

# Apprentissage profond

## Contents

### Introduction

- [Introduction aux réseaux de neurones](#)
- [Perceptrons multicouches](#)
- [Réseaux convolutifs](#)

Cours et notebooks Jupyter en Python sur quelques méthodes d'apprentissage profond.

Les exemples sont codés à l'aide de [Pytorch](#)

## Introduction aux réseaux de neurones

### Réseaux de neurones et apprentissage automatique

Les réseaux de neurones artificiels sont des techniques issues du domaine du connexionisme. Le courant connexionniste insiste sur le grand nombre de connexions (sous forme de réseau) réalisées entre les différents automates que sont les neurones. Le connexionisme permet :

- de disposer de nouveaux moyens de calcul : conversion de l'information des systèmes avec des applications pratiques ;
- de modéliser des phénomènes biologiques pour en apprendre davantage sur le cerveau en l'observant comme si c'était une machine de traitement électrique.

La démarche des réseaux de neurones s'oppose en certains points à celle de l'intelligence artificielle basée règles, qui sont manipulées selon les techniques de la logique formelle afin de fournir une représentation explicite du raisonnement. Cette méthodologie implique une approche « descendante » : elle part de l'analyse de la manière dont l'être humain procède pour résoudre des problèmes ou pour les apprendre, et tente de restituer cette démarche en la décomposant en unités élémentaires. Les réseaux de neurones, eux, procèdent selon une approche « ascendante » qui tente de produire des phénomènes complexes à partir d'opérations très élémentaires.

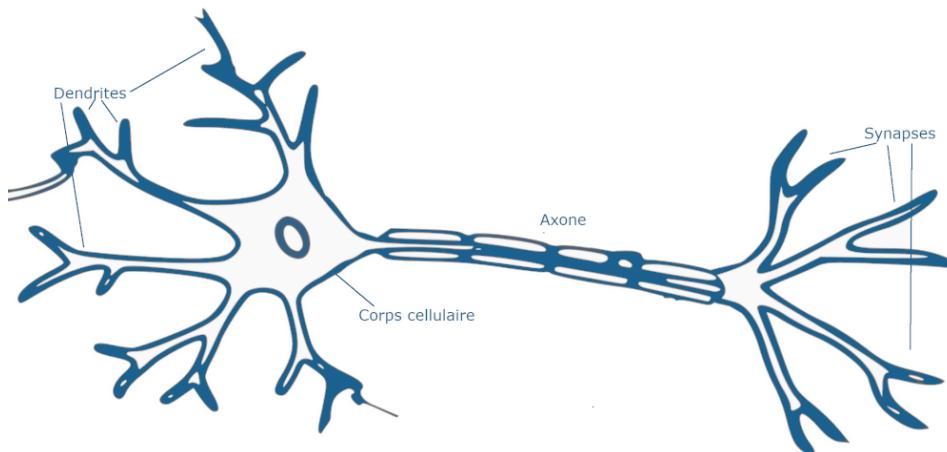
### Du neurone biologique au neurone formel

La reconnaissance du fait que le cerveau fonctionne de manière entièrement différente de celle d'un ordinateur conventionnel a joué un rôle très important dans le développement des réseaux de neurones artificiels. Les travaux effectués pour essayer de comprendre le comportement du cerveau humain ont mené à représenter celui-ci par un ensemble de composants structurels appelés neurones, massivement interconnectés entre eux. Le cerveau humain en contient en moyenne une dizaine de milliards, chacun d'entre eux étant connecté, encore une fois en moyenne, connecté à dix mille autres.

Le neurone biologique est composé de quatre parties distinctes ([Fig.1](#)) :

- le *corps cellulaire*, qui contient le noyau de la cellule nerveuse; c'est en cet endroit que prend naissance l'influx nerveux, qui représente l'état d'activité du neurone;
- les *dendrites*, ramifications tubulaires courtes formant une espèce d'arborescence autour du corps cellulaire; ce sont les entrées principales du neurone, qui captent l'information venant d'autres neurones;

- l'*axone*, longue fibre nerveuse qui se ramifie à son extrémité; c'est la sortie du neurone et le support de l'information vers les autres neurones;
- la *synapse*, qui communique l'information, en la pondérant par un poids synaptique, à un autre neurone; elle est essentielle dans le fonctionnement du système nerveux.



**Fig. 1** Neurone biologique

La transmission de l'information d'un neurone à l'autre s'effectue au moyen de l'influx nerveux, qui est constitué d'une impulsion électrique, d'une durée d'environ 2 ms et d'une amplitude de 100 mV. Une cellule nerveuse standard non sollicitée en émet en moyenne cinquante à la seconde (activité spontanée). La probabilité d'émettre une impulsion est accrue ou réduite selon que la somme pondérée des entrées du neurone est globalement excitatrice ou inhibitrice. Cette fréquence peut ainsi être portée jusqu'à 100 impulsions par seconde, pour un neurone bombardé d'effets synaptiques excitateurs ; dans le cas contraire, elle peut être réduite à néant (le neurone reste silencieux). Les effets synaptiques qui agissent sur le neurone entraînent donc une modulation de la fréquence d'émission de l'influx nerveux. Le message transmis est précisément contenu dans le nombre d'influx nerveux émis, défini par une moyenne sur quelques dizaines de ms. L'information contenue dans le cerveau, quant à elle, est représentée par les poids synaptiques attribués aux entrées de chaque neurone. Le cerveau est capable d'organiser ces neurones, selon un assemblage complexe, non-linéaire et extrêmement parallèle, de manière à pouvoir accomplir des tâches très élaborées. Du fait du grand nombre de neurones et de leurs interconnexions, ce système possède une propriété de tolérance aux fautes. Ainsi, la défectuosité d'un neurone n'entraînera aucune perte réelle d'information, mais seulement une faible dégradation en qualité de toute l'information contenue dans le système.

C'est la tentative de donner à l'ordinateur les qualités de perception du cerveau humain qui a conduit à une modélisation électrique de celui-ci. C'est cette modélisation que tentent de réaliser les réseaux de neurones artificiels, dont l'élaboration repose sur base de la définition suivante, proposée par Haykin :\|

Un réseau de neurones est un processus distribué de manière massivement parallèle, qui a une propension naturelle à mémoriser des connaissances de façon expérimentale et de les rendre disponibles pour utilisation. Il ressemble au cerveau en deux points :

1. la connaissance est acquise au travers d'un processus d'apprentissage;
2. les poids des connections entre les neurones sont utilisés pour mémoriser la connaissance

La première étude systématique du neurone artificiel est due au neuropsychiatre McCulloch et au logicien Pitts qui s'inspirèrent de leurs travaux sur les neurones biologiques.

## Classification des réseaux de neurones

Un réseau de neurones est constitué d'un grand nombre de cellules de base interconnectées. De nombreuses variantes sont définies selon le choix de la cellule élémentaire, de l'architecture et de la dynamique du réseau.

Une cellule élémentaire peut manipuler des valeurs binaires ou réelles. Les valeurs binaires sont représentées par 0 et 1 ou -1 et 1. Différentes fonctions d'activation peuvent être utilisées pour le calcul de la sortie. Le calcul de la sortie peut être déterministe ou probabiliste.

L'architecture du réseau peut être sans rétroaction, c'est à dire que la sortie d'une cellule ne peut influencer son entrée. Elle peut être avec rétroaction totale ou partielle.

La dynamique du réseau peut être synchrone : toutes les cellules calculent leurs sorties respectives simultanément. La dynamique peut être asynchrone. Dans ce dernier cas, on peut avoir une dynamique asynchrone séquentielle : les cellules calculent leurs sorties chacune à son tour en séquence ou avoir une dynamique asynchrone aléatoire.

Par exemple, si on considère des neurones à sortie stochastique -1 ou 1 calculée par une fonction à seuil basée sur la fonction sigmoïde, une interconnection complète et une dynamique synchrone, on obtient le modèle de Hopfield et la notion de mémoire associative.

Si on considère des neurones déterministes à sortie réelle calculée à l'aide de la fonction sigmoïde, une architecture sans rétroaction en couches successives avec une couche d'entrée et une couche de sortie, une dynamique asynchrone séquentielle, on obtient le modèle du Perceptron multi-couches (PMC).

## Applications

En apprentissage, les réseaux de neurones sont essentiellement utilisés pour :

- l'apprentissage supervisé ;
- l'apprentissage non supervisé ;
- l'apprentissage par renforcement.

Dans la suite, nous nous intéressons essentiellement au cas de l'apprentissage supervisé. Le cas des réseaux de neurones en apprentissage non supervisé concerne principalement les cartes de Kohonen, les machines de Boltzmann restreintes (RBM) et les autoencodeurs.

## Perceptron

### Définitions

#### Definition 1 (Neurone)

Un neurone est une fonction non linéaire, paramétrée à valeurs bornées. Les  $|D|$  variables sur lesquelles opère le neurone sont habituellement désignées sous le terme d'entrées du neurone (notées  $\{x_i, i \in [1, D]\}$ ), et la valeur de la fonction sous celui de sortie  $y$ .

Le neurone formel calcule la sortie selon la formule :

$$y = f(w_0 + \sum_{i=1}^D w_i x_i) = f(w_0 + \mathbf{w}^\top \mathbf{x})$$

où :

- $\mathbf{w} = (w_1 \dots w_D)^\top$  est le vecteur des poids synaptiques qui pondèrent les entrées du neurone,
- $w_0$  est un biais
- $\mathbf{w}^\top \mathbf{x}$  est le potentiel du neurone
- $f$  est la fonction d'activation associée au neurone.

## Definition 2 (Réseau de neurones)

Un réseau de neurones est un ensemble de neurones interconnectés. Les réseaux de neurones peuvent être visualisés par l'intermédiaire d'un graphe orienté. Chaque neurone est un noeud, les neurones étant connectés par des arêtes.

On distingue habituellement neurone d'entrée et neurone de sortie. Un neurone d'entrée calcule  $y = f(x)$  où  $x$  est une entrée unique du neurone. Les neurones de sortie prennent un nombre quelconque d'entrées. Interconnectés, l'ensemble de ces neurones calcule  $f(\mathbf{y}(x))$  dont la dimension est donnée par le nombre de neurones d'entrée et de sortie (l'entrée du réseau est acceptée par les neurones d'entrée, qui forment la rétine), et la sortie du réseau est formée par les neurones de sortie.

Le cas le plus simple est celui d'un réseau comportant un seul neurone de sortie. C'est le *perceptron*. Le perceptron est un modèle de réseau de neurones avec algorithme d'apprentissage (Rosenblatt en 1958). L'idée sous-jacente de ce modèle est le fonctionnement de la rétine, l'étude de la perception visuelle. Nous commençons par aborder le cas du perceptron linéaire à seuil.

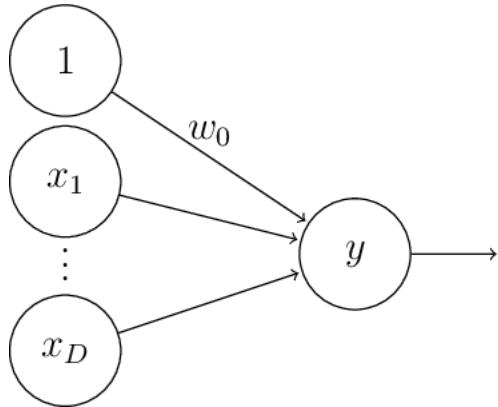
## Definition 3 (Perceptron linéaire à seuil)

Un perceptron linéaire à seuil prend en entrée  $D$  valeurs  $(x_1, \dots, x_D)$  (la rétine) et calcule une sortie  $y$ . Suivant la définition précédente, un perceptron est défini par la donnée de  $D+1$  constantes : les **poids synaptiques**  $(w_1, \dots, w_D)$  et un seuil (ou le biais)  $\theta$ . La sortie  $y$  est calculée par

$$y = \begin{cases} 1 & \text{si } \sum_{i=1}^D w_i x_i + \theta > 0 \\ 0 & \text{sinon} \end{cases}$$

Les entrées  $(x_1, \dots, x_D)$  peuvent être à valeurs dans {0,1} (ou {-1,1}) ou réelles, les poids peuvent être entiers ou réels.

Pour simplifier les notations et certaines preuves, on remplace souvent le seuil par un poids supplémentaire  $w_0$  associé à une entrée  $x_0=1$ . L'équivalence entre le modèle avec seuil et le modèle avec entrée supplémentaire à 1 est immédiate : le coefficient  $w_0$  est l'opposé du seuil  $\theta$ .



On note  $(w)$  (respectivement  $(x)$ )  $\in \mathbb{R}^{D+1}$  le vecteur des poids (resp. des entrées), augmenté de  $w_0$  (resp.  $x_0=1$ ). Comme suggéré par la définition, on peut décomposer le calcul de la sortie  $y$  en un premier calcul de la quantité  $\mathbf{w}^T \mathbf{x} = \sum_{i=0}^D w_i x_i$  appelée **potentiel post-synaptique** ou **entrée totale**, suivie d'une application d'une **fonction d'activation** sur cette entrée totale. Dans le cas du perceptron linéaire à seuil, la fonction d'activation est la fonction de Heaviside définie par  $f(x) = 1_{\{x>0\}}$  lorsque la sortie est en {0,1}, et  $g(x) = 2f(x) - 1$  lorsque la sortie est en {-1,1}.

## Utilisation : discrimination linéaire

Soit  $\{\mathcal{E}_a\}$  un ensemble d'exemples dans  $(\mathbb{R}^D)^n$ . On note  $\{\mathcal{E}_a^0 = \{(\mathbf{x}, 0) \in (\mathbf{x}, 0) \in \{\mathcal{E}_a\}\}$  et  $\{\mathcal{E}_a^1 = \{(\mathbf{x}, 1) \in (\mathbf{x}, 1) \in \{\mathcal{E}_a\}\}$

On dit que  $\{\mathcal{E}_a\}$  est **linéairement séparable** s'il existe un hyperplan  $(H)$  de  $(\mathbb{R}^D)$  tel que les ensembles  $\{\mathcal{E}_a^0\}$  et  $\{\mathcal{E}_a^1\}$  soient situés de part et d'autre de cet hyperplan.

On montre qu'un perceptron linéaire à seuil à  $(D)$  entrées divise l'espace des entrées  $(\mathbb{R}^D)$  en deux sous-espaces délimités par un hyperplan  $(w^T \mathbf{x} = \theta)$ .

Réiproquement, tout ensemble linéairement séparable peut être discriminé par un perceptron.

Un perceptron est donc un discriminant linéaire. On montre facilement qu'un échantillon de  $(\mathbb{R}^{D+1})$  est séparable par un hyperplan si et seulement si l'échantillon de  $(\mathbb{R}^{D+1})$  obtenu en rajoutant une entrée toujours égale à 1 est séparable par un hyperplan passant par l'origine.

Toute fonction de  $(\mathbb{R}^D)$  dans  $\{0,1\}$  n'est bien sûr pas calculable par un tel perceptron.

## Algorithme d'apprentissage par correction d'erreur

Étant donné un échantillon d'apprentissage  $\{\mathcal{E}_a\}$  de  $(\mathbb{R}^D)^n$ , c'est-à-dire un ensemble d'exemples dont les descriptions sont  $(D)$  attributs réels (respectivement binaires) et la classe est binaire, il s'agit de trouver un algorithme qui infère à partir de  $\{\mathcal{E}_a\}$  un perceptron qui classifie correctement les éléments de  $\{\mathcal{E}_a\}$  au vu de leurs descriptions si c'est possible, ou au mieux sinon.

L'algorithme d'apprentissage peut être décrit succinctement de la manière suivante. On initialise les poids du perceptron à des valeurs quelconques. A chaque fois que l'on présente un nouvel exemple, on ajuste les poids selon que le perceptron l'a correctement classé ou non. L'algorithme s'arrête lorsque tous les exemples ont été présentés sans modification d'aucun poids ou qu'un nombre maximum d'itération a été atteint.

