

Rétropropagation

Vidéo ■ partie 9.1. Principe de la rétropropagation

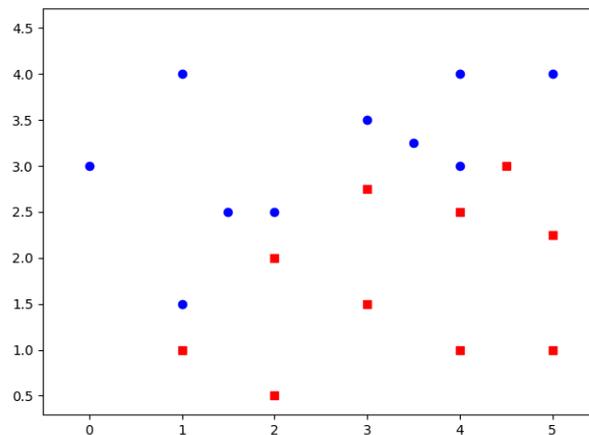
Vidéo ■ partie 9.2. Exemples de rétropropagation

Vidéo ■ partie 9.3. Sur-apprentissage et autres soucis

La rétropropagation, c'est la descente de gradient appliquée aux réseaux de neurones. Nous allons étudier des problèmes variés et analyser les solutions produites par des réseaux de neurones.

1. Principe de la rétropropagation

Voici un jeu de données : des ronds bleus et des carrés rouges. Nous souhaitons trouver un modèle simple qui caractérise ces données.

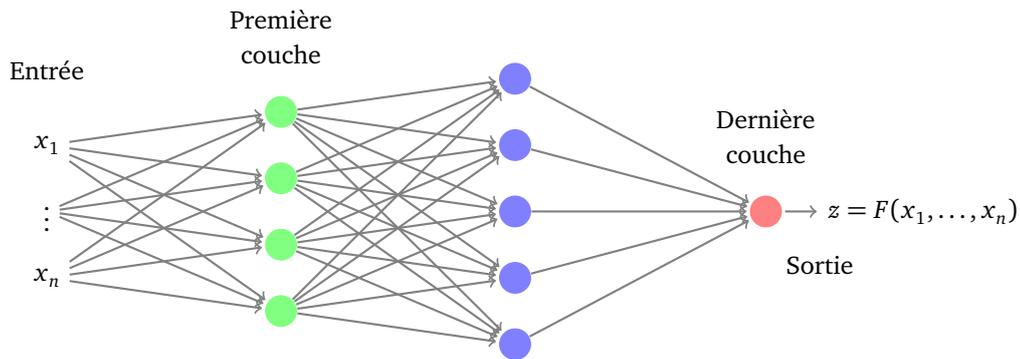


Plus exactement, nous souhaiterions pouvoir dire pour chaque point du plan s'il devrait être colorié en rouge ou bien en bleu et ceci pour des points qui ne sont pas dans les données de départ. De plus, nous voulons ne rien faire à la main, mais que la réponse soit calculée par la machine !

Nous allons revoir pas à pas l'utilisation d'un réseau de neurones pour résoudre un problème et traiterons en particulier l'exemple ci-dessus.

1.1. Objectif du réseau

- Soit \mathcal{R} un réseau de neurones. Celui-ci est défini par son architecture (le nombre de couches, le nombre de neurones par couche), les fonctions d'activation et l'ensemble $P = (a_1, a_2, \dots)$ des poids de tous les neurones.
- À ce réseau \mathcal{R} on associe une fonction $F : \mathbb{R}^n \rightarrow \mathbb{R}^p$ où n est la dimension des entrées (de la première couche) et p le nombre de neurones de la couche de sortie. Dans ce chapitre, nous supposons qu'il n'y a qu'une seule sortie, c'est-à-dire $p = 1$ et $F : \mathbb{R}^n \rightarrow \mathbb{R}$.



- On dispose de données (X_i, z_i) (pour $i = 1, \dots, N$) où $X_i \in \mathbb{R}^n$ est une **entrée** (de la forme $X = (x_1, \dots, x_n)$) et $z_i \in \mathbb{R}$ est la **sortie attendue** pour cette entrée.
- Le but est de trouver les poids du réseau afin que la fonction F qui lui est associée vérifie :

$$F(X_i) \simeq z_i \quad \text{pour tout } i = 1, \dots, N.$$

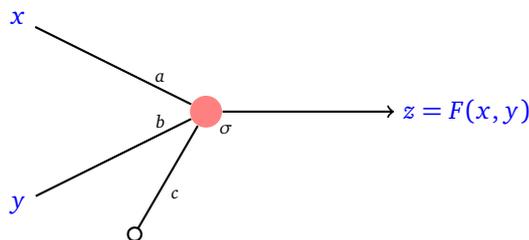
- Pour mesurer précisément la performance de l'approximation, on définit une **fonction erreur** :

$$E = \frac{1}{N} \sum_{i=1}^N E_i \quad \text{avec} \quad E_i = (F(X_i) - z_i)^2.$$

Exemple.

Nous traitons l'exemple donné en introduction.

- On décide de construire le réseau le plus simple possible : avec un seul neurone. Cela correspond à séparer les points du plan selon une droite. On choisit la fonction d'activation σ . La dimension de l'entrée est 2 et celle la sortie est 1. Il y a 3 poids (a, b, c) à calculer pour terminer le paramétrage du réseau.



- Pour chaque triplet de poids (a, b, c) notre réseau définit une fonction $F : \mathbb{R}^2 \rightarrow \mathbb{R}$ qui est en fait ici :

$$F(x, y) = \sigma(ax + by + c)$$

avec $\sigma(t) = \frac{1}{1+e^{-t}}$.

- Nos données sont les points rouges et bleus. Une entrée X_i est donc constituée des coordonnées (x_i, y_i) d'un point et la sortie attendue pour ce point est $z_i = 0$ (pour les points bleus) et $z_i = 1$ (pour les points rouges). Voici les coordonnées des ronds bleus (sortie attendue $z_i = 0$) :

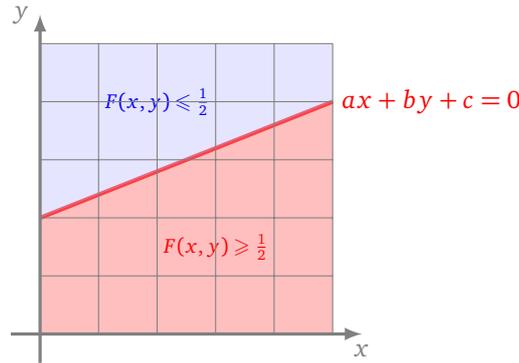
$(0, 3), (1, 1.5), (1, 4), (1.5, 2.5), (2, 2.5), (3, 3.5), (3.5, 3.25), (4, 3), (4, 4), (5, 4)$

et des carrés rouges (sortie attendue $z_i = 1$) :

$(1, 1), (2, 0.5), (2, 2), (3, 1.5), (3, 2.75), (4, 1), (4, 2.5), (4.5, 3), (5, 1), (5, 2.25).$

- Comment définir les poids (a, b, c) afin que $F(x_i, y_i) \simeq 0$ pour tous les ronds bleus et que $F(x_i, y_i) \simeq 1$ pour tous les carrés rouges? Si on sait trouver de tels poids alors on pourra colorier (presque) n'importe quel point (x, y) du plan (et pas seulement nos ronds et nos carrés) : si $F(x, y) \simeq 0$, on coloriera le point en bleu, si par contre $F(x, y) \simeq 1$, on le coloriera en rouge. Plus précisément, la fonction F (qui dépend de σ) a ses valeurs dans $[0, 1]$ et prend la valeur $F(x, y) = \frac{1}{2}$ exactement sur la droite $ax + by + c = 0$. Ainsi, la fonction F sépare le plan en deux demi-plans $\{(x, y) \mid F(x, y) \leq \frac{1}{2}\}$ et

$\{(x, y) \mid F(x, y) \geq \frac{1}{2}\}$ le long de la droite $ax + by + c = 0$.



- L'erreur commise par la fonction F associée aux poids a, b, c est :

$$E = E(a, b, c) = \frac{1}{N} \sum_{i=1}^N E_i(a, b, c)$$

où N est le nombre total des données (le nombre de ronds bleus plus le nombre de carrés rouges) et

$$E_i(a, b, c) = (F(x_i, y_i) - z_i)^2.$$

On peut détailler un peu plus pour chaque type de point. En effet, pour un rond bleu la sortie attendue est $z_i = 0$ donc $E_i(a, b, c) = (\sigma(ax_i + by_i + c) - 0)^2$, alors que pour un carré rouge la sortie attendue est $z_i = 1$ donc $E_i(a, b, c) = (\sigma(ax_i + by_i + c) - 1)^2$.

