principe

Soit $C \in \mathbb{N}^*$ (ce sera le nombre de couches). Pour tout $i \in [\![1, C-1]\!]$, soit $N_i \in \mathbb{N}^*$, ce sera le nombre de neurones de la couche i. On note également $N_0 = N_e$, et $N_C = N_s$.
Remarque : Cela fait $C + 1$ couches si on compte les neurones d'entrée.

Pour tout $i \in [\![1, C]\!]$, chaque neurone de la couche i est relié par une synapse à chaque neurone de la couche $i - 1$, et nous notons $P_i \in \mathcal{M}_{N_i, N_{i-1}}(\mathbb{R})$ la matrice des coefficients synaptiques.

Je note enfin $A_i \in \mathcal{M}_{N_i, 1}(\mathbb{R})$ la matrice colonne qui donne l'activation de chaque neurone de la couche i. Ainsi, A_0 correspond à l'image lue, et A_C et la sortie de notre réseau.

Si j'utilise la même formule que précédemment, j'obtiens $\forall i \in [\![0, C[\![$, $A_{i+1} = P_{i+1} \times A_i$, d'où :

$$A_C = A_C \times ... \times A_1 \times Im$$

On remarque immédiatement que cette méthode n'a aucun intérêt (pourquoi ?) ! Il faut modifier la formule pour obtenir quelque chose de non linéaire.

Inspiré par le fonctionnement d'un vrai neurone, on introduit le concept de la « fonction d'activation ». Un choix fréquent est d'utiliser la fonction tangente hyperbolique : si x est la quantité de signal reçu par un neurone, je décide que le degré d'activation du neurone, c'est-à-dire en pratique la quantité de signal qu'il va transmettre à la couche suivante, est $\text{th}(x)$. Donc en gros, si le neurone reçoit beaucoup de signal, il est activé et il renvoie 1 à la couche suivante. Si au contraire il reçoit peu de signal (c'est-à-dire beaucoup de signal négatif), il n'est pas activé et renvoie -1 à la couche suivante.
Remarques :

- Si on préfère qu'un neurone non activé renvoie 0, prendre comme fonction d'activation $\frac{1}{2}(\text{th} + 1)$.
- Dans la partie précédente on décidait que le neurone s'activait lorsque le signal reçu était $\geqslant 1$. Cela revient à utiliser une fonction d'activation « d'Heavyside » : la fonction qui vaut 0 sur $]-\infty, 1[$ et 1 sur $[1, \infty[$.

Notons f_i la fonction d'activation choisie pour la couche i et $\tilde{f}_i$ la fonction qui prend un tableau et qui applique f_i à chaque case du tableau (dans numpy, $\tilde{f}$ est encore notée f_i). La formule précédente devient $\forall i \in [\![0, C[\![$, $A_{i+1} = \tilde{0}_{i+1} f(P_{i+1} \times A_i)$. Ouf, ce n'est plus linéaire !

Pour permettre encore plus de réglages possibles, on remplace souvent la partie linéaire de la formule par une partie affine : on choisit encore $(B_1, ..., B_C) \in (\mathcal{M}_{N_{i+1}, 1}(\mathbb{R}))^C$, et on utilise la formule $A_{i+1} = \tilde{f}_{i+1}(P_{i+1} \times A_i + B_{i+1})$.

4.2 Lecture d'une image

Cette étape est facile, surtout avec numpy.
Ceux qui veulent de la programmation objet peuvent utiliser :

```
class couche():

  def  __init__(self, Ne, Ns, P=None, activation=np.tanh):
    """ Ne : np de neurones d'entrée
        Ns : nb de neurones de sortie
        P : matrice des poids. Si pas donnée elle est générée aléatoirement.
        activation : fonction d'activation"""
    self.Ne=Ne
    self.Ns=Ns
    self.activation = activation
    if P==None:
      self.P = np.random.rand ( (Ns, Ne) )
    else: self.P=P
```

4.3 Réglage des poids

La méthode de base s'appelle la «descente de gradient».
Pour tout $x \in \mathbb{R}^{N_e}$ une entrée, notons $\mathcal{S}(P_1, B_1, ..., P_C, B_C, x)$ la sortie obtenue. Donc

$$\mathcal{S}\left(P_1, B_1, ..., P_C, B_C, x\right) = B_C + \tilde{f}_C\left(P_C \times \left(...\tilde{f}_2(B_2 + P_2 \times \tilde{f}_1(B_1 + P_1 \times x))...\right)\right)$$

Notons y_a la valeur de la sortie attendue. Nous appellerons « erreur » le nombre :

$$err\left(P_1, B_1, ..., P_C, B_C, x\right) = \frac{1}{2}\left\|y_a - \mathcal{S}\left(P_1, B_1, ..., P_C, B_C, x\right)\right\|_2^2$$

Remarque : On prend une norme 2, donc dans le fond, on fait une méthode des moindres carrés.

Ainsi à la lecture de x, nous voulons modifier les P_k et les B_k en vue de minimiser l'erreur. La méthode de la descente de gradient consiste à calculer le gradient de l'erreur, et à avancer de η dans cette direction. Attention : les variables sont ici tous les coefficients des P_k et des B_k.

Ainsi pour tout i, j, k, le coefficient $(P_k)_{i,j}$ doit être modifié de $-\eta\dfrac{\partial err}{\partial (P_k)_{i,j}}$.

4.4 Cas d'un réseau à deux couches

Voici le résultat de mes calculs dans le cas d'un réseau à deux couches ($C = 2$), et sans biais ($B_0 = 0$, $B_1 = 0$). Je note f l'unique fonction d'activation.
Soit $x \in \mathbb{R}^{N_e}$ une entrée. La sortie obtenue est :

$$\mathcal{S}(P_1, P_2, x) = P_2 \times \tilde{f}\left(P_1 \times x\right)$$

Je pose $\mathcal{E} : z \mapsto \frac{1}{2}\left\|y_a - z\right\|$ de sorte que l'erreur se réécrit $\mathcal{E}\left(\mathcal{S}(P_1, P_2, x)\right)$.
La différentielle de $\mathcal{E}$ en un point (P_1, P_2) est $\left\langle \mathcal{B}(P_1, P_2) - y_a \mid \cdot \right\rangle$.
La jacobienne de $\tilde{f}$ au point $P_1 x$ est $Diag\left(f'\left((P_1x)_0\right), ..., f'\left((P_1x)_0\right)\right)$.

Soit $(j, k) \in [\![0, N_1[\![\times [\![0, N_2[\![$. Je trouve que

$$\frac{\partial err}{\partial P_2[j,k]} = \left(\mathcal{S}(P_1, P_2)_j - y_a[j]\right) \times f\left((P_1x)_k\right)$$

Puis pour $(i, j) \in [\![0, N_0[\![\times [\![0, N_1[\![$

$$\frac{\partial err}{\partial P_1[i,j]} = f'((P_1x)_i)x_j \sum_l (\mathcal{B}_l - y_a[l])P2[l,i]$$