Dans la suite, on note  $(x_n)$  une entrée. La  $i$ ème composante de  $(x_n)$  est notée  $(x_{n,i})$ . Pour simplifier l'explication de l'algorithme, cette composante sera supposée binaire. Un échantillon  $\{\mathcal{E}_a\}$  est un ensemble de couples  $((x_n, t_n))$  où  $(t_n)$  est la classe binaire de  $(x_n)$ . Si une entrée  $(x_n)$  est présentée en entrée d'un perceptron, on note  $(y_n)$  la sortie binaire calculée par le perceptron. Rappelons qu'il existe une  $((D+1)^n)$  entrée  $(x_0)$  de valeur 1 pour le perceptron. L'apprentissage par correction d'erreur du perceptron est donné dans l'[Algorithm 1](#)

---

### Algorithm 1 (Algorithme d'apprentissage du perceptron par correction d'erreur)

---

1. Initialisation aléatoire des  $(w_i)$
2. Tant que (test)
  1. Prendre un exemple  $((x_n, t_n))$  dans  $\{\mathcal{E}_a\}$
  2. Calculer la sortie  $(y_n)$  du perceptron pour l'entrée  $(x_n)$
  3.  $w_i \leftarrow w_i + (t_n - y_n)x_{n,i}$

La procédure d'apprentissage du perceptron est une procédure de correction d'erreur puisque les poids ne sont pas modifiés lorsque la sortie attendue  $(t_n)$  est égale à la sortie calculée  $(y_n)$  par le perceptron courant.

Étudions les modifications sur les poids lorsque  $(t_n)$  diffère de  $(y_n)$ , lorsque  $(x_n) \in \{0,1\}^D$  :

- si  $(y_n)=0$  et  $(t_n)=1$ , cela signifie que le perceptron n'a pas assez pris en compte les neurones actifs de l'entrée (c'est-à-dire les neurones ayant une entrée à 1). Dans ce cas,  $w_i \leftarrow w_i + x_{n,i}$  : l'algorithme ajoute la valeur de la rétine aux poids synaptiques (renforcement).
- si  $(y_n)=1$  et  $(t_n)=0$ , alors  $w_i \leftarrow w_i - x_{n,i}$  ; l'algorithme retranche la valeur de la rétine aux poids synaptiques (inhibition).

Remarquons que, en phase de calcul, les constantes du perceptron sont les poids synaptiques alors que les variables sont les entrées. Tandis que, en phase d'apprentissage, ce sont les coefficients synaptiques qui sont variables alors que les entrées de l'échantillon  $\{\{cal E\}_a\}$  apparaissent comme des constantes.

Certains éléments importants ont été laissés volontairement imprécis.

- en premier lieu, il faut préciser comment est fait le choix d'un élément de  $\{\{cal E\}_a\}$  : aléatoirement ? En suivant un ordre prédéfini ? Doivent-ils être tous présentés ?
- le critère d'arrêt de la boucle principale de l'algorithme n'est pas défini : après un certain nombre d'étapes ? Lorsque tous les exemples ont été présentés ? Lorsque les poids ne sont plus modifiés pendant un certain nombre d'étapes ?

Nous reviendrons sur toutes ces questions par la suite.

### Example 1 (Apprentissage du OU binaire)

Les descriptions appartiennent à  $\{0,1\}^{(2)}$ , les entrées du perceptron appartiennent à  $\{0,1\}^{(3)}$ , la première composante correspond à l'entrée  $\{x_0\}$  et vaut toujours 1, les deux composantes suivantes correspondent aux variables  $\{x_1\}$  et  $\{x_2\}$ . On suppose qu'à l'initialisation, les poids suivants ont été choisis :  $\{w_0\}=0$  ;  $\{w_1\} = 1$  et  $\{w_2\} = -1$ . On suppose que les exemples sont présentés dans l'ordre lexicographique.

Le tableau suivant présente la trace de l'algorithme à partir de cette initialisation. Aucune entrée ne modifie le perceptron à partir de l'itération 10.

étape	$\{w_0\}$	$\{w_1\}$	$\{w_2\}$	Entrée	$\mathbf{w^top}$	$\{y\}$	$\{t\}$	$\{w_0\}$	$\{w_1\}$	$\{w_2\}$
Init								0	1	-1
1	0	1	-1	100	0	0	0	0	1	-1
2	0	1	-1	101	-1	0	1	1	1	0
3	1	1	0	110	2	1	1	1	1	0
4	1	1	0	111	2	1	1	1	1	0
5	1	1	0	100	1	1	0	0	1	0
6	0	1	0	101	0	0	1	1	1	1
7	1	1	1	110	2	1	1	1	1	1
8	1	1	1	111	3	1	1	1	1	1
9	1	1	1	100	1	1	0	0	1	1
10	0	1	1	101	1	1	1	0	1	1

On peut montrer que si l'échantillon  $\{\{cal E\}_a\}$  est linéairement séparable et si les exemples sont présentés de manière équitable (c'est-à-dire que la procédure de choix des exemples n'en exclut aucun), la procédure d'apprentissage par correction d'erreur converge vers un perceptron linéaire à seuil qui sépare linéairement  $\{\{cal E\}_a\}$ .

```
from IPython.display import Video
Video("videos/correction_erreur_linsep.mp4", embed =True, width=500)
```

0:00

L'inconvénient majeur de cet apprentissage est que si l'échantillon présenté n'est pas linéairement séparable, l'algorithme ne convergera pas et l'on aura aucun moyen de le savoir.

```
from IPython.display import Video
Video("videos/correction_erreur_nonlinsep.mp4", embed =True, width=500)
```

0:00

On pourrait penser qu'il suffit d'observer l'évolution des poids synaptiques pour en déduire si l'on doit arrêter ou non l'algorithme. En effet, si les poids et le seuil prennent deux fois les mêmes valeurs sans que le perceptron ait appris et alors que tous les exemples ont été présentés, cela signifie que l'échantillon n'est pas séparable. Et l'on peut penser que l'on peut borner les poids et le seuil en fonction de la taille de la rétine.

C'est vrai mais les résultats de complexité suivants montrent que cette idée n'est pas applicable en pratique.

1. Toute fonction booléenne linéairement séparable sur  $(D)$  variables peut être réalisée par un perceptron dont les poids synaptiques entiers  $(w_i)$  sont tels que  $(\lfloor w_i \rfloor) \leq (D+1)^{\frac{D+1}{2}}$
2. Il existe des fonction booléennes linéairement séparables sur  $(D)$  variables qui requièrent des poids entiers supérieurs à  $(2^{\frac{D+1}{2}})$

Le premier résultat montre que l'on peut borner les poids synaptiques en fonction de la taille de la rétine, mais par un nombre tellement grand que toute application pratique de ce résultat semble exclue. Le second résultat montre en particulier que l'algorithme d'apprentissage peut nécessiter un nombre exponentiel d'étapes (en fonction de la taille de la rétine) avant de s'arrêter. En effet, les poids ne varient qu'au plus d'une unité à chaque étape. Même lorsque l'algorithme d'apprentissage du perceptron converge, rien ne garantit que la solution sera robuste, c'est-à-dire qu'elle ne sera pas remise en cause par la présentation d'un seul nouvel exemple. Pire encore, cet algorithme n'a aucune tolérance au bruit : si du bruit, c'est-à-dire une information mal classée, vient perturber les données d'entrée, le perceptron ne convergera jamais. En effet, des données linéairement séparables peuvent ne plus l'être à cause du bruit. En particulier, les problèmes non-déterministes, c'est-à-dire pour lesquels une même description peut représenter des éléments de classes différentes, ne peuvent pas être traités à l'aide d'un perceptron.

## Algorithme d'apprentissage par descente de gradient

Plutôt que d'obtenir un perceptron qui classifie correctement tous les exemples, il s'agit maintenant de calculer une erreur et d'essayer de minimiser cette erreur. Pour introduire cette notion d'erreur, on utilise des poids réels et donc des sorties réelles.

Un perceptron linéaire prend en entrée un vecteur  $\mathbf{x}_n$  et calcule une sortie  $y_n$ . Un perceptron est défini par la donnée d'un vecteur  $w$  de coefficients synaptiques. La sortie  $y_n$  est définie par  $y_n = w^T \mathbf{x}_n$ . L'erreur du perceptron sur un échantillon d'apprentissage  $\{\{E_a\}$  d'exemples  $(\mathbf{x}_n, t_n)$  est définie en utilisant par l'erreur quadratique

$$E(w) = \frac{1}{2} \sum_{(\mathbf{x}_n, t_n) \in \{E_a\}} (t_n - y_n)^2$$

L'erreur mesure donc l'écart entre les sorties attendue et calculée sur l'échantillon complet. On remarque que  $E(w) = 0$  si et seulement si le perceptron classifie correctement l'échantillon complet. On suppose  $\{E_a\}$  fixé, le problème est donc de déterminer, par descente de gradient, un vecteur  $\tilde{w}$  qui minimise  $E(w)$ . On a alors :

$$\begin{aligned} & \left[ \begin{aligned} & \frac{\partial E(w)}{\partial w_i} = \frac{1}{2} \sum_{(\mathbf{x}_n, t_n) \in \{E_a\}} (t_n - y_n)^2 \end{aligned} \right] \\ & \frac{\partial E(w)}{\partial w_i} = \sum_{(\mathbf{x}_n, t_n) \in \{E_a\}} (t_n - y_n)(x_n - w_i) \end{aligned}$$

L'application de la méthode du gradient invite donc à modifier le poids  $w_i$  après une présentation complète de  $\{E_a\}$  d'une quantité  $\Delta w_i$  définie par :  $\Delta w_i = -\epsilon \frac{\partial E(w)}{\partial w_i}$

L'algorithme d'apprentissage par descente de gradient du perceptron linéaire peut maintenant être défini l'[Algorithm 2](#).

---

#### Algorithm 2 (Algorithme d'apprentissage du perceptron par descente de gradient)

---

1. Initialisation aléatoire des  $w_i$
2. Tant que (test)
  1. Pour tout  $i$   $\Delta w_i \leftarrow 0$
  2. Pour tout  $(\mathbf{x}_n, t_n) \in \{E_a\}$ 
    1. Calculer  $y_n$
    2. Pour tout  $i$   $\Delta w_i \leftarrow \Delta w_i + \epsilon (t_n - y_n)x_n$
  3. Pour tout  $i$   $w_i \leftarrow w_i + \Delta w_i$

La fonction erreur quadratique ne possède qu'un minimum (la surface est une paraboloïde). La convergence est assurée, même si l'échantillon d'entrée n'est pas linéairement séparable, vers un minimum de la fonction erreur pour un  $\epsilon$  bien choisi, suffisamment petit.  $\epsilon$  est appelé le taux d'apprentissage (ou *learning rate*).

Si  $\epsilon$  est trop grand, on risque d'osciller autour du minimum. Pour cette raison, une modification classique est de diminuer graduellement la valeur de  $\epsilon$  en fonction du nombre d'itérations. Le principal défaut est que la convergence peut être très lente et que chaque étape nécessite le calcul sur tout l'ensemble d'apprentissage.

Au lieu de calculer les variations des poids en sommant sur tous les exemples de  $\{E_a\}$ , l'idée est alors de modifier les poids à chaque présentation d'exemple. La règle de modification des poids devient :  $\Delta w_i = \epsilon (t_n - y_n)x_n$

Cette règle est appelée règle delta, ou règle Adaline, ou encore règle de Widrow-Hoff, et l'[Algorithm 3](#) décrit cette règle :

---

**Algorithm 3 (Algorithme d'apprentissage du perceptron par descente de gradient)**

---

1. Initialisation aléatoire des  $\langle w_i \rangle$
2. Tant que (test)
  1. Prendre un exemple  $\langle (\mathbf{x}_n, t_n) \rangle \in \{\mathcal{E}_a\}$
  2. Calculer  $\langle y_n \rangle$
  3. Pour tout  $\langle i \rangle$   $\langle w_i \leftarrow w_i + \varepsilon (t_n - y_n) x_n^i \rangle$

En général, on parcourt l'échantillon dans un ordre prédefini. Le critère d'arrêt généralement choisi fait intervenir un seuil de modifications des poids pour un passage complet de l'échantillon.

Au coefficient  $\langle \varepsilon \rangle$  près dans la règle de modification des poids, on retrouve l'algorithme d'apprentissage par correction d'erreur. Pour l'algorithme de Widrow-Hoff, il y a correction chaque fois que la sortie totale (qui est un réel) est différente de la valeur attendue. Ce n'est donc pas une méthode d'apprentissage par correction d'erreur puisqu'il y a modification du perceptron dans (presque) tous les cas. Rappelons également que l'algorithme par correction d'erreur produit en sortie un perceptron linéaire à seuil alors que l'algorithme par descente de gradient produit un perceptron linéaire. L'avantage de l'algorithme de Widrow-Hoff par rapport à l'algorithme par correction d'erreur est que, même si l'échantillon d'entrée n'est pas linéairement séparable, l'algorithme va converger vers une solution optimale (sous réserve du bon choix du paramètre  $\langle \varepsilon \rangle$ ). L'algorithme est, par conséquent, plus robuste au bruit.

L'algorithme de Widrow-Hoff s'écarte de l'algorithme du gradient sur un point important : on modifie les poids après présentation de chaque exemple en fonction de l'erreur locale et non de l'erreur globale. On utilise donc une méthode de type *gradient stochastique*. Rien ne prouve alors que la diminution de l'erreur en un point ne va pas être compensée par une augmentation de l'erreur pour les autres points. La justification empirique de cette manière de procéder est commune à toutes les méthodes adaptatives : le champ d'application des méthodes adaptatives est justement l'ensemble des problèmes pour lesquels des ajustements locaux vont finir par converger vers une solution globale.

L'algorithme de Widrow-Hoff est souvent utilisé en pratique et donne de bons résultats. Il sera utilisé dans les autres réseaux de neurones rencontrés dans ce cours, avec sa variante où la modification des poids se fait après présentation d'un sous ensemble de données d'apprentissage (apprentissage par batchs). La convergence est, en général, plus rapide que par la méthode du gradient. Il est fréquent pour cet algorithme de faire diminuer la valeur de  $\langle \varepsilon \rangle$  en fonction du nombre d'itérations comme pour l'algorithme du gradient.