1.2. Descente de gradient

- Pour trouver les poids $P = (a_1, a_2, \dots)$ qui définissent le meilleur réseau \mathcal{R} (autrement dit la meilleure fonction F), il suffit de minimiser l'erreur E , vue comme une fonction des poids $P = (a_1, a_2, \dots)$. Pour cela on utilise la méthode de la descente de gradient.
- On part de poids initiaux $P_0 = (a_1, a_2, \dots)$, par exemple choisis au hasard. On fixe un pas δ .
- On construit par itérations des poids P_k selon la formule de récurrence :

$$P_{k+1} = P_k - \delta \text{grad } E(P_k).$$

À chaque itération, l'erreur $E(P_k)$ diminue. On s'arrête au bout d'un nombre d'itérations fixé à l'avance.

- Pour calculer le gradient $\text{grad } E = \frac{1}{N} \sum_{i=1}^N \text{grad } E_i$, il faut calculer chacun des $\text{grad } E_i$, c'est-à-dire les dérivées partielles par rapport à chacun des poids a_j selon la formule :

$$\frac{\partial E_i}{\partial a_j}(X_i) = 2 \frac{\partial F}{\partial a_j}(X_i) (F(X_i) - z_i).$$

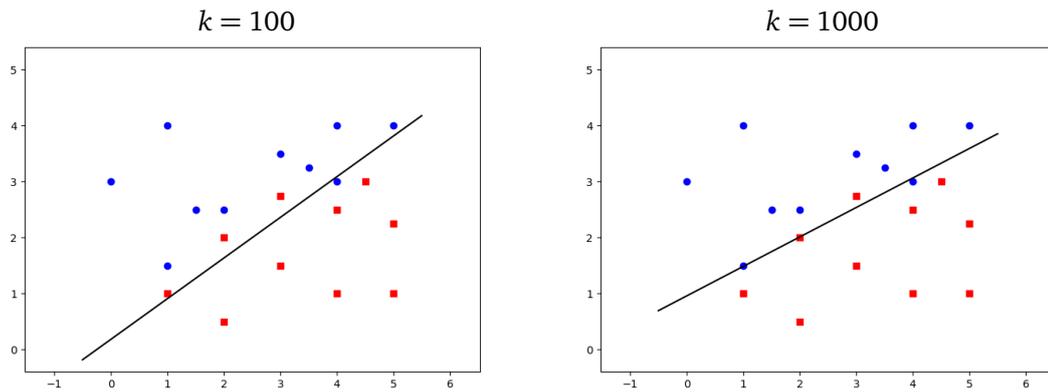
Exemple.

Poursuivons l'étude de notre exemple.

- La fonction $E(a, b, c)$ dépend des poids a, b, c .
- On part de poids initiaux $P_0 = (a_0, b_0, c_0)$, par exemple $P_0 = (0, 1, -2)$ qui correspond à séparer le plan selon la droite horizontale $y = 2$. On fixe le pas $\delta = 1$.
- On calcule l'erreur locale pour la donnée numéro i :

$$E_i(a, b, c) = (F(x_i, y_i) - z_i)^2 = (\sigma(ax_i + by_i + c) - z_i)^2$$

avec $z_i = 0$ ou $z_i = 1$.



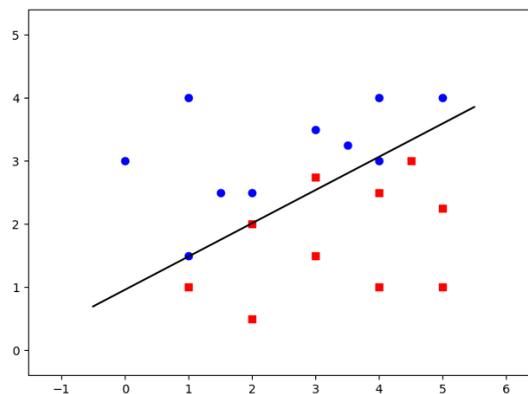
1.3. Prédiction

La conception d'un réseau de neurones est réalisée en modélisant au mieux les données injectées. Mais l'objectif réel est de faire des prédictions pour de nouvelles valeurs, jamais rencontrées auparavant. La descente de gradient produit un ensemble de poids P qui définit complètement notre réseau \mathcal{R} . Nous obtenons donc une fonction $F : \mathbb{R}^n \rightarrow \mathbb{R}^p$, construite de sorte que $F(X_i) \simeq z_i$. Nous pouvons évaluer cette fonction pour tout $X \in \mathbb{R}^n$, même pour des X différents des X_i .

Exemple.

Dans notre exemple, nous avons obtenu par la descente de gradient les poids $P_{1000} = (a, b, c) = (2.032, -3.860, 3.703)$. Notre neurone \mathcal{R} est maintenant opérationnel et définit ici la fonction

$$F(x, y) = \sigma(ax + by + c).$$



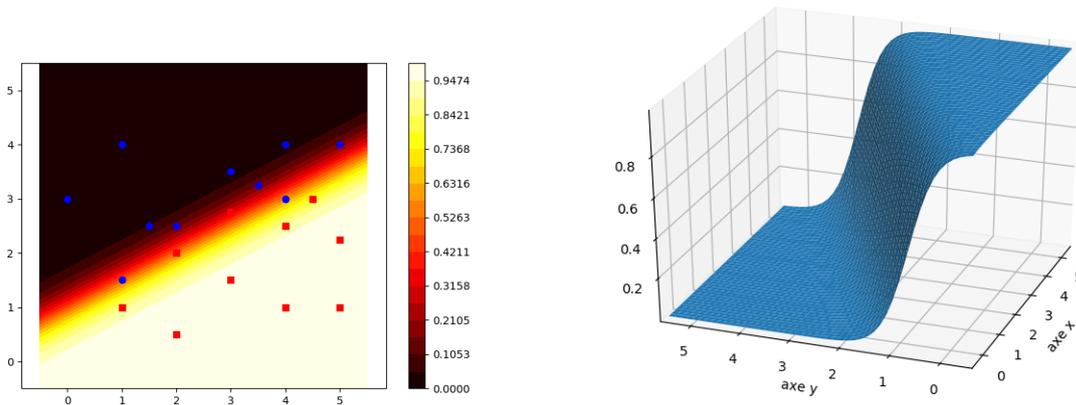
On souhaitait avoir $F(x_i, y_i) = 0$ pour les ronds bleus et $F(x_i, y_i) = 1$ pour les carrés rouges. Dans la pratique, on obtient des valeurs approchées, par exemple $F(0, 3) = 0.0003$ pour le rond bleu en $(0, 3)$ et $F(1, 1) = 0.86$ pour le carré rouge en $(1, 1)$.

Notre fonction F ne modélise pas parfaitement toutes nos données. Par exemple, pour le rond bleu en $(4, 3)$ on a $F(4, 3) = 0.56$ alors qu'on voudrait une valeur proche de 0 et de même pour le carré rouge en $(3, 2.75)$ on a $F(3, 2.75) = 0.30$ alors qu'on voudrait une valeur proche de 1.

Mais l'intérêt principal de F est d'être définie pour tous les points du plan, ceci permet d'attribuer une couleur à chaque point $(x, y) \in \mathbb{R}^2$ selon la convention : bleu si $F(x, y) \simeq 0$, rouge si $F(x, y) \simeq 1$. Il y a bien sûr une « zone grise » entre les zones rouge et bleue.

Par exemple, $F(2, 3) = 0.02$ donc $(2, 3)$ mérite d'être colorié en bleu, $F(2, 1) = 0.98$ donc $(2, 1)$ mérite d'être colorié en rouge. La frontière en laquelle F vaut $\frac{1}{2}$ est la droite d'équation $ax + by + c = 0$.

Voici les niveaux de F dans le plan (à gauche) et le graphe de F dans l'espace (à droite).

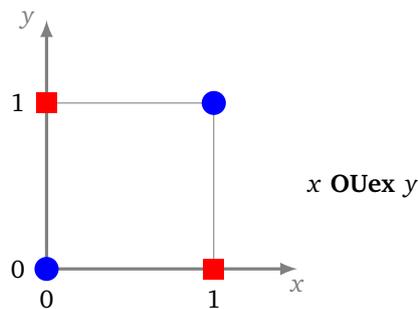


Nous avons choisi un réseau avec un seul neurone, la fonction F est donc nécessairement très simple (quels que soient les poids) et ne peut pas « coller » parfaitement aux données. Nous avons séparé au mieux les ronds bleus des carrés rouges par une droite. Avec un réseau plus complexe, et donc une frontière plus compliquée, nous aurions pu « coller » parfaitement aux données. Cependant est-ce vraiment ce que nous souhaitons faire ? Pour les données de notre exemple, avoir une séparation par une droite semble raisonnable. Peut-être que les points transfuges sont des erreurs de mesure.