## Implémentation

On illustre le pouvoir de séparation linéaire d'un perceptron sur trois jeux de données :

- un jeu de données linéairement séparable
- deux jeux de données non linéairement séparables classiques (« twocircles » et « moons »)

```

import numpy as np
from matplotlib import pyplot as plt
from matplotlib import colors
import torch
import torch.nn as nn
import torch.optim as optim

fichiers_train =
['./data/linear_data_train.csv','./data/twocircles_data_train.csv','./data/moon_da
ta_train.csv']
fichiers_test =
['./data/linear_data_eval.csv','./data/twocircles_data_eval.csv','./data/moon_data
_eval.csv']

# Fonction de lecture des jeux de données
def extract_data(filename):

    labels = []
    features = []

    for line in open(filename):
        row = line.split(",")
        labels.append(int(row[0]))
        features.append([float(x) for x in row[1:]])

    features_np = np.matrix(features).astype(np.float32)

    labels_np = np.array(labels).astype(dtype=np.uint8)
    labels_onehot = (np.arange(num_labels) == labels_np[:, None]).astype(np.float32)

    return features_np, labels_onehot

```

On écrit une fonction permettant de visualiser le résultat de la classification par le perceptron.

```

def plotResults(ax,ay,X,Y,model,title=pltloss,name):
    mins = np.amin(X[:,0]);
    mins = mins - 0.1*np.abs(mins);
    maxs = np.amax(X[:,0]);
    maxs = maxs + 0.1*maxs;

    xs,ys =
    np.meshgrid(np.linspace(mins[0,0],maxs[0,0],300),np.linspace(mins[0,1], maxs[0,1],
    300));

    toto = torch.FloatTensor(np.c_[xs.flatten(), ys.flatten()])
    Z = np.argmax(model(toto).detach().numpy(), axis=-1)
    Z=Z.reshape(xs.shape[0],xs.shape[1])

    labelY = np.matrix(Y[:, 0]+2*Y[:, 1])
    labelY = labelY.reshape(np.array(X[:, 0]).shape)

    ax.contourf(xs, ys, Z, cmap=colors.ListedColormap([[0,0.5,0.66,0],
    [0.93,0.76,0.27,0]]),alpha=.5)
    ax.scatter(np.array(X[:, 0]),np.array(X[:, 1]),c=
    np.array(labelY),s=20,cmap=colors.ListedColormap(['red', 'green']))
    ax.set_title(title)

    ay.plot(pltloss)
    ay.set_title("perte sur " + name)
    ay.set_xlabel("epoch")

    plt.tight_layout()

```

On construit ensuite le perceptron (extension de la classe `nn.Module`)

```

# Paramètres de l'apprentissage
batch_size = 100
num_epochs = 10000
num_labels = 2
num_features = 2

class Perceptron(nn.Module):
    def __init__(self,p):
        super(Perceptron, self).__init__()
        # Couche d'entrée
        self.fc = nn.Linear(num_features,num_labels)
        self.output = nn.Softmax(1)
    def forward(self, x):
        lin = self.fc(x)
        output = self.output(lin)
        return output

```

On écrit ensuite la fonction d'entraînement. La fonction de perte est l'["entropie croisée binaire"](#) et l'optimiseur est [Adam](#).

```
loss = nn.BCELoss()

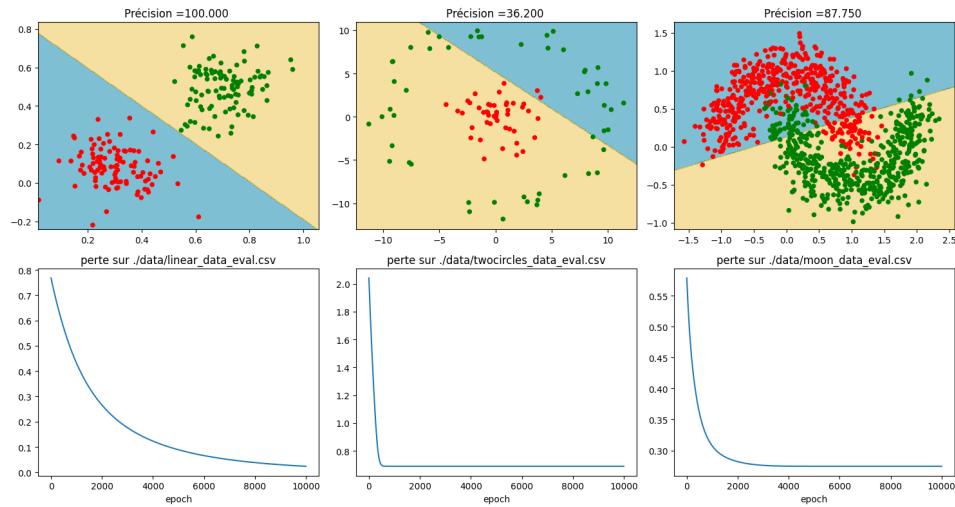
def train_session(X,y,classifier,criterion,optimizer,n_epochs=num_epochs):
    loss_values = []
    losses = np.zeros(n_epochs)
    correct = 0
    for iter in range(n_epochs):
        optimizer.zero_grad()
        yPred = classifier(X)
        loss = criterion(yPred,y)
        loss_values.append(loss.detach().numpy())
        #Gradient et rétropropagation
        loss.backward()
        #Mise à jour des poids
        optimizer.step()
        y2 = yPred>0.5
        correct = (y2 == y).sum().item()/2
    acc = 100 * correct / (X.shape[0])
    return loss_values,acc
```

On applique enfin le perceptron sur les jeux de données et on visualise les résultats.

```
fig,axs = plt.subplots(2, 3,figsize=(15,8))
for i,name_train,name_test in zip ([0,1,2],fichiers_train,fichiers_test):
    train_data,train_labels = extract_data(name_train)
    test_data, test_labels = extract_data(name_test)

    model = Perceptron(train_data.shape[1])
    optimizer = optim.Adam(model.parameters())
    pltloss,acc =
train_session(torch.FloatTensor(train_data),torch.FloatTensor(train_labels),model,
loss,optimizer)

    titre= "Précision ={0:5.3f} ".format(acc)
    plotResults(axs[0][i],axs[1][i],test_data, test_labels, model, titre, pltloss,
name_test)
```



## Pour en finir avec le perceptron

L'apprentissage par correction ou par la méthode du gradient ne sont rien d'autre que des techniques de séparation linéaire qu'il faudrait comparer aux techniques utilisées habituellement en statistiques (discriminant linéaire, machines à vecteurs de support,...). Ces méthodes sont non paramétriques, c'est-à-dire qu'elles n'exigent aucune autre hypothèse sur les données que la séparabilité.

On peut montrer que presque tous les échantillons de moins de  $|D|$  exemples sont linéairement séparables lorsque  $|D|$  est le nombre de variables. Une classification correcte d'un petit échantillon n'a donc aucune valeur prédictive. Par contre, lorsque l'on travaille sur suffisamment de données et que le problème s'y prête, on constate empiriquement que le perceptron appris par un des algorithmes précédents a un bon pouvoir prédictif.

Il est bien évident que la plupart des problèmes d'apprentissage qui se posent naturellement ne peuvent pas être résolus par des méthodes aussi simples : il n'y a que très peu d'espoir que les exemples naturels se répartissent sagement de part et d'autre d'un hyperplan. Deux manières de résoudre cette difficulté peuvent être envisagées :

- soit mettre au point des séparateurs non-linéaires,
- soit (ce qui revient à peu près au même) complexifier l'espace de représentation de manière à linéariser le problème initial.

Les réseaux multicouches abordent ce type de problème.

## Perceptrons multicouches

### Definition 4 (Perceptron multicouches)

Un perceptron à  $(L+1)$  couches (Fig. 2) est un réseau constitué d'une rétine à  $D$  neurones (auxquels on rajoute l'entrée  $x_0$ ),  $C$  neurones de sortie, et des neurones dits **cachés**, organisés dans  $L$  couches cachées intermédiaires. De fait, un tel réseau comporte  $(L+2)$  couches mais on compte rarement la rétine, puisque cette dernière n'effectue pas de calculs. Le  $i^{\text{e}}$  neurone dans la couche cachée  $(l)$  calcule la sortie

$$\begin{aligned} y_{i,l} &= f\left(\sum_{k=0}^{m-1} w_{i,k} y_{k,l-1}\right) \quad \text{avec } z_{i,l} = \sum_{k=0}^{m-1} w_{i,k} y_{k,l-1} + b_i \\ \text{où } w_{i,k} &\text{ est le poids de la connexion entre le } k^{\text{e}} \text{ neurone de la couche } (l-1) \text{ et le } i^{\text{e}} \text{ neurone de la couche } (l), \text{ et } b_i \text{ est le biais. De plus, } \\ m &\text{ est le nombre de neurones de la couche } (l), \text{ de sorte que } D = m^0 \text{ et } C = m^{L+1}. \text{ Enfin, } f \text{ est la fonction d'activation du neurone (supposée identique pour tous les neurones).} \end{aligned}$$

En introduisant dans chaque couche un neurone supplémentaire ( $y_0 = 1$ ) pour gérer le biais, on a :

$$\begin{aligned} \mathbf{z}_i &= \sum_{k=0}^{m-1} w_{i,k} y_{k,l-1} \\ \mathbf{y}_i &= f(\mathbf{z}_i) \end{aligned}$$

avec  $\mathbf{z}$ ,  $\mathbf{w}$  et  $\mathbf{y}$  les représentations vectorielles et matricielles des entrées ( $x_i$ ), des poids ( $w_{i,k}$ ) et des sorties ( $y_i$ ).

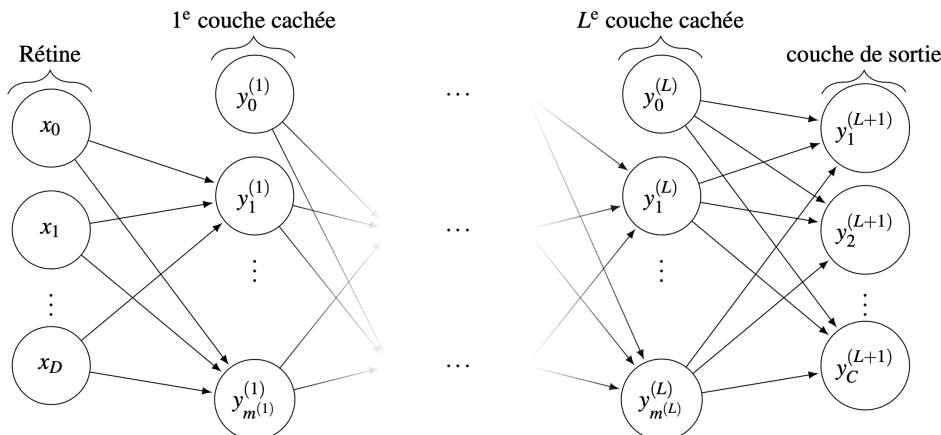


Fig. 2 Perceptron multicouches à  $(L+1)$  couches,  $D$  entrées et  $C$  sorties.

Un tel réseau représente une fonction

$$y = \mathbb{R}^D \rightarrow \mathbb{R}^C$$

$\|\mathbf{R}^C\| \{\mathbf{x}\} \& \mapsto \{\mathbf{y}(x, w)\}$  lorsque  $\|\mathbf{y}_i(x, w)\|$  est tel que  $\|\{\mathbf{y}_i(x, w)\}\| = \|\mathbf{y}_i\|^{(L+1)}$  et  $\{\mathbf{w}\}$  est la matrice de tous les poids du réseau.

On parlera de **réseau profond (Deep network)** lorsque le nombre de couches cachées est « suffisamment important » (supérieur à 3 par exemple).

## Fonctions d'activation

Trois grandes classes de fonction d'activation  $f$  sont généralement utilisées : les fonctions de seuils (comme dans le perceptron linéaire à seuil), les fonctions linéaires par morceau et les fonctions de type sigmoïde. Dans les deux premiers cas, de nombreux problèmes se présentent, notamment en raison de la non différentiabilité de ces fonctions (qui est nécessaire dans les algorithmes d'apprentissage du type descente de gradient), ou encore en raison de la faiblesse de leur pouvoir d'expression. Ainsi, il est préférable d'utiliser des fonctions de type sigmoïde, et par exemple la sigmoïde logistique est donnée par :

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

La tangente hyperbolique  $\tanh(z)$ , également utilisée pour ses bonnes propriétés de dérivabilité ( $(\tanh)' = 1 - \tanh^2(z)$ ), peut être vue comme une transformation linéaire de la sigmoïde dans l'intervalle  $[-1, 1]$ .

Ces réseaux peuvent être utilisés en régression (sortie à valeurs dans  $\mathbb{R}^C$ ) ou en classification. Dans ce dernier cas, la fonction d'activation softmax est utilisée à la sortie du réseau pour interpréter les sorties comme des valeurs de probabilité a posteriori. S'il s'agit de classer un exemple  $x$  à la classe  $c$ , la probabilité conditionnelle  $p(c|x)$  peut être calculée en utilisant la règle de Bayes :

$$p(c|x) = \frac{p(x|c)p(c)}{p(x)}$$

$p(c|x)$  est alors interprétée comme une probabilité a posteriori. Disposant de ces probabilités pour tout  $c=1, \dots, C$ , la règle de décision de Bayes donne :

$$c: \mathbb{R}^D \rightarrow \{1, \dots, C\}, x \mapsto \operatorname{argmax}_c \left( p(c|x) \right)$$

L'utilisation de la fonction d'activation softmax en sortie permet d'interpréter les sorties du réseau comme de telles probabilités : la sortie du  $i^{\text{th}}$  neurone de la couche de sortie est

$$\sigma(z^{(L+1)}, i) = \frac{\exp(z_i^{(L+1)})}{\sum_{k=1}^C \exp(z_k^{(L+1)})}$$

En apprentissage profond, il a été reporté que la sigmoïde et la tangente hyperbolique avaient des performances moindres que la fonction d'activation softsign :

$$s(z) = \frac{1}{1 + |z|}$$

En effet, les valeurs de  $|z|$  arrivant près des paliers de saturation de ces fonctions donnent des gradients faibles, qui ont tendance à s'annuler lors de la phase d'apprentissage détaillée plus loin (rétropropagation du gradient). Une autre fonction, non saturante elle, peut être utilisée :

$$r(z) = \max(0, z)$$

Les neurones cachés utilisant la fonction décrite dans l'équation précédente sont appelés neurones linéaires rectifiés (**Rectified Linear Units, ReLUs**), et sont en pratique très utilisés.

Quelques fonctions d'activation sont présentées dans la [Fig. 3](#).

Nom	Graphe	$f$	$f'$
Rampe		$f(x) = x$	$f'(x) = 1$
Heaviside		$f(x) = \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{si } x \neq 0 \\ ? & \text{si } x = 0 \end{cases}$
Logistique ou sigmoïde		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
Tangente hyperbolique		$f(x) = \tanh(x) \\ = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f^2(x)$
Arc Tangente		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
ReLU		$f(x) = \begin{cases} 0 & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{si } x \neq 0 \\ 1 & \text{si } x = 0 \end{cases}$
Exponentielle Linéaire		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases}$

**Fig. 3** Quelques fonctions d'activation

Les fonctions d'activation sous Pytorch sont résumées [ici](#).

## Entraînement des réseaux multicouches

Pour pouvoir utiliser les réseaux multicouches en apprentissage, deux ingrédients sont indispensables :

- une méthode indiquant comment choisir une architecture de réseau pour résoudre un problème donné. C'est-à-dire, pouvoir répondre aux questions suivantes : combien de couches cachées ? combien de neurones par couche cachée ?
- une fois l'architecture choisie, un algorithme d'apprentissage qui calcule, à partir d'un échantillon d'apprentissage  $\{\mathcal{E}_n\}_{n=1}^N$ , les valeurs des poids synaptiques pour construire un réseau adapté au problème (c'est à dire approchant une fonction  $g$  désirée mais inconnue, telle qu'en particulier  $\|\mathbf{t}_n - \mathbf{g}(\mathbf{x}_n)\|$ ).

Sur le premier point, quelques algorithmes d'apprentissage auto-constructifs ont été proposés. Leur rôle est double :

- apprentissage de l'échantillon avec un réseau courant,
- modification du réseau courant, en ajoutant de nouvelles cellules ou une nouvelle couche, en cas d'échec de l'apprentissage.

Il semble assez facile de concevoir des algorithmes auto-constructifs qui classent correctement l'échantillon, mais beaucoup plus difficile d'en obtenir qui aient un bon pouvoir de généralisation. Il a fallu attendre le milieu des années 1980 pour que le deuxième problème trouve une solution : l'algorithme de **rétropropagation du gradient**.

L'entraînement, comme dans le cas de l'algorithme de descente de gradient, consiste à trouver les poids qui minimisent une fonction d'erreur, mesurant l'écart entre la sortie du réseau  $y(\mathbf{w}, \mathbf{x}_n)$  et  $t_n$ , pour tous les exemples de  $\{\mathcal{E}_a\}$ . Les fonctions couramment choisies sont les sommes des fonctions de perte sur chaque exemple, et incluent l'erreur quadratique

$$\begin{aligned} E(\mathbf{w}) &= \sum_{n=1}^N E_n(\mathbf{w}) = \sum_{n=1}^N \sum_{i=1}^C (y_i(\mathbf{w}, \mathbf{x}_n) - t_i)^2 \\ \text{ou l'erreur d'entropie croisée} \\ E(\mathbf{w}) &= \sum_{n=1}^N E_n(\mathbf{w}) = \sum_{n=1}^N \sum_{i=1}^C t_i \log(y_i(\mathbf{w}, \mathbf{x}_n)) \end{aligned}$$

où  $t^i$  est la composante  $i$  de  $t_n$ .

## Stratégies d'entraînement

Parmi les stratégies d'entraînement qui peuvent être retenues, trois sont classiquement utilisées

- entraînement sur  $\{\mathcal{E}_a\}$ , les poids étant mis à jour après présentation, en fonction de l'erreur totale  $E(\mathbf{w}) = \sum_{n=1}^N E_n(\mathbf{w})$ .
- entraînement stochastique : un exemple est présenté et les poids sont mis à jour sur l'erreur  $E_n(\mathbf{w})$  calculée sur cet exemple (règle Adaline)
- entraînement par batch sur un sous-ensemble  $\{M \subset \{1, \dots, N\}\}$  de  $\{\mathcal{E}_a\}$ , les poids étant mis à jour en fonction de l'erreur cumulée  $E_M(\mathbf{w}) = \sum_{n \in M} E_n(\mathbf{w})$ .

## Optimisation des paramètres

Considérons le cas de l'entraînement stochastique. La condition nécessaire d'optimalité d'ordre 1 donne

$$\frac{\partial E_n}{\partial \mathbf{w}} = \nabla E_n(\mathbf{w}) = 0$$

Une méthode itérative est utilisée pour trouver une solution approchée. Si  $\mathbf{w}[t]$  est le vecteur de poids à la  $t^{\text{ème}}$  itération, une mise à jour des poids  $\Delta \mathbf{w}[t]$  est calculée et propagée à l'itération suivante :  $\mathbf{w}[t+1] = \mathbf{w}[t] + \Delta \mathbf{w}[t]$ .

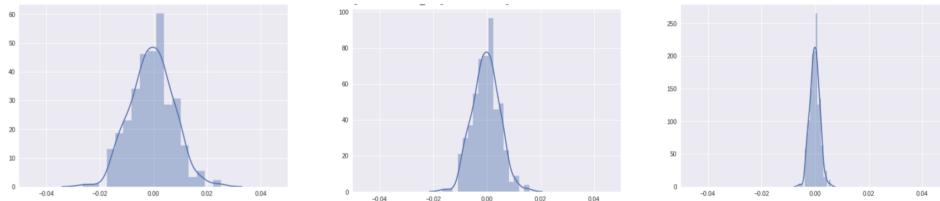
Comme dans le cas du perceptron, on peut utiliser une méthode de type descente de gradient (ordre 1), ou une méthode type Newton (ordre 2, qui nécessite alors le calcul ou l'estimation du Hessien  $H_n$ ) de  $E_n$  à chaque itération).

- pour la méthode de descente du gradient, la mise à jour est effectuée par  $\mathbf{w}[t+1] = \mathbf{w}[t] - \gamma \nabla E_n(\mathbf{w}[t])$  où  $\gamma$  est le taux d'apprentissage.
- pour les méthodes d'ordre 2 type Newton, la mise à jour s'effectue selon le schéma  $\mathbf{w}[t+1] = \mathbf{w}[t] - \gamma \left( \frac{\partial^2 E_n}{\partial \mathbf{w}^2} \right)^{-1} \nabla E_n(\mathbf{w}[t])$  où  $\gamma$  est le taux d'apprentissage. L'ordre 2 assure une convergence plus rapide, mais requiert le calcul et l'inversion du Hessien  $\frac{\partial^2 E_n}{\partial \mathbf{w}^2}$  de  $E_n$ , ce qui est coûteux.

## Initialisation des poids

Une méthode itérative d'optimisation étant utilisée, l'initialisation des poids requiert une attention toute particulière.

Une première idée est d'initialiser les poids selon une loi normale :  $\forall i; w_{ij} \sim \mathcal{N}(0,1)$ . Cependant, cela amène naturellement rapidement à une évolution des poids vers des valeurs nulles (Fig. 4) (phénomène de disparition du gradient).



**Fig. 4** Evolution de la distribution des poids au cours des epochs d'apprentissage

(10 epochs à gauche, 20 au centre et 50 epochs à droite).

Plus le réseau est profond, plus l'apprentissage sera sensible à ce phénomène. On parle de *disparition du gradient* (vanishing gradient).

En faisant l'hypothèse que les entrées de chaque cellule de la rétine sont distribuées selon une loi gaussienne, il est alors courant d'introduire une dépendance à la profondeur des couches considérées : on choisit les poids aléatoirement dans

$$\frac{\sqrt{m^{(l-1)}}}{\sqrt{m^l}} < w_{ij}^{(l)} < \frac{\sqrt{m^{(l-1)}}}{\sqrt{m^l}}$$

En utilisant des fonctions d'activation sigmoïde, il a été prouvé que l'apprentissage était alors optimal, en le sens que l'apprentissage est rapide et que les poids atteignent une valeur stable quasiment tous en même temps.

Un autre schéma d'initialisation est possible (initialisation normalisée, ou initialisation de Xavier) en choisissant

$$\frac{\sqrt{6}}{\sqrt{m^{(l-1)} + m^l}} < w_{ij}^{(l)} < \frac{\sqrt{6}}{\sqrt{m^{(l-1)} + m^l}}$$

Donnons quelques éléments qui amènent à ce schéma. Supposons un perceptron multicouches avec  $w \in \mathbb{R}^d$ ,  $w_i \sim \mathcal{N}(0,1)$  et d'entrées  $x \in \mathbb{R}^d$ .

Le potentiel post-synaptique d'un neurone de la première couche cachée est de la forme  $w^\top x$ .

Alors

$$\text{Var}(w^\top x) = \text{Var}(\sum_i w_i x_i) = \sum_i \text{Var}(w_i x_i) = \sum_i \text{Var}(w_i) \text{Var}(x_i)$$

Puisque  $\text{Var}(w_i) = 1$

$$\text{Var}(w^\top x) = \sum_i \text{Var}(x_i)$$

Or  $w_i \sim \mathcal{N}(0,1)$  donc  $\text{Var}(w_i) = 1$

Et plus généralement

$$\text{Var}(y) = \left( \text{Var}(w) \text{Var}(x) \right)^{1/2}$$

Chaque neurone peut varier d'un facteur  $d$  par rapport à son entrée.

Si  $d > 1$  le gradient croît à mesure que l'on s'enfonce dans le réseau. A l'inverse, si  $d < 1$  le gradient disparaît lorsque  $d$  croît.

Il est donc légitime pour imiter ces deux phénomènes d'imposer  $d = 1$ , et donc  $\text{Var}(w) = 1/d$

$$w_{ij} \sim \mathcal{N}\left(0, \frac{1}{d}\right)$$

Si la fonction d'activation du neurone est la fonction ReLU, on peut multiplier par  $\frac{1}{\sqrt{d}}$  pour prendre en compte la partie négative qui ne participe pas au calcul de la variance.

## Rétropropagation de l'erreur

L'[Algorithm 4](#), dit algorithme de rétropropagation du gradient, est utilisé pour évaluer le gradient  $\nabla E_n(\mathbf{w}[t])$  de l'erreur  $(E_n)$  à chaque itération, ceci pour tous les poids

---

### Algorithm 4 (Algorithme de rétropropagation du gradient)

---

1. Propager un exemple  $(\mathbf{x}_n)$  dans le réseau.
2. 
$$\frac{\partial E_n}{\partial w_{j,i}} = \frac{\partial E_n}{\partial z_i^{(L+1)}} f'(z_i^{(L+1)})$$
3. 
$$\delta_i^{(l)} = \sum_{k=1}^{m^{(l+1)}} w_{i,k}^{(l+1)} \delta_k^{(l+1)}$$
4. Calculer les composantes du gradient :

$$\begin{aligned} \frac{\partial E_n}{\partial w_{j,i}} &= \delta_j^{(l)} \\ &\quad \cdot \delta_i^{(l-1)} \end{aligned}$$

Dans le cas d'un apprentissage stochastique, cet algorithme est appliqué jusqu'à convergence, pour estimer les poids du réseau de neurones.

## Critères d'arrêt

Plusieurs critères d'arrêt peuvent être utilisés avec l'algorithme de rétropropagation du gradient. Le plus commun consiste à fixer un nombre maximum de périodes d'entraînement (sur  $(|\mathcal{E}_a|)$ ), ce qui fixe effectivement une limite supérieure sur la durée de l'apprentissage. Ce critère est important car la rétropropagation n'offre aucune garantie quant à la convergence de l'algorithme. Il peut arriver, par exemple, que le processus d'optimisation reste pris dans un minimum local. Sans un tel critère, l'algorithme pourrait ne jamais se terminer. Un deuxième critère commun consiste à fixer une borne inférieure sur l'erreur quadratique moyenne. Dépendant de l'application, il est parfois possible de fixer *a priori* un objectif à atteindre. Lorsque l'indice de performance choisi diminue en dessous de cet objectif, on considère simplement que le réseau a suffisamment bien appris ses données et on arrête l'apprentissage.

Les deux critères précédents sont utiles mais ils comportent aussi des limitations. Le critère relatif au nombre maximum de périodes d'entraînement n'est aucunement lié à la performance du réseau. Le critère relatif à l'erreur minimale obtenue mesure quant à lui un indice de performance mais ce dernier peut engendrer du sur-apprentissage qui n'est pas désirable dans la pratique, surtout si l'on ne possède pas une grande quantité de données d'apprentissage, ou si ces dernières ne sont pas de bonne qualité.

Un processus d'apprentissage comme celui de la rétropropagation, vise à réduire autant que possible l'erreur que commet le réseau. Mais cette erreur est mesurée sur un ensemble de données d'apprentissage  $(|\mathcal{E}_a|)$ . Si les données sont bonnes, c'est-à-dire qu'elles représentent bien le processus physique sous-jacent que l'on tente d'apprendre ou de modéliser, et que l'algorithme a convergé sur un optimum global, alors il devrait bien se comporter sur d'autres données issues du même processus physique. Cependant, si les données d'apprentissage sont partiellement corrompues par du bruit ou par des erreurs de mesure, alors il n'est pas évident que la performance optimale du réseau soit atteinte en minimisant l'erreur, lorsqu'on la testera sur un jeu de données différent de celui qui a servi à l'entraînement. On parle alors de la **capacité de généralisation** du réseau, c'est-à-dire sa capacité à bien se comporter avec des données qu'il n'a jamais vu auparavant.

Une solution à ce problème consiste à faire appel à un autre critère d'arrêt basé sur une technique de validation croisée. Cette technique consiste à utiliser deux ensembles indépendants de données. En pratique, il s'agit de partitionner  $(|\mathcal{E}_a|)$  pour entraîner le réseau en un ensemble d'apprentissage (ajustement des poids) un ensemble de validation (calcul d'un indice de performance). Le critère d'arrêt consiste alors à stopper l'apprentissage lorsque l'indice de performance calculé sur les données de validation cesse de s'améliorer pendant plusieurs périodes d'entraînement. Lors de deux périodes successives d'entraînement, des exemples peuvent être échangés entre ensembles d'apprentissage et de validation.

## Propriété fondamentale

Terminons par une dernière remarque sur la puissance de représentation des réseaux multicouches. La plupart des fonctions numériques peuvent être approximées avec une précision arbitraire par des réseaux à une seule couche cachée. Mais cette couche cachée peut être démesurément grande et le théorème de Hornik, qui affirme cette propriété d'approximateurs universels des réseaux multicouches, est essentiellement un résultat théorique sur l'expressivité des réseaux.

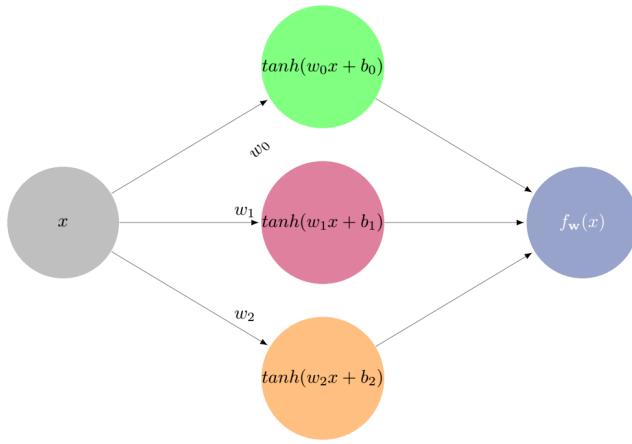
Plus formellement, la propriété fondamentale des réseaux de neurones est l'**approximation parcimonieuse**, qui traduit deux propriétés distinctes : d'une part les réseaux de neurones sont des **approximateurs universels**, et d'autre part, une approximation à l'aide d'un réseau de neurones nécessite, en général, moins de paramètres ajustables que les approximateurs usuels.

- Approximateurs universels : Cybenko a énoncé en 1989 la propriété suivante : toute fonction bornée, suffisamment régulière, peut être approchée uniformément, avec une précision arbitraire, dans un domaine fini de l'espace de ses variables, par un réseau de neurones comportant une couche de neurones cachés en nombre fini, possédant tous la même fonction d'activation, et un neurone de sortie linéaire.
- Parcimonie : Hornik a montré en 1994 que si la sortie d'un réseau de neurones est une fonction non linéaire des paramètres ajustables, elle est plus parcimonieuse que si elle était une fonction linéaire de ces paramètres. De plus, pour les réseaux dont la fonction d'activation des neurones est une sigmoïde, l'erreur commise dans l'approximation varie comme l'inverse du nombre de neurones cachés, et elle est indépendante du nombre de variables de la fonction à approcher. Ainsi, pour une précision donnée (*i.e.* étant donné un nombre de neurones cachés) le nombre de paramètres du réseau est proportionnel au nombre de variables de la fonction à approcher.

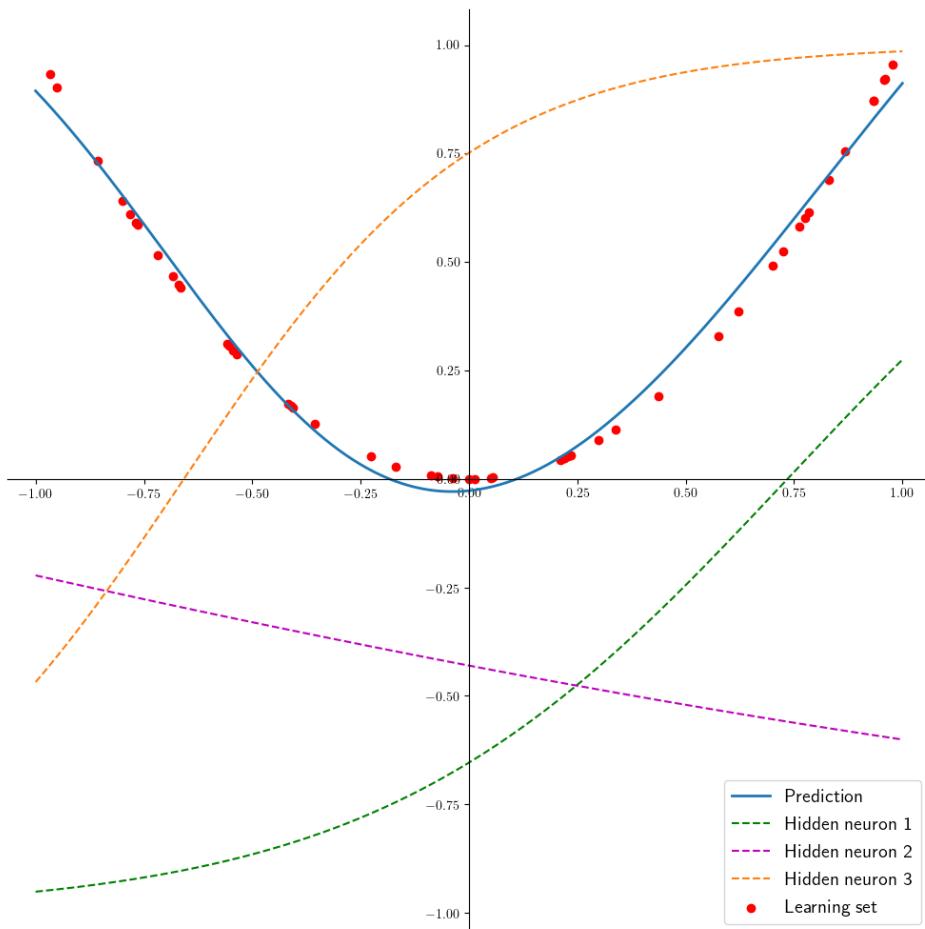
Dans la plupart des cas d'utilisation des réseaux de neurones, il va s'agir d'établir un modèle d'une fonction inconnue à partir de mesures bruitées de l'ensemble d'apprentissage, permettant de reproduire les sorties à partir des entrées, et de proposer une généralisation à des données test. On cherche alors la **fonction de régression** du processus considéré, *i.e.* la fonction obtenue en calculant la moyenne d'une infinité de mesures effectuées en chaque point du domaine de validité du modèle. Le nombre de points de ce domaine étant lui-même infini, la connaissance de la fonction de régression nécessiterait donc une infinité de mesures en un nombre infini de points.

Les réseaux de neurones, en raison de leur propriété fondamentale, sont de bons candidats pour réaliser une approximation de la fonction de régression à partir d'un nombre fini de mesures. Ils entrent donc dans le cadre des méthodes statistiques d'apprentissage, et élargissent ce domaine déjà bien exploré pour des fonctions de régression linéaire au cas non linéaire.