2. Exemples

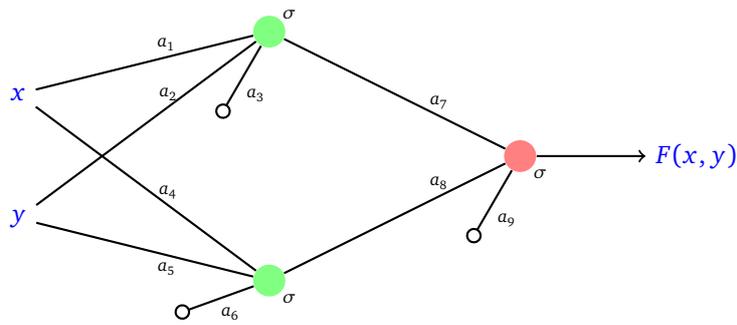
2.1. Le « ou exclusif »

Rappelons le problème du « ou exclusif » : il s'agit de trouver un réseau de neurones dont la fonction associée F vaut 1 en $(0, 1)$ et $(1, 0)$ (les carrés rouges) et vaut 0 en $(0, 0)$ et $(1, 1)$ (les ronds bleus).



Réseau.

On a vu qu'il existe une solution avec un réseau à deux couches de fonction d'activation la fonction de Heaviside (voir le chapitre « Réseau de neurones »). On cherche à l'aide de la machine quels poids pourraient convenir avec la fonction d'activation sigmoïde :



Il y a donc 9 coefficients à déterminer. La fonction d'erreur moyenne est :

$$E = \frac{1}{4} \left((F(0,0) - 0)^2 + (F(1,1) - 0)^2 + (F(0,1) - 1)^2 + (F(1,0) - 1)^2 \right)$$

sachant que F et donc l'erreur E dépendent des coefficients a_1, \dots, a_9 . Il s'agit de trouver ceux qui rendent l'erreur E minimale.

Descente de gradient. On applique la méthode de la descente de gradient à la fonction E pour les variables (a_1, \dots, a_9) . On utilise la descente de gradient classique avec un pas $\delta = 1$. Le choix des poids initiaux est déterminant pour les résultats. On choisit comme poids de départ :

$$P_0 = (a_1, \dots, a_9) = (1.0, 2.0, -3.0, -3.0, -2.0, 1.0, 1.0, 1.0, -1.0).$$

L'erreur initiale vaut $E(P_0) = 0.2961$. Le gradient $\text{grad } E(P_0)$ vaut

$$(0.00349, -0.00303, -0.00832, -0.00696, -0.01350, -0.01047, -0.00324, 0.01258, -0.04671).$$

Avec $\delta = 1$, la première itération modifie un petit peu les coefficients :

$$P_1 = (0.9965, 2.0030, -2.9916, -2.9930, -1.9864, 1.0104, 1.0032, 0.9874, -0.9532),$$

l'erreur a légèrement diminué et vaut maintenant $E(P_1) = 0.2935$. La seconde itération donne :

$$P_2 = (0.9925, 2.0055, -2.9839, -2.9860, -1.9731, 1.0206, 1.0053, 0.9738, -0.9105)$$

et l'erreur vaut maintenant $E(P_2) = 0.2911$.

Au bout de 1000 itérations :

$$P_{1000} = (3.5623, 3.5675, -5.5344, -5.3682, -5.3935, 1.8403, -6.3234, -6.3801, 3.1216)$$

et l'erreur vaut maintenant $E(P_{1000}) = 0.0097$.

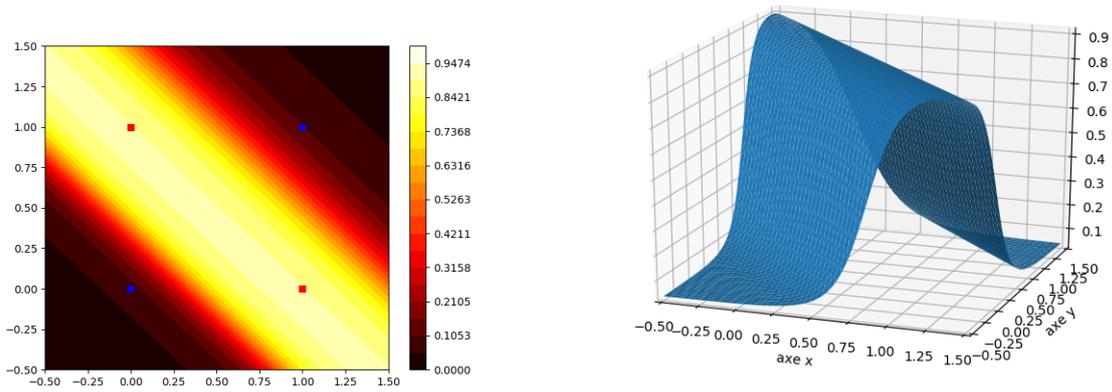
Validation et prédiction. Au bout de 1000 itérations notre réseau de neurones est assez performant. La fonction $F(x, y)$ obtenue vérifie :

$$F(0,0) = 0.08, \quad F(1,1) = 0.10, \quad F(1,0) = 0.89, \quad F(0,1) = 0.89.$$

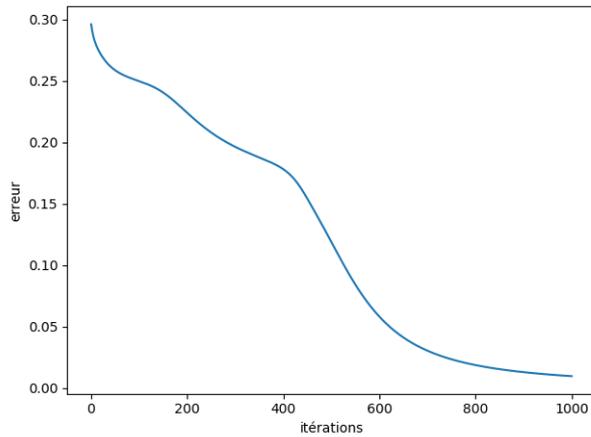
Les ronds bleus sont donc clairement séparés des carrés rouges par les valeurs de F .

Quelle couleur est-il naturel d'associer au points $(0.2, 0.9)$? On calcule $F(0.2, 0.9) = 0.87$ qui est assez proche de 1, il est donc naturel de le colorier en rouge.

Visualisation graphique. Voici deux représentations graphiques de la fonctions F obtenue. À gauche par les lignes de niveau dans le plan et à droite par son graphe dans l'espace. La fonction F prend des valeurs proches de 1 autour de la droite passant par $(1, 0)$ et $(0, 1)$ et se rapproche de 0 partout ailleurs.



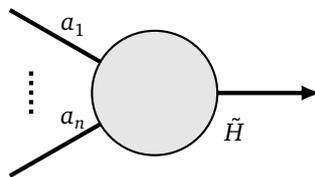
Analyse. Voici l'évolution de l'erreur en fonction du nombre d'itérations. L'erreur décroît au fil des itérations : c'est le principe de la descente de gradient !



Comparer les poids obtenus avec ceux du « ou exclusif » définis dans le chapitre « Réseau de neurones ».

2.2. Le perceptron et la règle de Hebb

Nous revenons sur l'exemple historique du cas d'un seul neurone (sans biais). Que donne la rétropropagation dans ce cas très simple ?



Nous souhaitons une fonction d'activation du type marche Heaviside, qui renvoie 0 ou 1, mais pour la descente du gradient nous avons besoin d'une dérivée non nulle. Nous allons imaginer qu'il existe une fonction marche de Heaviside virtuelle \tilde{H} telle que

$$\tilde{H}(x) = 0 \quad \text{si } x < 0 \quad \tilde{H}(x) = 1 \quad \text{si } x \geq 0 \quad \text{et} \quad \tilde{H}'(x) = 1 \quad \text{pour tout } x \in \mathbb{R}.$$

Il est clair qu'une telle fonction n'existe pas (car la dérivée d'une fonction constante par morceaux est toujours nulle) mais faisons comme si c'était le cas.

Ce neurone définit une fonction F telle que $F(x_1, \dots, x_n) = \tilde{H}(a_1x_1 + \dots + a_nx_n)$. Comme d'habitude, nous avons des données (X_i, z_i) , ici $z_i = 0$ ou bien $z_i = 1$. Il s'agit de trouver les poids (a_1, \dots, a_n) tels que $F(X_i) \simeq z_i$. L'erreur locale est donnée par $E_i = (F(X_i) - z_i)^2$.

La descente de gradient (stochastique) pour la donnée i s'écrit :

$$P_{k+1} = P_k - \delta \text{grad } E_i(P_k).$$

On a :

$$\frac{\partial E_i}{\partial a_j}(X_i) = 2 \frac{\partial F}{\partial a_j}(X_i)(F(X_i) - z_i).$$

Mais $\frac{\partial F}{\partial a_j}(X_i) = x_j$ car $\tilde{H}'(x) = 1$ et comme $F(X_i)$ et z_i valent 0 ou 1, alors $\frac{\partial E_i}{\partial a_j}(X_i)$ vaut $\pm 2x_j$ ou 0. Autrement dit, $\text{grad } E_i = 2\epsilon(x_1, \dots, x_n) = 2\epsilon X_i$ avec $\epsilon = 0, 1$ ou -1 .

Nous avons ainsi obtenu la **règle de Hebb**, qui est en fait l'ancêtre de la rétropropagation.

Règle de Hebb.

- On fixe un pas $\delta > 0$.
- On part d'un poids P_0 (choisi au hasard par exemple).
- On calcule par récurrence les poids P_k en parcourant la liste (X_i, y_i) et en distinguant les cas suivants :
 - si la sortie prédite $F(X_i)$ vaut la sortie attendue z_i alors on ne change rien :

$$P_{k+1} = P_k,$$

- si la sortie prédite $F(X_i)$ vaut 1 alors que la sortie attendue z_i vaut 0 alors :

$$P_{k+1} = P_k - 2\delta X_i,$$

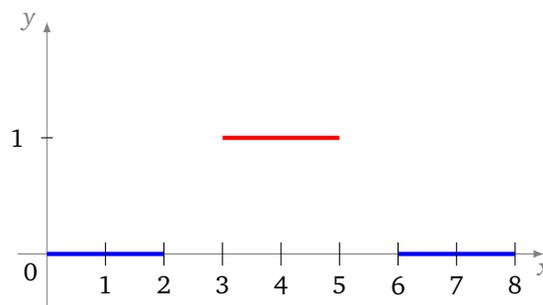
- si la sortie prédite $F(X_i)$ vaut 0 alors que la sortie attendue z_i vaut 1 alors :

$$P_{k+1} = P_k + 2\delta X_i.$$

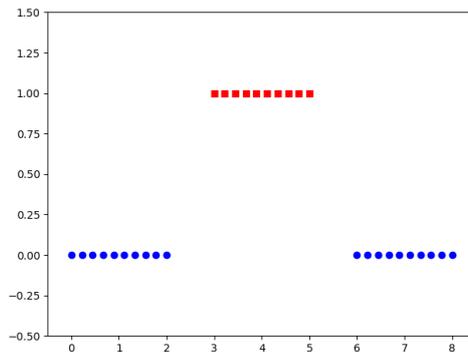
- Une fois toutes les données (X_i, z_i) utilisées, on recommence depuis la première donnée.
- On s'arrête au bout d'un nombre d'itérations fixé à l'avance.
- Le dernier poids obtenu $P_k = (a_1, \dots, a_n)$ définit le perceptron et la fonction associée qui répond au problème est : $F(x_1, \dots, x_n) = 1$ si $a_1 x_1 + \dots + a_n x_n \geq 0$ et $F(x_1, \dots, x_n) = 0$ sinon.

2.3. Une fonction marche

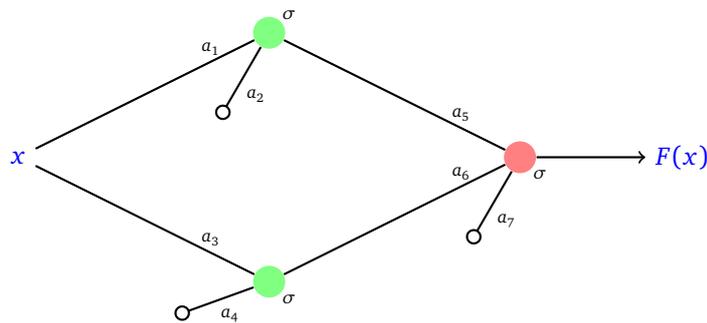
Nous avons vu dans le chapitre « Réseau de neurones » qu'il est facile de réaliser des fonctions « marche » à l'aide d'un réseau simple et de la fonction de Heaviside. Nous souhaitons construire un réseau de neurones dont la fonction $x \mapsto F(x)$ réalise au mieux les contraintes suivantes : $F(x)$ vaut 0 sur $[0, 2]$, puis vaut 1 sur $[3, 5]$ et de nouveau vaut 0 sur $[6, 8]$ (il y a une certaine liberté sur $[2, 3]$ et $[5, 6]$).



Données. On décide de placer 10 ronds bleus espacés régulièrement sur $[0, 2]$, la même chose sur $[6, 8]$, là où la fonction doit être nulle, et également 10 carrés rouges espacés régulièrement sur $[3, 5]$ là où la fonction doit valoir 1. Ces points fournissent donc les $N = 30$ données d'apprentissage.



Réseau. On décide d'utiliser un réseau avec deux neurones sur la couche d'entrée et un neurone sur la couche de sortie, tous utilisant la fonction d'activation σ .



Il y a 7 coefficients à déterminer. La fonction d'erreur moyenne E est la somme des $(F(x_i) - 0)^2$ pour les x_i abscisses des ronds bleus et des $(F(x_i) - 1)^2$ pour les x_i abscisses des carrés rouges, le tout divisé par $N = 30$.

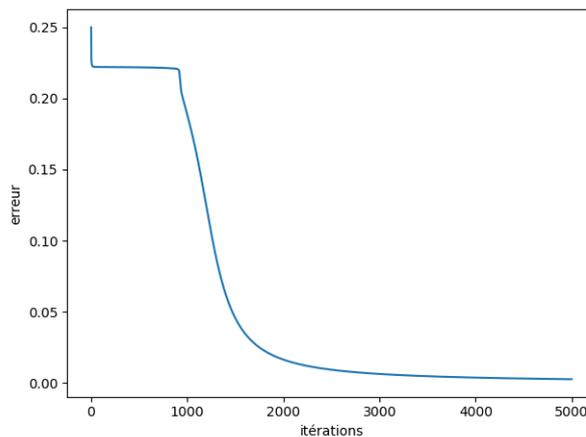
Descente de gradient. On applique la descente de gradient à la fonction E de variables (a_1, \dots, a_7) pour un pas $\delta = 1$ et un choix arbitraire de poids initiaux :

$$P_0 = (a_1, \dots, a_7) = (0.0, 1.0, 0.0, -1.0, 1.0, 1.0, -1.0).$$

Au bout de 5000 itérations, on obtient les poids :

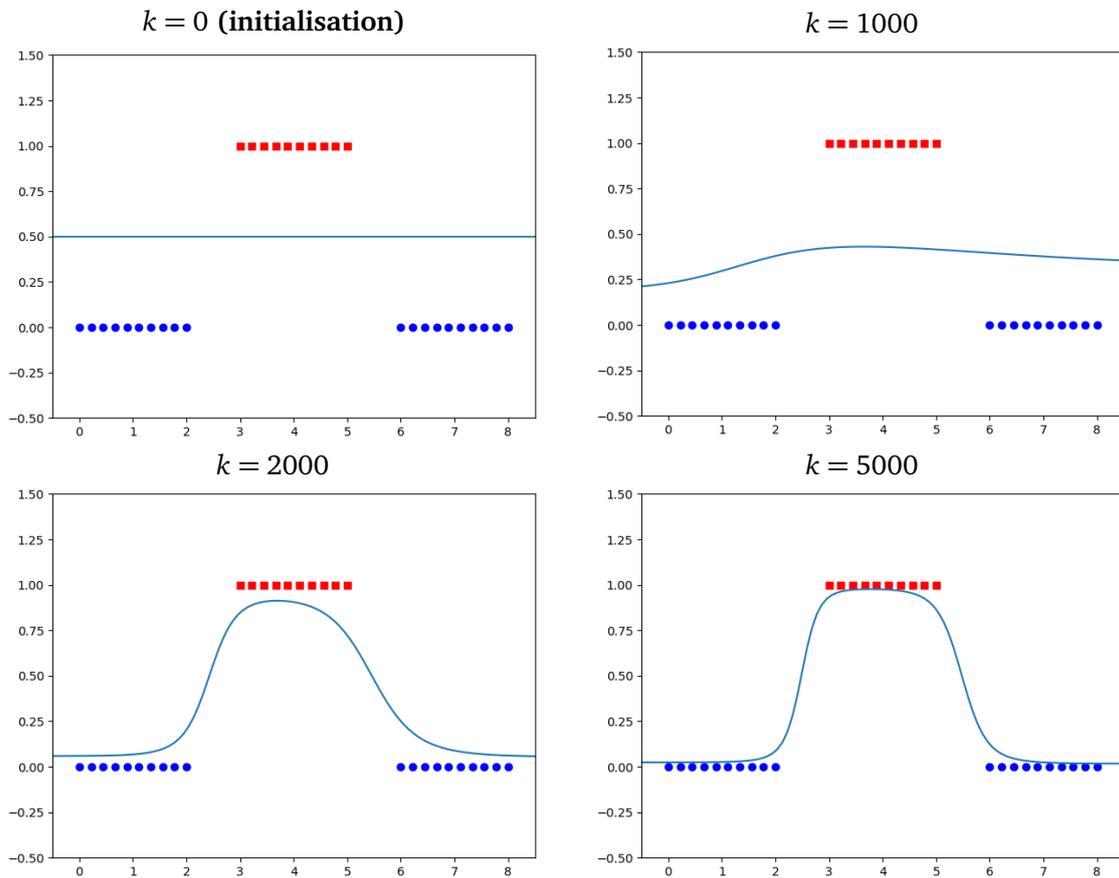
$$P_{5000} = (a_1, \dots, a_7) = (-2.0142, 10.9964, 3.0543, -7.7086, 8.2334, 7.8735, -11.9037).$$

Voici l'évolution de l'erreur en fonction du nombre d'itérations.