Pour illustration simple, on considère le problème d'apprentissage de la fonction  $|f(x)=x^2|$  en utilisant un réseau à une couche cachée de trois neurones, équipés de la fonction d'activation  $\tanh()$  ([Fig. 5](#)). La ([Fig. 6](#)) présente le résultat de l'apprentissage : en rouge sont représentés 50 points d'apprentissage, en bleu la fonction prédite sur l'intervalle  $[-1,1]$ . Les fonctions apprises par les trois neurones sont en pointillés, et la combinaison linéaire de ces fonctions donne la courbe prédite.



**Fig. 5** Un perceptron multicouches à une couche cachée



**Fig. 6** Approximation de  $|f(x)=x^2|$  par un réseau à une couche cachée.

## Régularisation

La notion d'approximateur universel peut induire également un problème de surapprentissage (overfitting) de  $\|\{\mathcal{E}\}_a\|$ . Les techniques de régularisation permettent d'éviter ce problème, et permettent aux réseaux de neurones (et à d'autres algorithmes d'ailleurs, comme les autoencodeurs ou les SVM par exemple) d'avoir une bonne capacité de généralisation.

Il existe plusieurs techniques permettant d'introduire de la régularisation dans les réseaux. Parmi elles, on note :

- $\{\text{cal E}\}_a$  est enrichi pour introduire certaines invariances que le réseau est supposé apprendre.
- à chaque exemple/batch présenté, chaque neurone caché est supprimé du calcul de la sortie avec probabilité  $\{p\}$  (dropout). Cette technique peut être vue comme la construction d'un modèle moyen d'apprentissage de plusieurs réseaux distincts.
- lorsque  $\{\text{cal E}\}_a$  est séparé en un ensemble d'apprentissage  $\{\text{cal E}\}^1_a$  et un ensemble de validation  $\{\text{cal E}\}^2_a$ , il est courant de voir que l'erreur baisse sur  $\{\text{cal E}\}^1_a$  au fil des itérations, alors que l'erreur sur  $\{\text{cal E}\}^2_a$  tend à augmenter lorsque le réseau commence à sur-apprendre sur  $\{\text{cal E}\}^1_a$ . L'entraînement est alors stoppé dès que l'erreur sur  $\{\text{cal E}\}^2_a$  atteint un minimum. Cette technique est appelée early stopping (arrêt précoce).
- le partage de poids : plusieurs neurones d'une même couche partagent des mêmes valeurs de poids. La complexité du réseau est réduite et des informations *a priori* peuvent être introduites par ce biais dans l'architecture du réseau. L'algorithme de rétropagation du gradient s'en trouve modifié et l'étape 4 devient
$$\begin{aligned} \frac{\partial \text{E}_n}{\partial w_{j,i}} &= \sum_{k=1}^m \delta_k \frac{\partial \text{E}_n}{\partial w_{k,i}} \\ y_{i-1} &= \sum_{k=1}^m w_{k,i} \end{aligned}$$
en supposant que tous les neurones de la couche  $\{l\}$  sont tels que  $\{w_{j,i}\} = \{w_{k,i}\}$  pour  $\{1 \leq j, k \leq m\}$
- un terme de régularisation est ajouté à la fonctionnelle à minimiser pour contrôler la complexité et la forme de la solution et, par exemple
$$\|P(\mathbf{w})\| = E_n(\mathbf{w}) + \eta P(\mathbf{w})$$
où  $\|P(\mathbf{w})\|$  influence la forme de la solution et  $\eta$  contrôle l'influence du terme de régularisation.  $\|P(\mathbf{w})\|$  peut prendre la forme d'une fonction de la norme  $\|L_p\|$  de  $\mathbf{w}$ . Deux exemples classiques sont :
  - la régularisation  $\|\mathbf{w}\|_2$  :  $\|P(\mathbf{w})\| = \|\mathbf{w}\|_2^2 = \mathbf{w}^\top \mathbf{w}$ , où le principe est de pénaliser les poids de fortes valeurs, qui tendent à amplifier le problème de surapprentissage.
  - la régularisation  $\|\mathbf{w}\|_1$  :  $\|P(\mathbf{w})\| = \|\mathbf{w}\|_1 = \sum_{k=1}^m |w_k|$  où  $|W|$  est la dimension de  $\mathbf{w}$ , qui tend à rendre épars le vecteur de poids (beaucoup de valeurs de poids deviennent nulles).

## Exemple

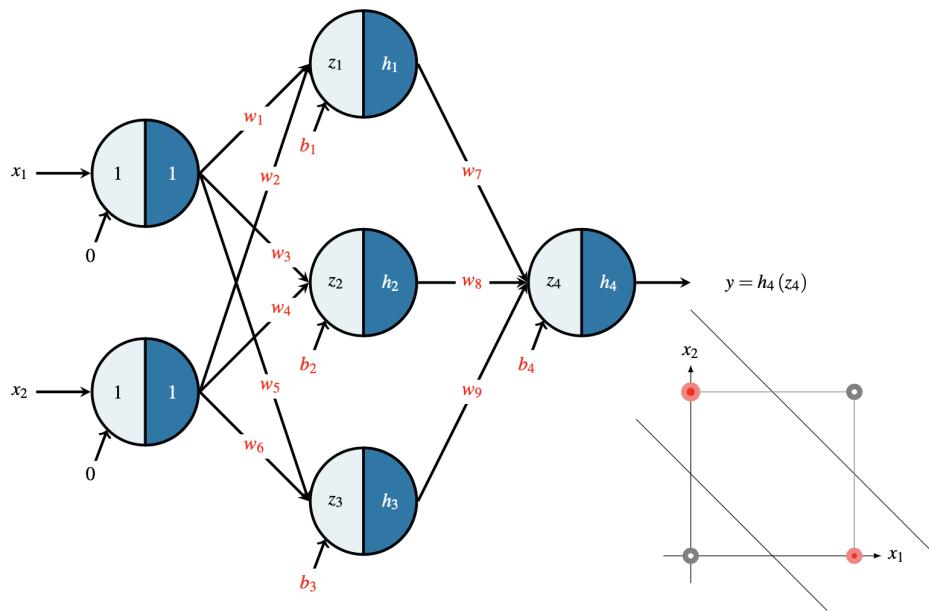
On va considérer le réseau décrit sur la ([Fig. 7](#)) pour apprendre la fonction du OU exclusif (aussi appelé XOR). L'opérateur XOR est défini par sa table de vérité donnée par le tableau suivant

$\{(x_1 \mid\backslash x_2)\}$	0	1
0	0	1
1	1	0

## Réseau

Sur le réseau de la figure ([Fig. 7](#)) les différentes relations sont données par l'équation suivante où les paramètres en rouge correspondent aux poids à calculer durant la phase d'apprentissage.

$$\begin{aligned} z_1 &= w_1 x_1 + w_2 x_2 + b_1 \\ z_2 &= w_3 x_1 + w_4 x_2 + b_2 \\ z_3 &= w_5 x_1 + w_6 x_2 + b_3 \\ z_4 &= w_7 x_1 + w_8 x_2 + b_4 \\ h_1 &= \text{sigmoide}(z_1) \\ h_2 &= \text{sigmoide}(z_2) \\ h_3 &= \text{sigmoide}(z_3) \\ h_4 &= \text{sigmoide}(z_4) \end{aligned}$$



**Fig. 7** Exemple de réseau pour apprendre le XOR

## Phase d'apprentissage

Durant la phase d'apprentissage, on minimise un risque empirique par une fonction de coût. Dans cet exemple, nous allons choisir la minimisation de l'écart quadratique avec la base d'apprentissage labelisée  $\mathcal{E}_a = \left( \mathbf{x}, y \right)$  ou une partie de cette base d'apprentissage  $\mathcal{E}'_a$  :

$$\mathcal{E}'_a = \left( \mathbf{x}, y \right) = \frac{1}{2} \sum_k (y[k] - y)^2 = \frac{1}{2} \sum_k (y[k] - h_4(z_4))^2.$$

On peut utiliser qu'une partie de la base, voire que le  $(k^{\text{ieme}})$  échantillon de la base (gradient stochastique) :

$$\mathcal{E}'_a = E_a = \frac{1}{2} (y[k] - y)^2 = \frac{1}{2} (y[k] - h_4(z_4))^2.$$

L'objectif de la phase d'apprentissage est de mettre à jour les poids du réseau par une approche de descente du gradient. Si l'on considère un poids quelconque du réseau que l'on note  $\theta$ , sa mise à jour durant l'itération  $(n+1)$  se fait pas l'équation suivante:

$$\theta^{(n+1)} = \theta^{(n)} + \gamma \Delta \theta$$

où  $\Delta \theta = -\nabla \mathcal{E}$ .

On peut choisir  $(E = E_{\text{tot}})$  ou  $(E = E_k)$  et pour cet exemple nous choisissons le deuxième cas.

### Couche de sortie

Pour la couche de sortie, prenons par exemple le paramètre  $w_7$ , sa mise à jour est donnée par la relation suivante :

$$w_7^{(n+1)} = w_7^{(n)} - \eta \frac{\partial E}{\partial w_7}.$$

Le problème consiste à calculer  $\frac{\partial E}{\partial w_7}$ , pour cela nous allons utiliser le théorème de dérivation des fonctions composées, d'où:

$$\frac{\partial E}{\partial w_7} = \frac{\partial E}{\partial h_4} \times \frac{\partial h_4}{\partial z_4} \times \frac{\partial z_4}{\partial w_7}.$$

Cette relation est représentée graphiquement sur la (Fig. 8). Utilisant l'expression de  $A$  ( $E_k$ ) on a alors :

$$\begin{aligned} \frac{\partial E}{\partial h_4} &= \frac{\partial E}{\partial z_4} \times \frac{\partial z_4}{\partial h_4} = -\frac{\partial}{\partial z_4} (y[k] - h_4(z_4))^2 = -2(y[k] - h_4(z_4)) \cdot (-h_4'(z_4)) \\ &= -2(y[k] - h_4(z_4)) \cdot (-e^{-z_4}) = 2e^{-z_4}(h_4(z_4) - y[k]) \end{aligned}$$

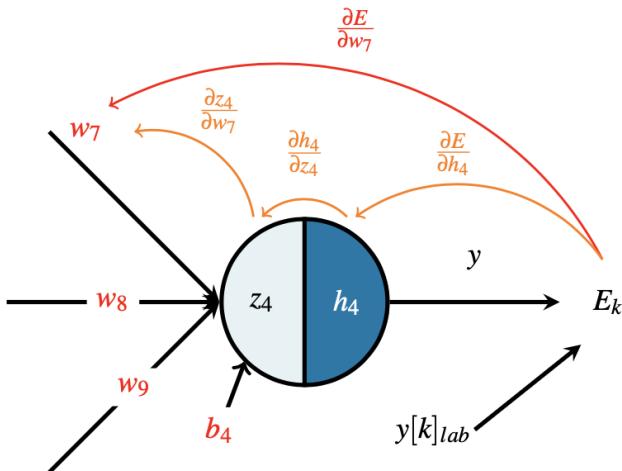
$$h_4 \left( z_4 \right) = w_7 \left( n+1 \right) + (y[k]_{lab} - h_4(z_4)) h_4(z_4) (1 - h_4(z_4)) h_1(z_1)$$

d'où

$$w_7 \left( n+1 \right) = w_7 \left( n \right) + (y[k]_{lab} - h_4(z_4)) h_4(z_4) (1 - h_4(z_4)) h_1(z_1)$$

On peut réaliser la même démarche pour les poids  $w_8$ ,  $w_9$  et  $b_4$ , pour obtenir les relations suivantes :

$$\begin{aligned} & \left[ \begin{array}{l} w_8 \left( n+1 \right) = w_8 \left( n \right) + (y[k]_{lab} - h_4(z_4)) h_4(z_4) (1 - h_4(z_4)) h_2(z_2) \\ w_9 \left( n+1 \right) = w_9 \left( n \right) + (y[k]_{lab} - h_4(z_4)) h_4(z_4) (1 - h_4(z_4)) h_3(z_3) \\ b_4 \left( n+1 \right) = b_4 \left( n \right) + (y[k]_{lab} - h_4(z_4)) h_4(z_4) \end{array} \right. \end{aligned}$$



**Fig. 8** Rétropropagation du gradient sur la couche de sortie.

### Couche cachée

Pour la couche cachée du réseau, c'est exactement le même raisonnement. Prenons par exemple, le paramètre  $w_1$  pour le calcul :

$$w_1 \left( n+1 \right) = w_1 \left( n \right) + \eta \frac{\partial E_k}{\partial w_1}$$

Le problème maintenant consiste à calculer  $\frac{\partial E_k}{\partial w_1}$ , pour cela nous allons utiliser le théorème de dérivation des fonctions composées, d'où :

$$\frac{\partial E_k}{\partial w_1} = \frac{\partial E_k}{\partial z_4} \times \frac{\partial z_4}{\partial w_1} + \frac{\partial E_k}{\partial h_1} \times \frac{\partial h_1}{\partial z_4} + \frac{\partial E_k}{\partial b_1} \times \frac{\partial b_1}{\partial z_4}$$

Cette relation est représentée graphiquement sur la (Fig. 9). Utilisant les équations précédentes, on obtient alors

$$\begin{aligned} & \left[ \begin{array}{l} \frac{\partial E_k}{\partial z_4} = \frac{\partial E_k}{\partial h_1} \times \frac{\partial h_1}{\partial z_4} + \frac{\partial E_k}{\partial b_1} \times \frac{\partial b_1}{\partial z_4} \\ \left( y[k]_{lab} - h_4(z_4) \right) h_4(z_4) (1 - h_4(z_4)) h_1(z_1) \\ \left( y[k]_{lab} - h_4(z_4) \right) h_4(z_4) (1 - h_4(z_4)) \end{array} \right. \end{aligned}$$

d'où

$$w_1 \left( n+1 \right) = w_1 \left( n \right) + (y[k]_{lab} - h_4(z_4)) h_4(z_4) (1 - h_4(z_4)) h_1(z_1) w_7 x_1$$

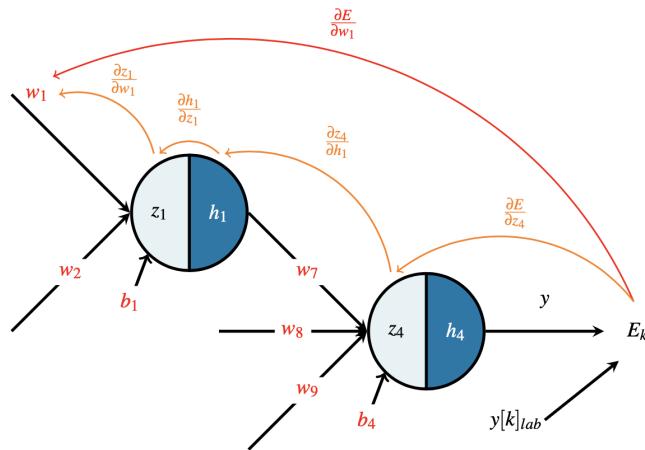
On peut réaliser la même démarche pour les poids  $w_2$  et  $b_1$ , pour obtenir les relations suivantes :

$$\begin{aligned} & \left[ \begin{array}{l} w_2 \left( n+1 \right) = w_2 \left( n \right) + (y[k]_{lab} - h_4(z_4)) h_4(z_4) (1 - h_4(z_4)) h_2(z_2) \\ b_1 \left( n+1 \right) = b_1 \left( n \right) + (y[k]_{lab} - h_4(z_4)) h_4(z_4) (1 - h_4(z_4)) \end{array} \right. \end{aligned}$$

```

h_4\left(z_4\right) \leftarrow h_4\left(1 - h_4\left(z_4\right)\right) \\
h_1\left(z_1\right) \leftarrow h_1\left(1 - h_1\left(z_1\right)\right) \\
w_7 \leftarrow \dots \\
\vdots \\
w_9 \leftarrow \dots

```



**Fig. 9** Rétropropagation du gradient sur la couche cachée.

Il suffit de réaliser des calculs identiques pour les autres neurones et on obtient les relations suivantes :

```

\begin{array}{l}
w_3 \leftarrow w_3 + \eta (y[k]_{lab} - h_4(z_4)) \\
w_4 \leftarrow w_4 + \eta (y[k]_{lab} - h_4(z_4)) \\
w_5 \leftarrow w_5 + \eta (y[k]_{lab} - h_4(z_4)) \\
w_6 \leftarrow w_6 + \eta (y[k]_{lab} - h_4(z_4)) \\
w_7 \leftarrow w_7 + \eta (y[k]_{lab} - h_4(z_4)) \\
w_8 \leftarrow w_8 + \eta (y[k]_{lab} - h_4(z_4)) \\
w_9 \leftarrow w_9 + \eta (y[k]_{lab} - h_4(z_4))
\end{array}

```

#### Initialisation des poids

- les biais sont initialisés à zéro :  $b_1, b_2, b_3, b_4 = 0$  ;
- pour les poids  $w_i$ , ils sont initialisées de façon aléatoire dépendant de la taille de la couche d'avant  $(m^{(l-1)})$  et d'après  $(m^l)$ . L'initialisation de Xavier propose un tirage uniforme dans  $\left[-\sqrt{\frac{6}{m^{(l-1)}+m^l}}, \sqrt{\frac{6}{m^{(l-1)}+m^l}}\right]$ .