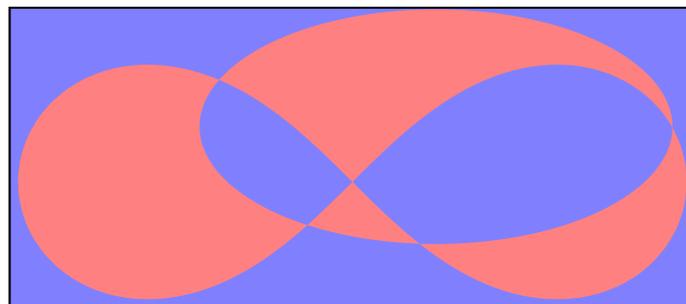
Visualisation graphique.

Voici le graphe de la fonction $x \mapsto F(x)$ au bout de différents nombres d'itérations.

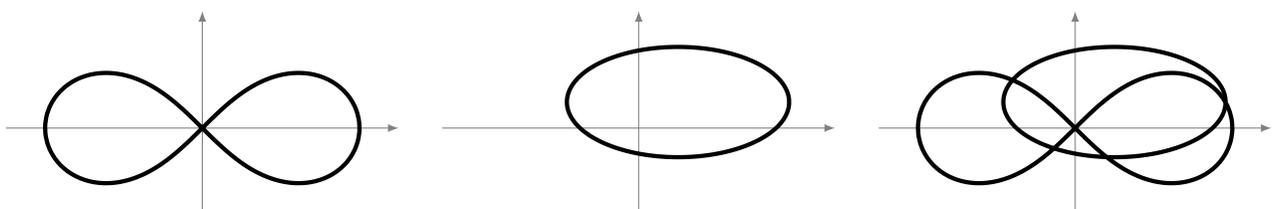


2.4. Plus de neurones

Le but est ici de trouver un réseau de neurones qui distingue deux zones compliquées du plan : la zone bleue et la zone rouge.



Voici comme sont construites ces zones : on part d'une lemniscate de Bernoulli d'équation $(x^2 + y^2)^2 = 4(x^2 - y^2)$ et d'une ellipse d'équation $(x - \frac{1}{2})^2 + 4(y - \frac{1}{3})^2 = 2$.



L'union de ces deux courbes a pour équation :

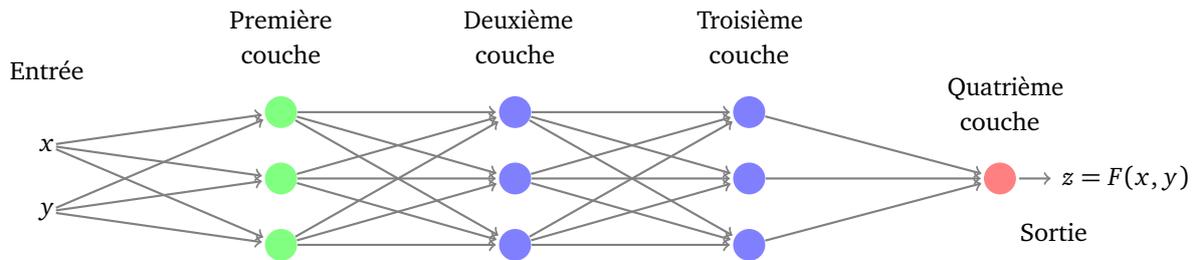
$$f(x, y) = ((x^2 + y^2)^2 - 4(x^2 - y^2)) \cdot ((x - \frac{1}{2})^2 + 4(y - \frac{1}{3})^2 - 2) = 0.$$

La zone rouge correspond aux points de coordonnées (x, y) pour lesquels $f(x, y) \leq 0$ et la zone bleue à $f(x, y) > 0$. Nous allons limiter notre étude à une zone rectangulaire. Le rectangle est choisi de sorte que l'aire bleue et l'aire rouge soient à peu près égales.

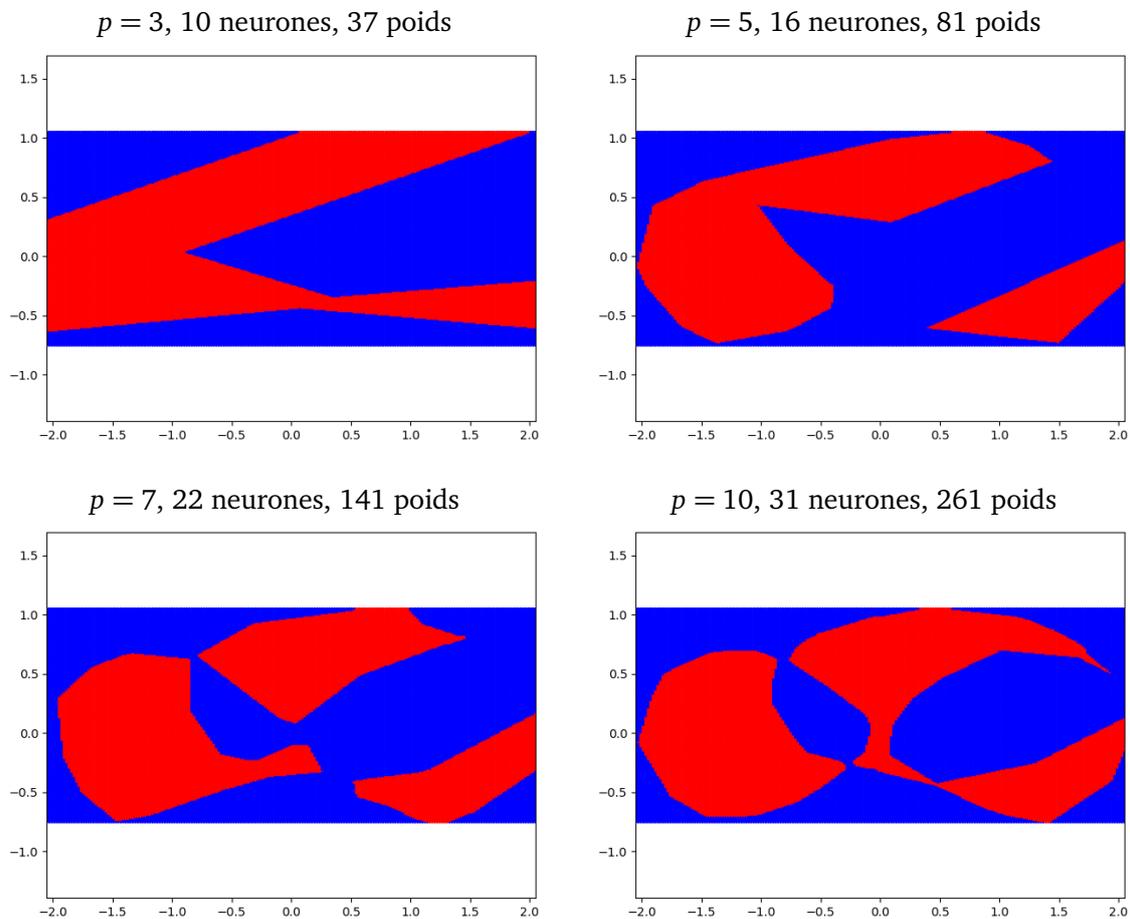
Objectifs. On oublie maintenant la fonction f et on ne retient que quelques points rouges et quelques points bleus. On cherche un réseau et une fonction F telle que $F(x, y) \simeq 1$ pour les points rouges et $F(x, y) \simeq 0$ pour les points bleus.