Pour notre exemple, on obtient l'initialisation suivante :

```

w_1 \dots w_6 \leftarrow U\left(-\sqrt{\frac{6}{2+3}}, \sqrt{\frac{6}{2+3}}\right)
w_7 \dots w_9 \leftarrow U\left(-\sqrt{\frac{6}{3+1}}, \sqrt{\frac{6}{3+1}}\right)

```

## Implémentation

Nous reprenons les exemples du perceptron et montrons qu'un PMC à une couche cachée permet de résoudre le problème de séparation non linéaire.

```

import numpy as np
from matplotlib import pyplot as plt
from matplotlib import colors
import torch
import torch.nn as nn
import torch.optim as optim

fichiers_train =
['./data/linear_data_train.csv','./data/twocircles_data_train.csv','./data/moon_da
ta_train.csv']
fichiers_test =
['./data/linear_data_eval.csv','./data/twocircles_data_eval.csv','./data/moon_data
_eval.csv']

# Fonction de lecture des jeux de données
def extract_data(filename):

    labels = []
    features = []

    for line in open(filename):
        row = line.split(",")
        labels.append(int(row[0]))
        features.append([float(x) for x in row[1:]])

    features_np = np.matrix(features).astype(np.float32)

    labels_np = np.array(labels).astype(dtype=np.uint8)
    labels_onehot = (np.arange(num_labels) == labels_np[:, None]).astype(np.float32)

    return features_np, labels_onehot

```

On écrit une fonction permettant de visualiser le résultat de la classification par le perceptron.

```

def plotResults(ax,ay,X,Y,model,title=pltloss,name):
    mins = np.amin(X,0);
    mins = mins - 0.1*np.abs(mins);
    maxs = np.amax(X,0);
    maxs = maxs + 0.1*maxs;

    xs,ys =
    np.meshgrid(np.linspace(mins[0,0],maxs[0,0],300),np.linspace(mins[0,1], maxs[0,1],
    300));

    toto = torch.FloatTensor(np.c_[xs.flatten(), ys.flatten()])
    Z = np.argmax(model(toto).detach().numpy(), axis=-1)
    Z=Z.reshape(xs.shape[0],xs.shape[1])

    labelY = np.matrix(Y[:, 0]+2*Y[:, 1])
    labelY = labelY.reshape(np.array(X[:, 0]).shape)

    ax.contourf(xs, ys, Z, cmap=colors.ListedColormap([[0,0.5,0.66,0],
    [0.93,0.76,0.27,0]]),alpha=.5)
    ax.scatter(np.array(X[:, 0]),np.array(X[:, 1]),c=
    np.array(labelY),s=20,cmap=colors.ListedColormap(['red', 'green']))
    ax.set_title(title)

    ay.plot(pltloss)
    ay.set_title("perte sur " + name)
    ay.set_xlabel("epoch")

    plt.tight_layout()

```

On implémente une classe PMC, dérivée de la classe `nn.Module`.

```

# Nombre de neurones de la couche cachée
num_hidden = 5

class PMC(nn.Module):
    def __init__(self,p):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(num_features, num_hidden),
            nn.Tanh(),
            nn.Linear(num_hidden, num_labels),
            nn.Softmax(1),
        )

    def forward(self, x):
        return self.layers(x)

```

et on définit la fonction d'entraînement. La fonction de perte est l'["entropie croisée binaire](#) et l'optimiseur est [Adam](#).

```
# Paramètres de l'apprentissage
batch_size = 100
num_epochs = 10000
num_labels = 2
num_features = 2

loss = nn.BCELoss()

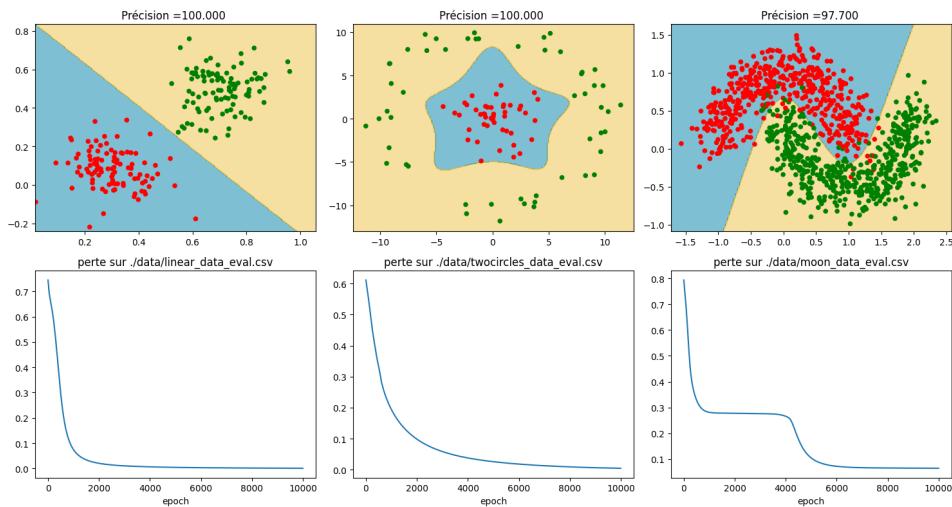
def train_session(X,y,classifier,criterion,optimizer,n_epochs=num_epochs):
    loss_values = []
    losses = np.zeros(n_epochs)
    correct = 0
    for iter in range(n_epochs):
        optimizer.zero_grad()
        yPred = classifier(X)
        loss = criterion(yPred,y)
        loss_values.append(loss.detach().numpy())
        #Gradient et rétropropagation
        loss.backward()
        #Mise à jour des poids
        optimizer.step()
        y2 = yPred>0.5
        correct = (y2 == y).sum().item()/2
    acc = 100 * correct / (X.shape[0])
    return loss_values,acc
```

On applique enfin le perceptron sur les jeux de données et on visualise les résultats.

```
fig,axs = plt.subplots(2, 3, figsize=(15,8))
for i,(name_train,name_test) in enumerate(zip([0,1,2],fichiers_train,fichiers_test)):
    train_data,train_labels = extract_data(name_train)
    test_data, test_labels = extract_data(name_test)

    model = PMC(train_data.shape[1])
    optimizer = optim.Adam(model.parameters())
    pltloss,acc =
    train_session(torch.FloatTensor(train_data),torch.FloatTensor(train_labels),model,
    loss,optimizer)

    titre= "Précision ={0:.3f} ".format(acc)
    plotResults(axs[0][i],axs[1][i],test_data, test_labels, model, titre, pltloss,
    name_test)
```



## Réseaux convolutifs

### Introduction

#### Inspiration biologique

Un réseau de neurones convolutif (CNN, *Convolutional Neural Network* ou ConvNet) est un type de réseau de neurones artificiels acyclique à propagation avant, dans lequel le motif de connexion entre les neurones est inspiré par le cortex visuel des animaux. Les neurones de cette région du cerveau sont arrangés de sorte à ce qu'ils correspondent à des régions (appelées champs réceptifs) qui se chevauchent lors du pavage du champ visuel. Ils sont de plus organisés de manière hiérarchique, en couches (aire visuelle primaire V1, secondaire V2, puis aires V3, V4, V5 et V6, gyrus temporal inférieur), chacune des couches étant spécialisée dans une tâche, de plus en plus abstraite en allant de l'entrée vers la sortie. En simplifiant à l'extrême, une fois que les signaux lumineux sont reçus par la rétine et convertis en potentiels d'action :

- L'aire primaire V1 s'intéresse principalement à la détection de contours, ces contours étant définis comme des zones de fort contraste de signaux visuels reçus.
- L'aire V2 reçoit les informations de V1 et extrait des informations telles que la fréquence spatiale, l'orientation, ou encore la couleur.
- L'aire V4, qui reçoit des informations de V2, mais aussi de V1 directement, détecte des caractéristiques plus complexes et abstraites liées par exemple à la forme.
- Le gyrus temporal inférieur est chargé de la partie sémantique (reconnaissance des objets), à partir des informations reçues des aires précédentes et d'une mémoire des informations stockées sur des objets.

L'architecture et le fonctionnement des réseaux convolutifs sont inspirés par ces processus biologiques. Ces réseaux consistent en un empilage multicouche de perceptrons, dont le but est de prétraiter de petites quantités d'informations. Les réseaux convolutifs ont de larges applications dans la reconnaissance d'image et vidéo, les systèmes de recommandation et le traitement du langage naturel.

Un réseau convolutif se compose de deux types de neurones, agencés en couches traitant successivement l'information. Dans le cas du traitement de données de type images, on a ainsi :

- des *neurones de traitement*, qui traitent une portion limitée de l'image (le champ réceptif) au travers d'une fonction de convolution;
- des *neurones de mise en commun* des sorties dits *d'agrégation totale ou partielle (pooling)*.

Un traitement correctif non linéaire est appliqué entre chaque couche pour améliorer la pertinence du résultat. L'ensemble des sorties d'une couche de traitement permet de reconstituer une image intermédiaire, dite carte de caractéristiques (feature map), qui sert de base à la couche suivante. Les couches et leurs connexions apprennent des niveaux d'abstraction croissants et extraient des caractéristiques de plus en plus haut niveau des données d'entrée.

Dans la suite, le propos sera illustré sur des images 2D en niveaux de gris, de taille  $(n_1 \times n_2)$  :

$\begin{aligned} & \mathbf{R}_{(i,j)} : \dots \times \dots & \mathbf{R}_{(i,j)} & \text{ sera indifféremment vue comme une fonction ou une matrice.} \end{aligned}$

## Convolution discrète

Pour reproduire la notion de champ réceptif, et ainsi permettre aux neurones de détecter des caractéristiques de petite taille mais porteurs d'information, l'idée est de laisser un neurone caché voir et traiter seulement une petite portion de l'image qu'il prend en entrée. L'outil retenu dans les réseaux convolutifs est la convolution discrète.

### Definition 5 (Convolution discrète)

Soient  $\{h_1, h_2\} \in \mathbb{N}$ ,  $K \in \mathbb{R}^{(2h_1+1) \times (2h_2+1)}$ . La convolution discrète de  $\{I\}$  par le filtre  $\{K\}$  est donnée par :

$$\begin{aligned} & \begin{aligned} & \text{\left(\left(\mathbf{K}\right)\right)} \\ & \text{\left(\left(\mathbf{K}\right)\right)} \end{aligned} \\ & \text{où } \{K\} \text{ est donné par :} \\ & \begin{aligned} & \begin{aligned} & \text{\left(\left(\mathbf{K}\right)\right)} \\ & \text{\left(\left(\mathbf{K}\right)\right)} \end{aligned} \end{aligned} \end{math>$$

La taille du filtre  $((2h_1+1) \times (2h_2+1))$  précise le champ visuel capturé et traité par  $\{K\}$ .

Lorsque  $\{K\}$  parcourt  $\{I\}$ , le déplacement du filtre est réglé par deux paramètres de *stride* (horizontal et vertical). Un stride de 1 horizontal (respectivement vertical) signifie que  $\{K\}$  se déplace d'une position horizontale (resp. verticale) à chaque application de l'équation précédente. Les valeurs de stride peuvent également être supérieures et ainsi sous-échantillonner  $\{I\}$ .

Le comportement du filtre sur les bords de  $\{I\}$  doit également être précisé, par l'intermédiaire d'un paramètre de *padding*. Si l'image convoluée  $\{\left(\left(I\right)\right) \ast K\}$  doit posséder la même taille que  $\{I\}$ , alors  $(2h_1)$  lignes de 0 ( $(h_1)$  en haut et  $(h_1)$  en bas) et  $(2h_2)$  colonnes de 0 ( $(h_2)$  à gauche et  $(h_2)$  à droite) doivent être ajoutées. Dans le cas où la convolution est réalisée sans padding, l'image convoluée est de taille  $((n_1-2h_1) \times (n_2-2h_2))$ .

## Définition des couches

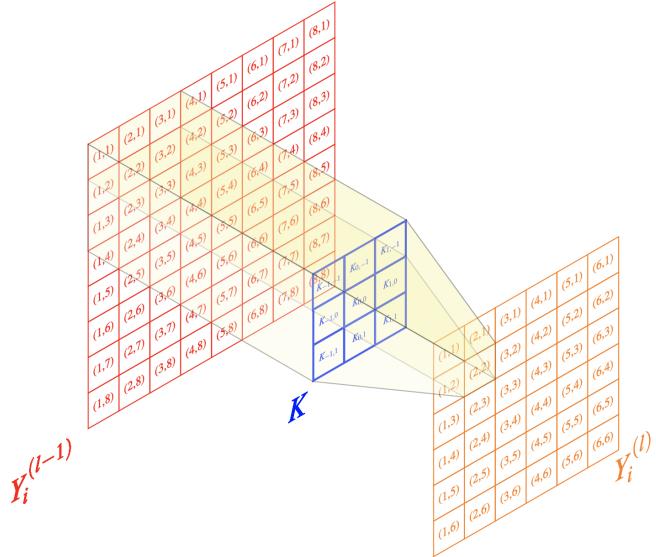
Nous introduisons ici les principaux types de couches utilisées dans les réseaux convolutifs.

L'assemblage de ces couches permet de construire des architectures complexes pour la classification ou la régression.

### Couche de convolution

Soit  $\{I\}$  une couche de convolution. L'entrée de la couche  $\{I\}$  est composée de  $n^{(l-1)}$  cartes provenant de la couche précédente, de taille  $(n_1^{(l-1)} \times n_2^{(l-1)})$ . Dans le cas de la couche d'entrée du réseau ( $(l=1)$ ), l'entrée est l'image  $\{I\}$ . La sortie de la couche  $\{I\}$  est formée de  $n^l$  cartes de taille  $(n_1^l \times n_2^l)$ . La  $i^{\text{e}}$  carte de la couche  $\{I\}$ , notée  $\{Y_i\}$ , se calcule comme :

$$\begin{aligned} & \begin{aligned} & \mathbf{Y}_i = \mathbf{B}^l \mathbf{Y}_{i-1} + \sum_{j=1}^{n^l} \mathbf{K}^l_{i,j} \end{aligned} \\ & \text{où } \{B^l\} \text{ est une matrice de biais et } \{K^l\} \text{ est le filtre de taille } \\ & ((2h_1^{(l)} + 1) \times (2h_2^{(l)} + 1)) \text{ connectant la } j^{\text{e}} \text{ carte de la couche } ((l-1)) \text{ à la } i^{\text{e}} \text{ carte de la couche } (l) \text{ (Fig. 10).} \end{aligned}$$



**Fig. 10** Illustration des calculs effectués dans une opération de convolution discrète.

Le pixel  $(2,2)$  de l'image  $\mathbf{Y}_i^{(l)}$  est une combinaison linéaire des pixels \

$((i,j)) \in \{1, 2, 3\}^2$  de  $\mathbf{Y}_i^{(l-1)}$  les coefficients de la combinaison étant

portés par le filtre  $\mathbf{K}$ .

$n_1^{(l)}$  et  $n_2^{(l)}$  doivent prendre en compte les effets de bords : lors du calcul de la convolution, seuls les pixels dont la somme est définie avec des indices positifs doivent être traités. Dans le cas où le padding n'est pas utilisé, les cartes de sortie ont donc une taille de  $n_1^{(l)} = n_1^{(l-1)} - 2h_1^{(l)}$  et  $n_2^{(l)} = n_2^{(l-1)} - 2h_2^{(l)}$ .