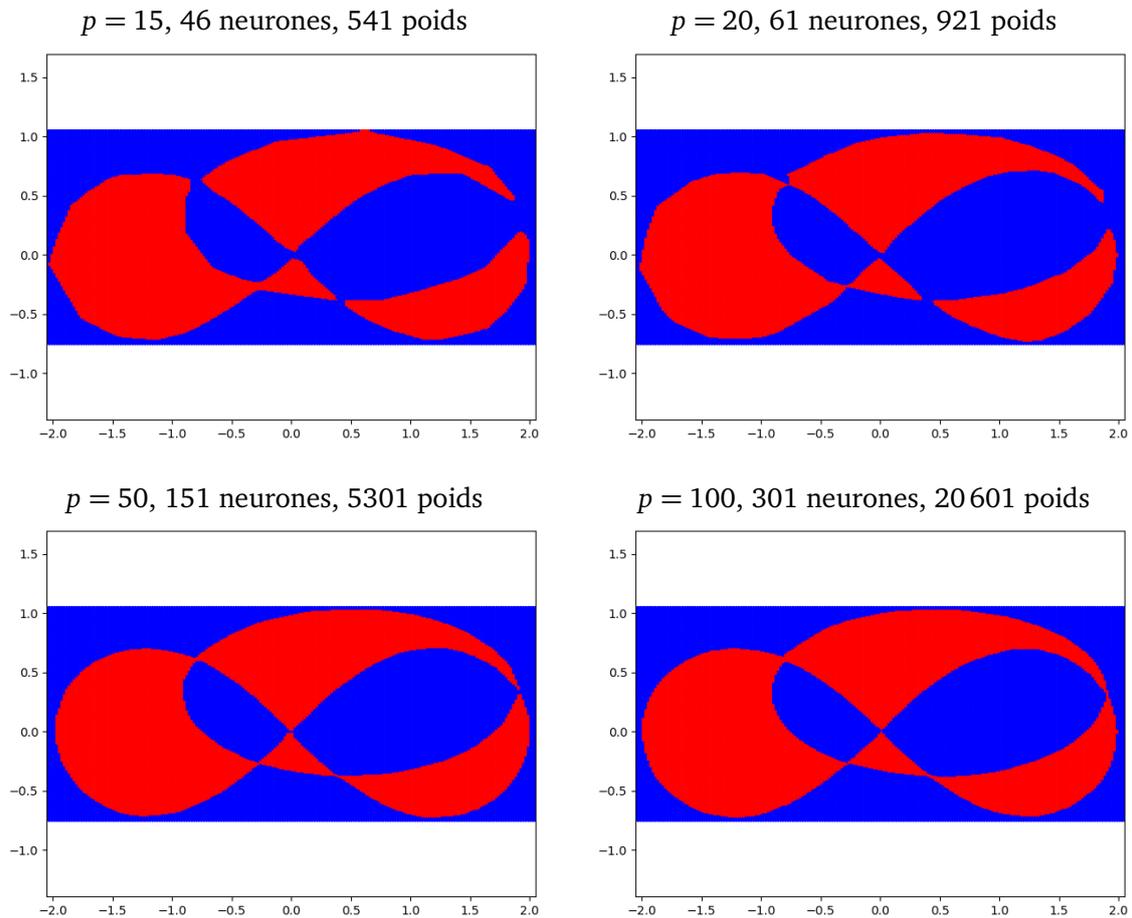
Données. On divise notre rectangle en une grille de $n \times n$ points. Les exemples ci-dessous sont calculés pour $n = 200$. Nous avons donc 40 000 points (x, y) , repartis (à peu près) équitablement entre rouge ($z = 1$) et bleu ($z = 0$).

Architecture. On décide de concevoir un réseau à 4 couches, avec le même nombre p de neurones par couche pour les trois premières couches et un seul neurone sur la couche de sortie. La fonction d'activation choisie pour tous les neurones est ReLU. Voici une illustration de la configuration pour $p = 3$.



Poids. On calcule les poids avec une méthode de descente de gradient stochastique. Les poids initiaux sont choisis aléatoirement, le nombre d'itérations est suffisamment grand. Le réseau obtenu après calculs fournit donc une fonction F . On colorie en rouge les points pour lesquels $F(x, y) \simeq 1$ (en fait, là où $F(x, y) \geq \frac{1}{2}$) et en bleu les points pour lesquels $F(x, y) \simeq 0$ (en fait, là où $F(x, y) < \frac{1}{2}$). On regarde si le résultat obtenu est proche de la situation envisagée. On présente ici quelques résultats typiques intéressants (les résultats pouvant varier selon le choix aléatoire des poids initiaux).



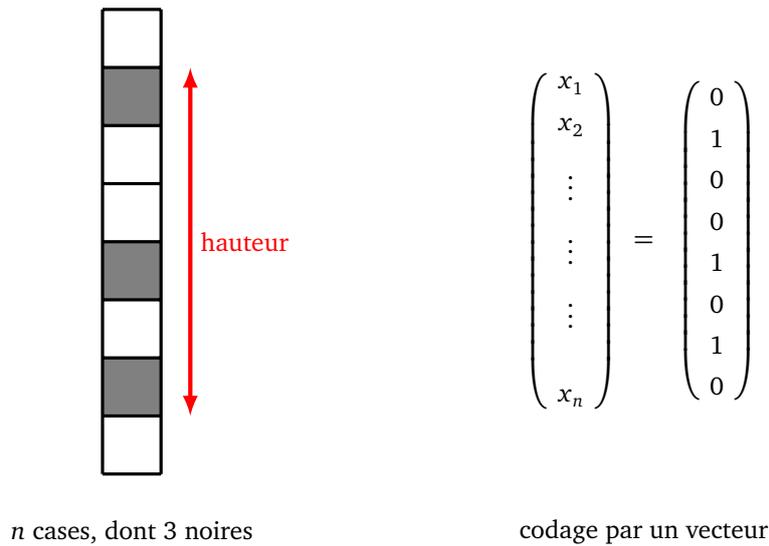


Conclusion : il faut un nombre assez grand de neurones pour pouvoir modéliser des phénomènes compliqués. Dans l'exemple traité ici, on obtient une bonne approximation à partir de $p = 20$, soit plus de 60 neurones et environ 1000 poids.

2.5. Apprentissage

N'oublions pas le but principal des réseaux de neurones : apprendre à partir des données afin de prédire des résultats pour des situations nouvelles.

Voici un problème pour lequel nous allons tester si les prédictions sur des données nouvelles sont correctes ou pas. On dispose de n cases, dont 3 sont noircies. On appelle *hauteur*, le nombre total de cases entre la plus haute et la plus basse case noircie. Si les trois cases ont pour rang $i < j < k$ alors la hauteur vaut $h = k - i + 1$ (la case noircie du milieu n'intervient pas dans la formule). Sur le dessin ci-dessous, $n = 8$ et la configuration représentée est de hauteur 6.



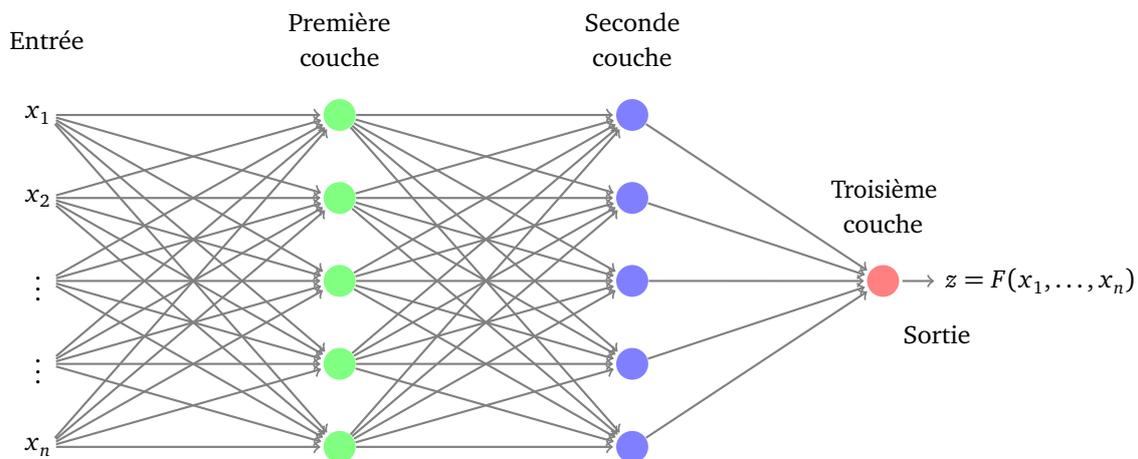
Nous voulons tester si un réseau de neurones permet de retrouver un résultat qu'on peut vérifier ici à l'aide d'une formule simple : étant donné une configuration il s'agit de faire calculer à la machine quelle est sa hauteur.

Il y a en tout $N = \frac{n(n-1)(n-2)}{6}$ configurations possibles pour les trois cases noires parmi n cases. On décide d'utiliser la moitié des configurations comme données d'apprentissage et l'autre moitié va nous permettre de tester si le réseau calculé avec les données d'apprentissage se comporte « correctement » sur les nouvelles données.

Dans la suite nous prendrons $n = 20$ cases, ce qui fait $N = 1140$ configurations possibles.

Données en entrées. On calcule les N configurations possibles. Après un mélange aléatoire, on ne retient que $N/2$ configurations pour l'apprentissage. Une configuration est codée sous la forme d'un vecteur $X = (x_1, x_2, \dots, x_n)$ avec $x_i = 0$ (case blanche) ou $x_i = 1$ (case noire). Pour chaque configuration X des données d'apprentissage, on calcule la hauteur $z = h(X)$. Les données d'apprentissage sont les $N/2$ données (X_k, z_k) formées des entrées X_k et de la sortie attendue z_k correspond à la hauteur.

Réseau. Le réseau comporte n entrées : une entrée par case. Il est constitué de 3 couches. Les deux premières couches possèdent p neurones et la couche de sortie un seul. La fonction d'activation est la fonction ReLU pour tous les neurones. Sur la figure ci-dessous est illustré le cas $p = 5$.



Apprentissage. Le réseau est initialisé avec des poids aléatoires. Ensuite ces poids sont modifiés par une descente de gradient stochastique avec un nombre suffisant d'itérations.

Sortie prédite. Le réseau paramétré fournit une fonction F à valeur réelle. La sortie attendue étant un entier, on décide que la sortie prédite sera arrondie à l'entier le plus proche. Pour chaque X_k , la prédiction est correcte si la valeur arrondie $F(X_k)$ vaut la hauteur $h(X_k)$.

Tests. Il faut tester l'efficacité du réseau : tout d'abord le réseau modélise-t-il correctement les données d'apprentissage ? En effet, même si la fonction F a été construite dans le but d'avoir $F(X_k) \simeq z_k$, il n'y a pas nécessairement égalité pour toutes les données d'apprentissage. Mais surtout, il faut tester si la fonction F donne de bonnes prédictions pour de nouvelles données. Nous disposons pour cela des $N/2$ données de test non utilisées lors de l'apprentissage.

Les résultats dépendent des poids initiaux (aléatoires) et de la liste des configurations choisies pour l'apprentissage (liste qui a été mélangée), on ne retient que les meilleurs résultats parmi plusieurs essais. Par exemple, pour $p = 10$, voici un des meilleurs résultats obtenus :

- *Apprentissage.* 560 données prédites correctement sur un total de de 570 données d'apprentissage, soit 98% de réussite.
- *Test.* 506 données prédites correctement sur un total de de 570 données de test, soit 89% de réussite.

Voici quelques résultats pour différentes valeurs de p .

p	neurones	poids	pourcentage apprentissage	pourcentage test
3	7	79	50%	45%
5	11	141	85%	75%
7	15	211	90%	85%
10	21	331	98%	90%
15	31	571	99%	90%
20	41	861	100%	85%

Commentaires. Plus il y a de neurones, plus la fonction F modélise correctement les données d'apprentissage : à partir de 20 neurones ($p \geq 10$), la modélisation est quasi-parfaite. Les pourcentages de réussite pour les données de test sont toujours inférieures et plafonnent à 90%. Un phénomène nouveau apparaît avec plus de 40 neurones (pour $p = 20$), les pourcentages de réussite sur les tests régressent par rapport à des réseaux ayant moins de neurones, bien que les données d'apprentissage soient parfaitement modélisées : c'est un phénomène de sur-apprentissage.

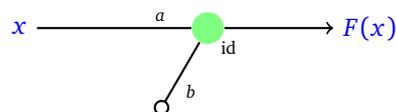
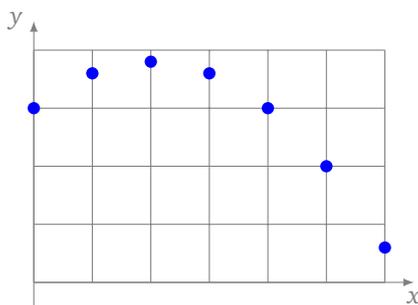
3. Sur-apprentissage et autres soucis

Nous allons voir différents problèmes qui peuvent intervenir dans le calcul des poids d'un réseau de neurones, le problème le plus subtil étant le *sur-apprentissage*.

3.1. Modèle insuffisant

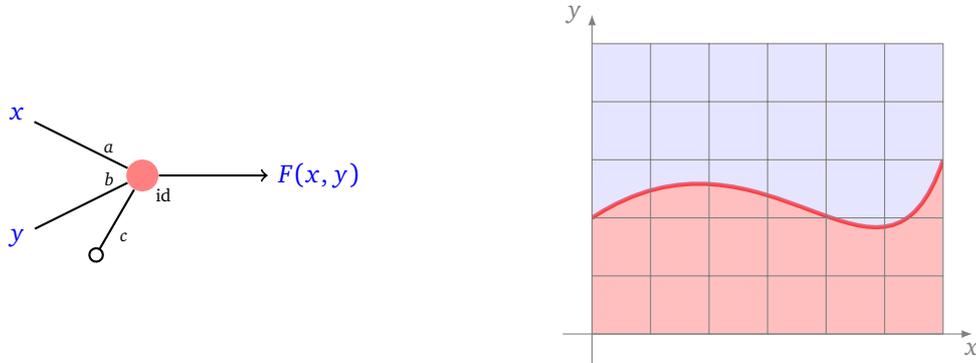
Problème. Voici deux exemples de situations dans lesquelles le choix de l'architecture du réseau pose problème, car le réseau est trop simple.

Exemple 1. Imaginons que nous voulions modéliser une situation à partir des données fournies par les points (x_i, y_i) du plan (figure de gauche). Si on construit un réseau d'un seul neurone (figure de droite), alors la fonction de sortie sera du type $y = F(x) = ax + b$.



Aucun paramètre (a, b) ne permettra une modélisation correcte de la situation, car les points ne sont franchement pas alignés.

Exemple 2. Le même problème se produirait si on voulait trouver une fonction F , construite à partir d'un seul neurone, valant 0 pour la zone bleue et 1 pour la zone rouge de la figure ci-dessous. Ce n'est pas possible car la frontière de la solution n'est pas linéaire.

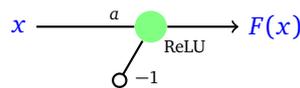


Conclusion. Dans ces situations, le problème ne se situe pas au niveau des poids, augmenter la taille des données ou le nombre d'itérations n'y changera rien. La conception du réseau est mauvaise pour répondre à la question posée. C'est comme vouloir faire une course de voitures avec une deux-chevaux. La solution est de changer l'architecture du réseau, par exemple en rajoutant des neurones.

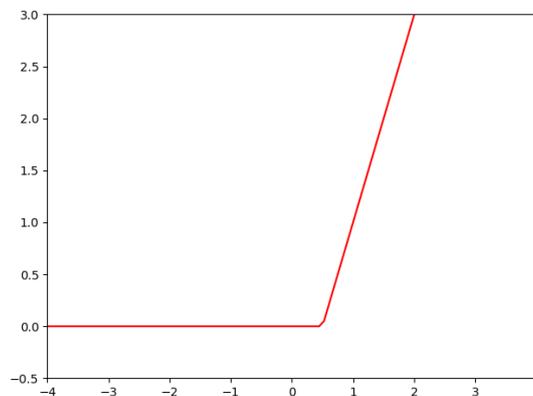
3.2. Minimum local

Problème. La descente de gradient a pour objectif de fournir un minimum local. Cependant rien n'affirme que ce minimum local est un minimum global.

Exemple. Voici un réseau composé d'un seul neurone de fonction d'activation ReLU et possédant un seul coefficient à déterminer, le biais étant imposé à la valeur -1 .

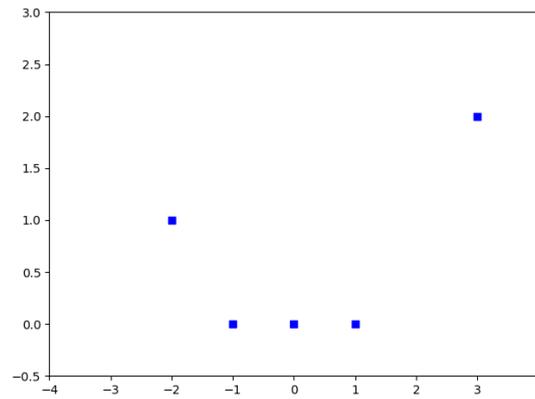


Le graphe de la fonction F correspondant à ce neurone est une portion de la droite d'équation $y = ax - 1$, prolongée par l'axe des abscisses. Voici l'exemple du graphe de F , avec $a = +2$.

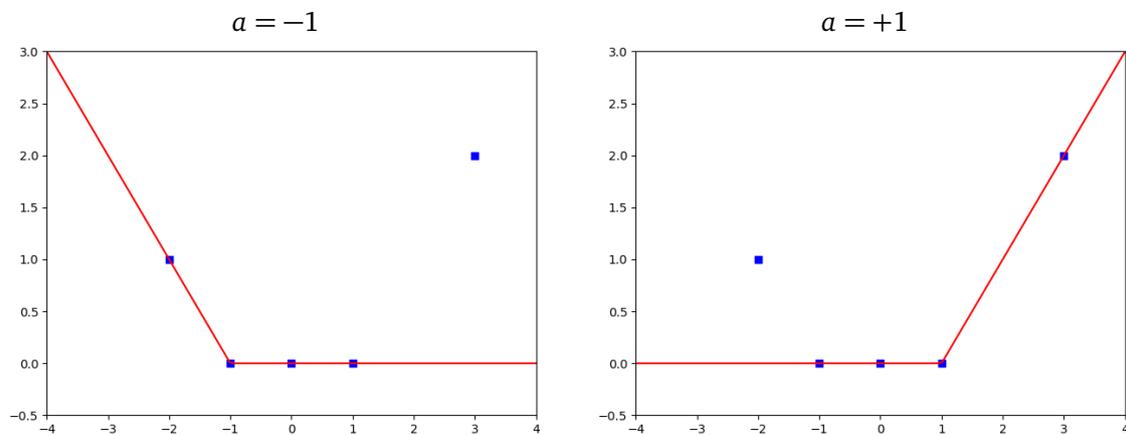


Les données sont des points déterminés par leurs coordonnées (x_i, y_i) :

$$(-2, 1), \quad (-1, 0), \quad (0, 0), \quad (1, 0), \quad (3, 2).$$

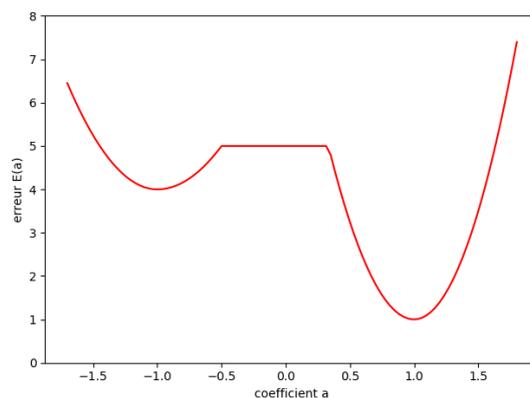


Il s'agit de trouver le meilleur coefficient a , qui définisse F tel que $F(x_i) \simeq y_i$. Autrement dit, on souhaite minimiser $E(a) = \sum E_i(a)$ où $E_i(a) = (F(x_i) - y_i)^2$. Géométriquement il y a deux paramètres qui semblent meilleurs que les autres $a = -1$ (figure de gauche) et $a = +1$ (figure de droite) car alors les portions de droites passent par des carrés bleus.