Souvent, les filtres utilisés pour calculer  $\mathbf{Y}_i^{(l)}$  sont les mêmes, i.e.  $\mathbf{K}_{i,j}^{(l)} = \mathbf{K}_{i,k}^{(l)}$  pour  $j \neq k$ . De plus, la somme dans l'équation de la convolution peut être conduite sur un sous-ensemble des cartes d'entrée.

Il est possible de mettre en correspondance la couche de convolution, et l'opération qu'elle effectue, avec un perceptron multicouche. Pour cela, il suffit de réécrire l'équation : chaque carte  $\mathbf{Y}_i^{(l)}$  de la couche  $l$  est formée de  $n_1^{(l)} \cdot n_2^{(l)}$  neurones organisés dans un tableau à deux dimensions. Le neurone en position  $(r,s)$  calcule :

$$\begin{aligned} & \left[ \begin{aligned} & \text{left}(\mathbf{Y}_i^{(l)}) \right]_{r,s} = \left[ \begin{aligned} & \text{left}(\mathbf{B}_i^{(l)}) \right]_{r,s} \\ & + \sum_{j=1}^{n_1^{(l-1)}} \sum_{k=1}^{n_2^{(l-1)}} \mathbf{K}_{i,j}^{(l)} \mathbf{Y}_j^{(l-1)}_{k,s} \\ & = \left[ \begin{aligned} & \text{left}(\mathbf{B}_i^{(l)}) \right]_{r,s} + \sum_{j=1}^{n_1^{(l-1)}} \sum_{k=1}^{n_2^{(l-1)}} \left[ \begin{aligned} & \text{left}(\mathbf{K}^{(l)}) \right]_{i,j} \mathbf{Y}_j^{(l-1)}_{k,s} \end{aligned} \right] \end{aligned} \right] \end{aligned}$$

Les paramètres du réseau à entraîner (poids) peuvent alors être trouvés dans les filtres  $\mathbf{K}^{(l)}$  et les matrices de biais  $\mathbf{B}_i^{(l)}$ .

Comme nous le verrons plus loin, un sous-échantillonnage est utilisé pour diminuer l'influence du bruit et des distorsions dans les images. Le sous-échantillonnage peut être également réalisé simplement avec des paramètres de stride, en sautant un nombre fixe de pixels dans les dimensions horizontale (saut  $s_1^{(l)}$ ) et verticale (saut  $s_2^{(l)}$ ) avant d'appliquer de nouveau le filtre. La taille des images de sortie est alors :

$$\begin{aligned} n_1^{(l)} &= \frac{n_1^{(l-1)} - 2h_1^{(l)}}{s_1^{(l)}} + 1 \quad \text{et} \quad n_2^{(l)} \\ &= \frac{n_2^{(l-1)} - 2h_2^{(l)}}{s_2^{(l)}} + 1. \end{aligned}$$

Un point clé des réseaux convolutifs est d'exploiter la corrélation spatiale des données. L'utilisation des noyaux permet d'alléger le modèle, plutôt que d'utiliser des couches complètement connectées.

## Couche non linéaire

Pour augmenter le pouvoir d'expression des réseaux profonds, on utilise des couches non linéaires.

Les entrées d'une couche non linéaire sont  $(n^{(l-1)})$  cartes et ses sorties  $(n^{(l)}) = n^{(l-1)}$  cartes  $(\mathbf{Y}_i^{(l)})$ , de taille  $(n_1^{(l-1)} \times n_2^{(l-1)})$  telles que  $(n_1^{(l)}) = n_1^{(l-1)}$  et  $(n_2^{(l)}) = n_2^{(l-1)}$ , données par  $(\mathbf{Y}_i^{(l)}) = (f(\mathbf{Y}_i^{(l-1)}))$ , où  $f$  est la fonction d'activation utilisée dans la couche  $(l)$ .

La [Fig. 3](#) du cours sur les perceptrons multicouches présente quelques fonctions d'activation classiques.

En apprentissage profond, il a été reporté que la sigmoïde et la tangente hyperbolique avaient des performances moindres que la fonction d'activation *softsign* :

$$[\begin{aligned} \mathbf{Y}_i^{(l)} &= \frac{1}{1 + \exp(-\mathbf{Y}_i^{(l-1)})}. \end{aligned}]$$

En effet, les valeurs des pixels des cartes  $(\mathbf{Y}_i^{(l-1)})$  arrivant près des paliers de saturation de ces fonctions donnent des gradients faibles, qui ont tendance à s'annuler (*gradient évanescant ou vanishing gradient*) lors de la phase d'apprentissage par rétropropagation du gradient. Une autre fonction, non saturante elle, est très largement utilisée. Il s'agit de la fonction ReLU (Rectified Linear Unit) [\[1\]](#) :

$$[\begin{aligned} \mathbf{Y}_i^{(l)} &= \max(0, \mathbf{Y}_i^{(l-1)}). \end{aligned}]$$

Les neurones utilisant la fonction ReLU sont appelés neurones linéaires rectifiés. Glorot et Bengio [\[2\]](#) ont montré que l'utilisation d'une couche ReLU en tant que couche non linéaire permettait un entraînement efficace de réseaux profonds sans pré-entraînement non supervisé. Plusieurs variantes de cette fonction existent, par exemple pour assurer une différentiabilité en 0 ou pour proposer des valeurs non nulles pour des valeurs négatives de l'argument.

## Couches de normalisation

La normalisation prend aujourd'hui une place de plus en plus importante, notamment depuis les travaux de Ioffe et Szegedy [\[3\]](#). Les auteurs suggèrent qu'un changement dans la distribution des activations d'un réseau profond, résultant de la présentation d'un nouveau mini batch d'exemples, ralentit le processus d'apprentissage. Pour pallier ce problème, chaque activation du mini batch est centrée et normée (variance unité), la moyenne et la variance étant calculées sur le mini batch entier, indépendamment pour chaque activation. Des paramètres d'offset  $(\beta)$  et multiplicatif  $(\gamma)$  sont alors appliqués pour normaliser les données d'entrée ([Algorithm 5](#)).

---

### Algorithm 5 (Normalisation par batch sur la présentation d'un mini batch $(\mathcal{B})$ )

---

Entrées : valeurs de l'activation  $(x)$  sur un mini batch  $(\mathcal{B}) = (x_1 \dots x_m)$ , Paramètres  $(\beta, \gamma)$  à apprendre

Sortie : Données normalisées  $((y_1 \dots y_m) = \text{BatchNorm}_{\gamma, \beta}(x_1 \dots x_m))$

1.  $(\mu_{\mathcal{B}} = \frac{1}{m} \sum_{i=1}^m x_i)$
2.  $(\sigma^2_{\mathcal{B}} = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2)$
3. Pour  $i=1$  à  $m$ 
  1.  $(y_i = \gamma \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} + \beta)$

Lorsque la descente de gradient est achevée, un post apprentissage est appliqué dans lequel la moyenne et la variance sont calculées sur l'ensemble d'entraînement et remplacent  $(\mu_{\mathcal{B}})$  et  $(\sigma^2_{\mathcal{B}})$  ([Algorithm 6](#)).

---

**Algorithm 6 (Normalisation par batch d'un réseau)**


---

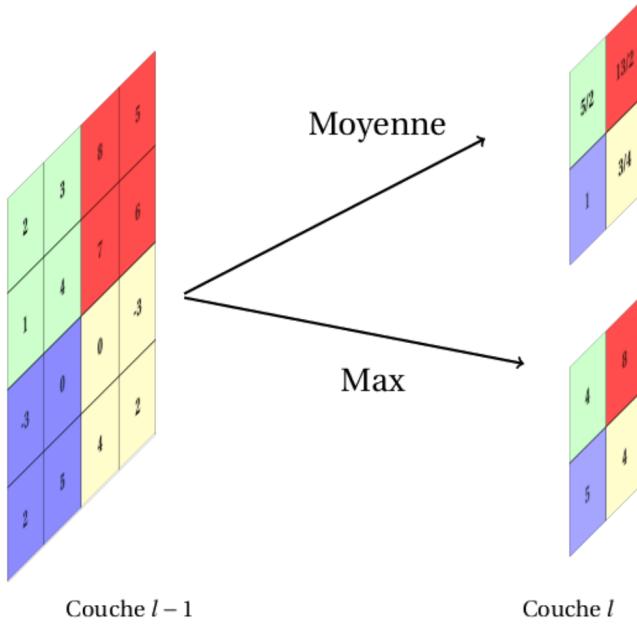
{ Entrées : {un réseau  $\mathcal{N}$ , un ensemble d'activations  $\{x^1 \dots x^K\}$ }

1.  $\mathcal{N}_n = N$
2. Pour  $i=1$  à  $K$ 
  1. Calculer  $y^i = BN_{\{\gamma, \beta\}}(x^i)$  à l'aide de l'[Algorithm 5](#)
  2. Modifier chaque couche de  $\mathcal{N}_n$  : l'entrée  $y^i$  remplace l'entrée  $x^i$
3. Entrainer  $\mathcal{N}_n$  pour optimiser les paramètres de  $\mathcal{N}$  et  $\{\gamma_i, \beta_i\}_{i \in [1, K]}$
4.  $N^f = \mathcal{N}_n$
5. Pour  $i=1$  à  $K$ 
  1. Utiliser  $N^f$  sur des batchs  $\{\mathcal{B}\}$  de taille  $m$
  2. Calculer la moyenne des moyennes  $\{\bar{x}^i\}$  et des variances  $\{Var(x^i)\}$
  3. Remplacer dans  $N^f$  la transformation  $y^i = BN_{\{\gamma, \beta\}}(x^i)$  par
$$y^i = \frac{\gamma_i}{\sqrt{Var(x^i) + \epsilon}} x^i + \left(\beta_i - \frac{\gamma_i}{\sqrt{Var(x^i) + \epsilon}} \bar{x}^i\right)$$

### Couche d'agrégation et de sous-échantillonnage

Le sous-échantillonnage (pooling) des cartes obtenues par les couches précédentes a pour objectif d'assurer une robustesse au bruit et aux distorsions.

La sortie d'une couche d'agrégation  $\mathcal{I}$  ([Fig. 11](#)) est composée de  $n^{|\mathcal{I}|} = n^{|\mathcal{I}-1|}$  cartes de taille réduite. En général, l'agrégation est effectuée en déplaçant dans les cartes d'entrée une fenêtre de taille  $(2p \times 2p)$  toutes les  $q$  positions (il y a recouvrement si  $q < p$  et non recouvrement sinon), et en calculant, pour chaque position de la fenêtre, une seule valeur, affectée à la position centrale dans la carte de sortie.



**Fig. 11** Couche d'agrégation et de sous-échantillonnage  $\mathcal{I}$ . Chacune des  $n^{|\mathcal{I}-1|}$  cartes de la couche  $\mathcal{I}-1$  est traitée individuellement. Chaque neurone des  $n^{|\mathcal{I}|} = n^{|\mathcal{I}-1|}$  cartes de sortie est la moyenne (ou le maximum) des valeurs contenues dans une fenêtre de taille donnée dans la carte correspondante de la couche  $\mathcal{I}-1$ .

On distingue généralement deux types d'agrégation :

- La moyenne : on utilise un filtre  $\mathbf{K}_B$  de taille  $((2h_1 + 1) \times (2h_2 + 1))$  défini par
$$\left[ \left( \mathbf{K}_B \right)_{r,s} = \frac{1}{(2h_1 + 1)(2h_2 + 1)} \right]$$
- Le maximum : la valeur maximum dans la fenêtre est retenue.

Le maximum est souvent utilisé pour assurer une convergence rapide durant la phase d'entraînement. L'agrégation avec recouvrement, elle, semble assurer une réduction du phénomène de surapprentissage

## Couche complètement connectée

Si  $(l)$  et  $(l-1)$  sont des couches complètement connectées, l'équation :

$$\begin{aligned} z_i^{(l)} = \sum_{k=0}^m w_{ik}^{(l-1)} y_k^{(l-1)} \quad \text{où } \\ \mathbf{Z}^{(l)} = \mathbf{W}^{(l-1)} \mathbf{Y}^{(l-1)} \end{aligned}$$

avec  $(\mathbf{Z}^{(l)})$ ,  $(\mathbf{W}^{(l-1)})$  et  $(\mathbf{Y}^{(l-1)})$  les représentations vectorielle et matricielle des entrées  $(z_i^{(l)})$ , des poids  $(w_{ik}^{(l-1)})$  et des sorties  $(y_k^{(l-1)})$ , permet de relier ces deux couches.

Dans le cas contraire, la couche  $(l)$  attend  $(n^{(l-1)})$  entrées de taille  $(n_1^{(l-1)} \times n_2^{(l-1)})$  et le  $(i^{(text{e})})$  neurone de la couche  $(l)$  calcule :

$$y_i^{(l)} = f(\sum_{j=1}^{n^{(l-1)}} \sum_{r=1}^{n_1^{(l-1)}} \sum_{s=1}^{n_2^{(l-1)}} w_{irs}^{(l-1)} Y_{rs}^{(l-1)})$$

où  $(w_{irs}^{(l-1)})$  est le poids connectant le neurone en position  $((r,s))$  de la  $(j^{(text{e})})$  carte de la couche  $((l-1))$  au  $(i^{(text{e})})$  neurone de la couche  $(l)$ .

En pratique, les réseaux convolutifs sont utilisés pour apprendre une hiérarchie dans les données et la (ou les) couche(s) complètement connectée(s) est(sont) utilisée(s) en bout de réseau pour des tâches de classification ou de régression.

Une couche de classification classiquement mise en œuvre utilise le classifieur *softmax*, qui généralise la régression logistique au cas multiclasse ( $k$  classes). L'ensemble d'apprentissage  $\{\mathcal{E}_a = \{(x^{(i)}, y^{(i)}), i \in [1, m]\}\}$  est donc tel que  $(y^{(i)}) \in [0, 1]^k$  et le classifieur estime la probabilité  $(P(y^{(i)} = j) | \mathbf{x}^{(i)})$  pour chaque classe  $(j \in [1, k])$ . Le classifieur softmax calcule cette probabilité selon :

$$\forall j \in [1, k] \quad P(y^{(i)} = j | \mathbf{x}^{(i)}) = \frac{e^{\mathbf{W}_j^\top \mathbf{x}^{(i)}}}{\sum_{l=1}^k e^{\mathbf{W}_l^\top \mathbf{x}^{(i)}}}$$

où  $(\mathbf{W})$  est la matrice des paramètres du modèle (les poids). Ces paramètres sont obtenus en minimisant une fonction de coût, qui peut par exemple s'écrire :

$$\begin{aligned} J(\mathbf{W}) &= -\frac{1}{m} \sum_{i=1}^m \sum_{j=1}^k y_j^{(i)} \log (\mathbf{W}_j^\top \mathbf{x}^{(i)}) + \frac{\lambda}{2} \sum_{i=1}^m \sum_{j=1}^k W_{ji}^2 \end{aligned}$$

où  $(\lambda)$  est un paramètre de régularisation contrôlant le second terme du coût qui pénalise les grandes valeurs des poids (régularisation  $(\ell_2)$ ).

## Régularisation

Un des enjeux principaux en apprentissage automatique est de construire des algorithmes ayant une bonne capacité de généralisation. Les stratégies mises en œuvre pour arriver à cette fin rentrent dans la catégorie générale de la régularisation et de nombreuses méthodes sont aujourd'hui proposées en ce sens. La régularisation a déjà été abordée dans le chapitre consacré aux perceptrons multicouches (voir section [Régularisation](#)). Nous faisons ici un focus sur trois stratégies largement utilisées en apprentissage profond.

### Régularisation de la fonction de coût

L'équation précédente est un exemple de régularisation de la fonction de coût, utilisée lors de la phase d'entraînement. À la fonction d'erreur est ajoutée une fonction des poids du réseau, qui peut prendre de multiples formes. Les deux principales stratégies sont :

- La régularisation  $\|\ell_2\|$  (ou ridge regression), qui force les poids à avoir une faible valeur absolue : un terme de régularisation fonction de la norme  $\|\ell_2\|$  de la matrice des poids est ajouté. On parle souvent de *weight decay*.
- La régularisation  $\|\ell_1\|$ , qui tend à rendre épars le réseau profond, *i.e.* à imposer à un maximum de poids de s'annuler. Un terme de régularisation, somme pondérée des valeurs absolues des poids, est ajouté à la fonction objectif.