Cette intuition se vérifie lorsque l'on trace le graphe de la fonction d'erreur $a \mapsto E(a)$. La fonction possède deux minimums locaux en $a = -1$ et $a = +1$ (et aussi une portion constante, due à l'usage de la fonction ReLU).

Erreur $E(a)$ en fonction du coefficient a



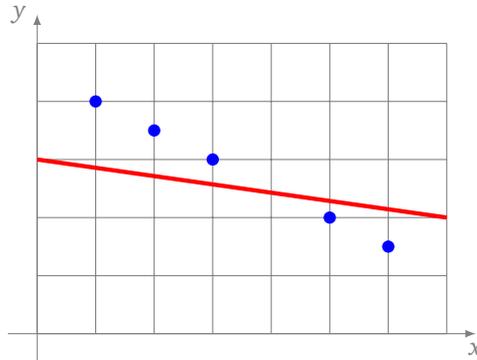
Le minimum en $a = +1$ est le minimum global. Mais si on applique la descente de gradient en partant d'un coefficient initial $a_0 < 0$, il y a de grandes chances d'arriver au minimum local $a = -1$, qui ne sera pas la solution optimale.

Conclusion. Vous êtes une fourmi qui se promène dans une boîte d'œufs avec des trous de différentes profondeurs : vous ne pouvez pas voir à l'avance quel est l'emplacement le plus profond ! Une solution peut être de tester différentes valeurs initiales, dans le but d'obtenir différents minimums locaux.

3.3. Sous-apprentissage

Problème. Le *sous-apprentissage* révèle une conception correcte de l'architecture du réseau mais une mauvaise mise en œuvre. On obtient alors des poids qui ne répondent pas correctement au problème.

Exemple. Une droite obtenue par un réseau approche mal une suite de points pourtant alignés.



Cela peut être dû aux raisons suivantes :

- les données ne sont pas en nombre suffisant,
- le nombre d'itérations est insuffisant,
- le pas δ est trop grand.

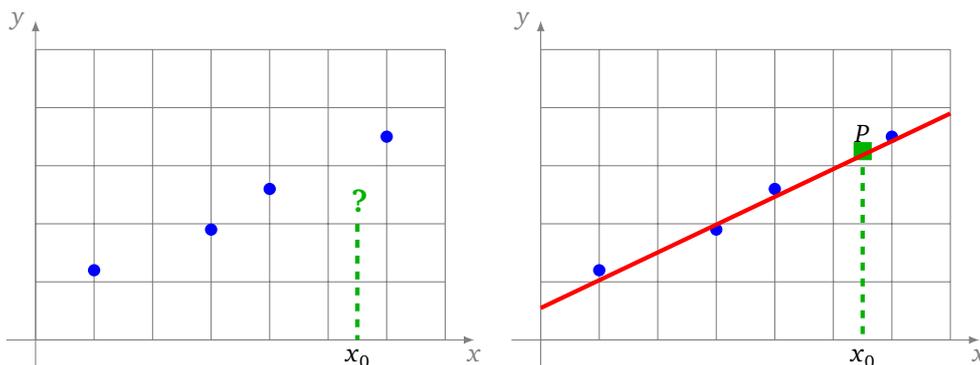
Conclusion. Il est difficile de savoir à l'avance quelle est la bonne taille des données à utiliser et combien d'itérations sont nécessaires pour arriver à un modèle correct. Le sous-apprentissage, c'est comme faire une course de voiture avec une porsche mais en ne dépassant jamais les 80 km/h ! A posteriori, la solution est simple : ajouter des données, augmenter le nombre d'itérations ou diminuer la pas.

3.4. Sur-apprentissage

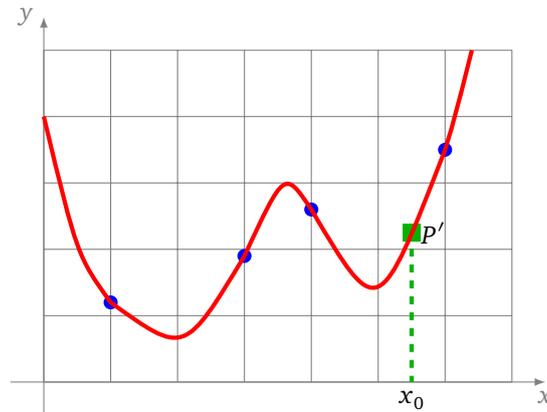
Problème. Le modèle obtenu « colle » parfaitement aux données d'apprentissage, mais cependant les prédictions pour de nouvelles valeurs sont mauvaises. Il s'agit donc d'un problème délicat : la fonction F obtenue vérifie bien $F(X_i) \simeq z_i$ pour toutes les données, mais pour une nouvelle entrée X , la sortie $F(X)$ n'est pas une bonne prédiction. Cela se produit lorsque l'on se concentre uniquement sur l'apprentissage à partir des données, mais que l'on a oublié que le but principal est la prédiction.

Exemple 1. Nous avons 4 points bleus. Pour une valeur x_0 donnée, on souhaite prédire une valeur y_0 (figure de gauche) cohérente avec nos données.

On propose dans un premier temps d'approcher la solution en utilisant une droite (figure de droite) même si les points ne sont pas exactement alignés. Cela permet de prédire une valeur y_0 pour placer le point $P = (x_0, y_0)$. Ce modèle ne sera jamais parfait car les points ne sont pas alignés, donc aucune droite ne convient exactement, autrement dit l'erreur ne sera jamais nulle.

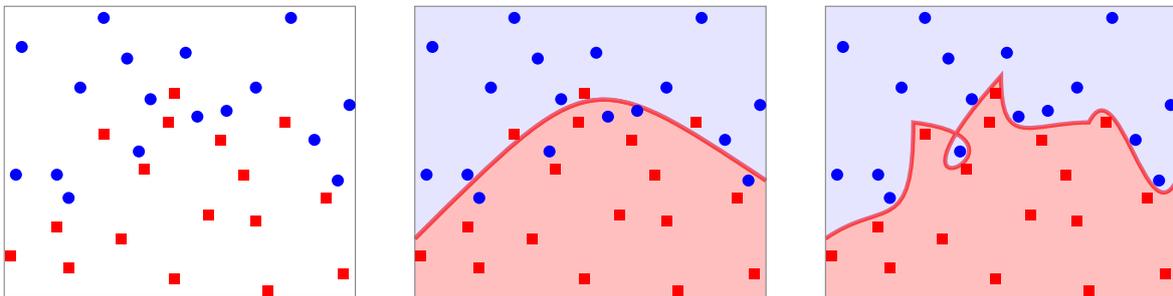


On peut construire un modèle plus compliqué, par exemple chercher une courbe polynomiale de degré 3 ou plus qui passe *exactement* par tous les points d'apprentissage. Ainsi, pour cette courbe, l'erreur sera nulle. Cette courbe permet de prédire une valeur y'_0 pour placer un point $P'(x_0, y'_0)$.



Ce dernier modèle répond entièrement au problème, mais la solution proposée ne semble pas raisonnable par rapport aux données de départ. C'est un cas de sur-apprentissage : le modèle est correct sur les données, mais les prédictions seront mauvaises.

Exemple 2. Le problème est similaire si on essaye de séparer les ronds bleus des carrés rouges de la figure de gauche ci-dessous. Un modèle simple permet à une parabole de séparer l'essentiel des ronds bleus des carrés rouges (figure centrale). On peut entraîner un modèle plus complexe pour qu'il délimite parfaitement les deux types de points, mais au prix d'une complexité non nécessairement voulue (figure de droite).



Conclusion. Le sur-apprentissage, c'est comme emmener ses enfants à l'école en conduisant à 200 km/h : cela répond de façon correcte à un problème, mais cela provoque beaucoup d'autres ennuis ! Quelle est la meilleure solution entre un modèle simple mais imparfait et un modèle parfait mais compliqué ? Il n'y a pas une réponse immédiate à la question : pour le savoir, il faut tester le modèle sur d'autres données, jamais rencontrées précédemment, afin de déterminer si le réseau n'a pas été sur-entraîné.