## Dropout

Les techniques de dropout se rapprochent des stratégies classiques de bagging en apprentissage automatique. L'objectif est d'entraîner un ensemble constitué de tous les sous-réseaux qui peuvent être construits en supprimant des neurones (hors neurones d'entrée et de sortie) du réseau initial. Si le réseau comporte  $\|\mathbf{W}\|$  neurones cachés, il existe ainsi  $(2^{\|\mathbf{W}\|})$  modèles possibles. En pratique, les neurones cachés se voient perturbés par un bruit binomial, qui a pour effet de les empêcher de fonctionner en groupe et de les rendre, au contraire, plus indépendants. Le phénomène de surapprentissage est ainsi fortement réduit sur le réseau, qui doit décomposer les entrées en caractéristiques pertinentes, indépendamment les unes des autres. Les réseaux construits par dropout partagent partiellement leurs paramètres, ce qui diminue l'empreinte mémoire de la méthode.

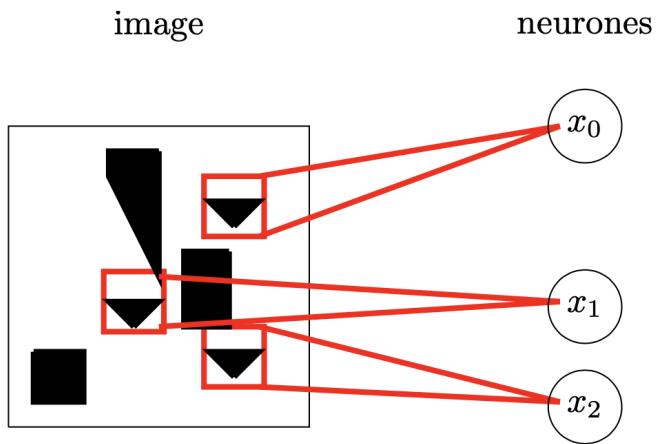
Lors de la phase de prédiction, le réseau complet est utilisé, mais les neurones cachés sont pondérés par la fraction de bruit utilisé pendant l'apprentissage (*i.e.* pour chaque neurone le nombre de fois où il a été supprimé d'un sous-réseau, rapporté au nombre total de réseaux), afin de conserver la valeur moyenne des activations des neurones identiques à celles durant l'apprentissage.

Notons qu'il est également possible d'éteindre non pas un neurone, mais un poids. La stratégie correspondante est appelée DropConnect.

## Partage de paramètres

La régularisation de la fonction de coût permet d'imposer aux poids certaines contraintes (par exemple de rester faibles en amplitude pour la régularisation  $\|\ell_2\|$ , ou de s'annuler pour la régularisation  $\|\ell_1\|$ ). Il peut également être intéressant d'imposer certains *a priori* sur les poids, par exemple une dépendance entre les valeurs des paramètres.

Une dépendance classique consiste à imposer que les valeurs de certains poids soient proches les unes des autres (dans le cas par exemple où deux modèles de classification  $\|\mathbf{M}_1\|$  et  $\|\mathbf{M}_2\|$ , de paramètres  $\|\mathbf{W}_1\|$  et  $\|\mathbf{W}_2\|$ , opèrent sur des données similaires et sur des classes identiques) et, là encore, une stratégie de pénalisation de la fonction objectif peut être utilisée. Cependant, il est plus courant dans ce cas d'imposer que les paramètres soient égaux (dans l'exemple précédent imposer  $\|\mathbf{W}_1\| = \|\mathbf{W}_2\|$ ) et d'arriver à une stratégie dite de partage des paramètres. Dans le cas des réseaux convolutifs utilisés en vision, cette régularisation est assez intuitive puisque les entrées (images) possèdent de nombreuses propriétés invariantes par transformations affines (une image de voiture reste une image de voiture, même si l'image est translatée ou mise à l'échelle ([Fig. 12](#))). Le réseau exploite alors ce partage de paramètres, en calculant une même caractéristique (un neurone et son poids) à différentes positions dans l'image. De ce fait, le nombre de paramètres est drastiquement réduit, ainsi que l'empreinte mémoire du réseau appris.



**Fig. 12** Partage de paramètres : les neurones voient des champs réceptifs distincts, mais partagent les mêmes paramètres (poids). Leur capacité de détection d'un triangle restera la même, quelle que soit la position de l'objet dans l'image.

## Initialisation

Une initialisation convenable des poids est essentielle pour assurer une convergence de la phase d'entraînement. Un choix arbitraire des poids (à zéro, à de petites ou grandes valeurs aléatoires) peut ralentir, voire causer de la redondance dans le réseau (problème de la symétrie). Ces aspects ont déjà été abordés dans la section [Initialisation des poids](#), où l'initialisation de Xavier

$$\begin{aligned} - \frac{\sqrt{6}}{\sqrt{m^l + m^{l-1}}} < w_{i,l} < \frac{\sqrt{6}}{\sqrt{m^l + m^{l-1}}} \end{aligned}$$

a été notamment détaillée.

## Apprentissage

La présence de nombreuses couches cachées va permettre de calculer des caractéristiques beaucoup plus complexes et informatives des entrées. Chaque couche calculant une transformation non linéaire de la couche précédente, le pouvoir de représentation de ces réseaux s'en trouve amélioré. On peut par exemple montrer qu'il existe des fonctions qu'un réseau à  $k$  couches peut représenter de manière compacte (avec un nombre de neurones cachés qui est polynomial en le nombre des entrées), alors qu'un réseau à  $k-1$  couches ne peut pas le faire, à moins d'avoir une combinatoire exponentielle sur le nombre de neurones cachés.

## Le problème de l'entraînement

Si l'intérêt de ces réseaux est manifeste, la complexité de leur utilisation vient de l'étape d'apprentissage. Jusqu'à récemment, l'algorithme utilisé était classique et consistait en une initialisation aléatoire des poids du réseau, suivie d'un entraînement sur un ensemble d'apprentissage, en minimisant une fonction objectif. Cependant, dans le cas des réseaux profonds, cette approche peut ne pas être adaptée :

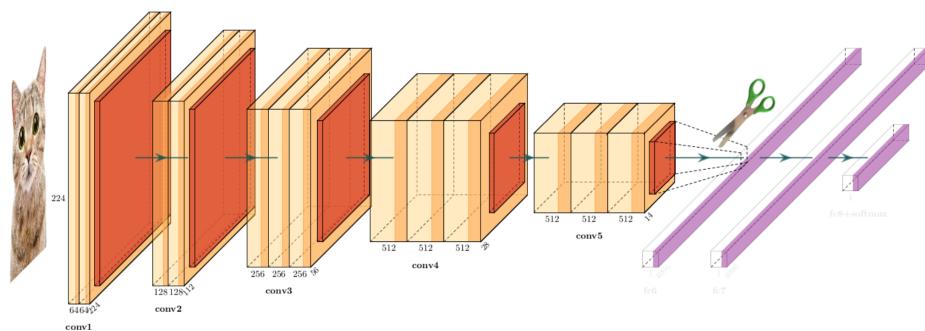
- Les données étiquetées doivent être en nombre suffisant pour permettre un entraînement efficace, d'autant plus que le réseau est complexe. Dans le cas contraire, un surapprentissage peut notamment être induit.
- Sur un tel réseau, l'apprentissage se résume à l'optimisation d'une fonction fortement non convexe, qui amène presque sûrement à des minima locaux lorsque des algorithmes classiques sont utilisés.
- Dans l'étape de rétropropagation, les gradients diminuent rapidement à mesure que le nombre de couches cachées augmente. La dérivée de la fonction objectif par rapport à  $\mathbf{W}$  devient alors très faible à mesure que le calcul se rétropropage vers la couche d'entrée. Les

poids des premières couches changent donc très lentement et le réseau n'est plus en capacité d'apprendre. Ce problème est connu sous le nom de problème de gradient évanescant (vanishing gradient).

L'algorithme principalement utilisé pour l'apprentissage des réseaux convolutifs reste la rétropropagation du gradient. Le choix de la fonction objectif, de sa régularisation, de la méthode d'optimisation (descente de gradient, méthodes à taux d'apprentissage adaptatifs telles qu'AdaGrad, RMSProp ou Adam) et des paramètres associés, ou des techniques de présentation des exemples (batchs, minibatchs) sont autant de facteurs importants permettant aux modèles non seulement de converger vers un optimum local satisfaisant, mais également de proposer un modèle final ayant une bonne capacité de généralisation.

Aujourd'hui, de nombreux réseaux, déjà entraînés, sont mis à disposition. En effet, ces entraînements nécessitent de grandes bases d'apprentissage (type ImageNet) et une puissance de calcul assez élevée (GPUs obligatoires). Pour le traitement de problèmes précis, des méthodes existent, qui partent de ces réseaux préentraînés et les modifient localement pour, par exemple, apprendre de nouvelles classes d'images non encore vues par le réseau. L'idée sous-jacente est que les premières couches capturent des caractéristiques bas niveau et que la sémantique vient avec les couches profondes. Ainsi, dans un problème de classification, où les classes n'ont pas été apprises, on peut supposer qu'en conservant les premières couches on extraîtra des caractéristiques communes des images (bords, colorimétrie,...) et qu'en changeant les dernières couches (information sémantique et haut niveau et étage de classification), c'est-à-dire en réapprenant les connexions, on spécifiera le nouveau réseau pour la nouvelle tâche de classification. Cette approche rentre dans le cadre des méthodes de *transfer learning* et de *fine tuning*, cas particulier d'adaptation de domaine :

- Les méthodes de transfert prennent un réseau déjà entraîné, enlèvent la dernière couche complètement connectée et traitent le réseau restant comme un extraiteur de caractéristiques. Un nouveau classifieur, la dernière couche, est alors entraîné sur le nouveau problème ([Fig. 13](#)).



**Fig. 13** Illustration de la technique de transfer Learning. Le réseau appris à classer des images de chat est amputé de sa partie classification. Un nouveau classifieur est mis au bout du réseau, dont les poids sont entraînés sur la nouvelle tâche de classification.

- Les méthodes de fine tuning ré-entraînent tout ou partie d'un réseau (suivant les données disponibles) et conservent les poids non ré-entraînés.

Plusieurs facteurs influent sur le choix de la méthode à utiliser : la taille des données d'apprentissage du nouveau problème et la ressemblance du nouveau jeu de données avec celui qui a servi à entraîner le réseau initial :

- Pour un jeu de données similaire de petite taille, on utilise du transfer learning, avec un classifieur utilisé sur les caractéristiques calculées sur les dernières couches du réseau initial.

- Pour un jeu de données de petite taille et un problème différent, on utilise du transfer learning, avec un classifieur utilisé sur les caractéristiques calculées sur les premières couches du réseau initial.
- Pour un jeu de données, similaire ou non, de grande taille, on utilise le fine tuning.

Notons qu'il est toujours possible d'augmenter la taille du jeu de données par des techniques d'augmentation de données.

Dans le cas où un réseau ad hoc doit être construit et où une base d'apprentissage suffisante est disponible, l'entraînement par optimisation reste possible mais peut nécessiter des ressources de calcul importantes. De plus, il a été montré que :

- le transfer Learning ou le fine tuning permettaient souvent d'aboutir à de meilleures performances que l'entraînement depuis un réseau initial aléatoire (on se sert des poids du réseau pré entraîné comme initialisation, plutôt qu'une initialisation type Xavier).
- le fine tuning améliorait la capacité de généralisation du réseau.

## Visualisation du mécanisme des réseaux convolutifs

Le mécanisme interne des réseaux convolutifs est mal compris et l'analyse des raisons qui font que leur puissance de prédiction est importante n'est pas aisée. S'il est toujours possible de rétroprojeter les activations depuis la première couche de convolution, les couches d'agrégation et de rectification empêchent de comprendre le fonctionnement des couches suivantes, ce qui peut être gênant dans la construction et l'amélioration de ces réseaux.

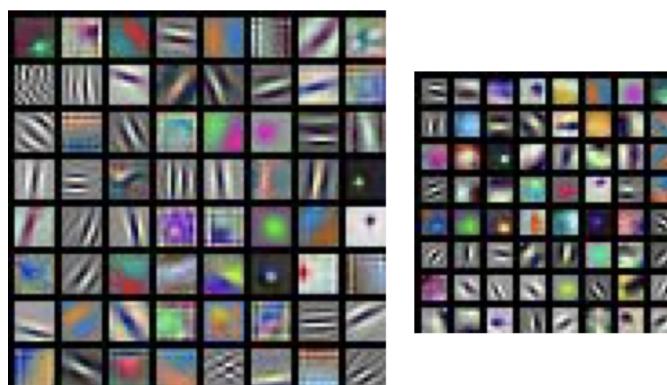
Les méthodes de visualisation du fonctionnement des réseaux convolutifs peuvent être rangées en trois catégories, décrites brièvement dans les paragraphes suivants.

### Méthodes de visualisation de base

Les méthodes les plus simples consistent à visualiser les activations lors du passage d'une image dans le réseau. Pour des activations type ReLU, ces activations sont ininterprétables au début de l'entraînement, mais à mesure que ce dernier progresse, les cartes d'activation ( $\{\mathbf{Y}_i^l\}$ ) deviennent localisées et éparses.

Il est également possible de visualiser les filtres des différentes couches de convolution (Fig. 14). Les filtres des premières couches agissent comme des détecteurs de bords et coins et, à mesure que l'on s'enfonce dans le réseau, les filtres capturent des concepts haut niveau comme des objets ou encore des visages.

Citons encore d'autres méthodes qui proposent de visualiser les dernières couches (les couches complètement connectées) de grande dimension (par exemple 4096 pour AlexNet) via une méthode de réduction de dimension.



**Fig. 14** Visualisation des filtres de la première couche d'AlexNet (à gauche, 64 filtres

11 $\times$ 11) et de ResNet-18 (à droite, 64 filtres 7 $\times$ 7).

## Méthodes fondées sur les activations

Plusieurs stratégies peuvent être adoptées pour sonder le fonctionnement d'un réseau convolutif, en utilisant les informations portées par les cartes  $\{\mathbf{Y}_i^l\}$ , parmi lesquelles :

- Utiliser des couches de convolution transposée (improprement appelées parfois couches de déconvolution), ajoutées à chaque couche de convolution du réseau. Étant données les cartes d'entrée de la couche  $\{\mathbf{Y}_i^l\}$ , les cartes de sortie  $\{\mathbf{Y}_{i-1}^{l-1}\}$  sont envoyées dans la couche de convolution transposée correspondante au niveau  $\{l\}$ . Cette dernière reconstruit les  $\{\mathbf{Y}_{i-1}^{l-1}\}$  qui ont permis le calcul des activations de la couche  $\{l\}$ . Le processus est alors itéré jusqu'à atteindre la couche d'entrée  $\{l=1\}$ , les activations de la couche  $\{l\}$  étant alors rétroprojétées dans le plan image [ZE13]. La présence de couches d'agrégation et de rectification rend ce processus non inversible (par exemple, une couche d'agrégation maximum nécessite de connaître à quelles positions de l'image  $\{\mathbf{Y}_i^l\}$  sont situés les maxima retenus).
- Faire passer un grand nombre d'images dans le réseau et, pour un neurone particulier, conserver celle qui a le plus activé ce neurone. Il est alors possible de visualiser les images pour comprendre ce à quoi le neurone s'intéresse dans son champ réceptif ([Fig. 15](#)).



**Fig. 15** Champ réceptif de quelques neurones de la dernière couche d'agrégation du

réseau AlexNet, superposées aux images ayant le plus fortement activé ces

neurones. Le champ est encadré en blanc, et la valeur d'activation correspondante

est reportée en haut. On voit par exemple que certains neurones sont très sensibles

aux textes, d'autres aux réflexions spéculaires, ou encore aux hauts du corps

(source :[\[4\]](#))

- Cacher (par un rectangle noir par exemple) différentes parties de l'image d'entrée qui est d'une certaine classe (disons un chien) et observer la sortie du réseau (la probabilité de la classe de l'image d'entrée). En représentant les valeurs de probabilité de la classe d'intérêt comme une fonction de la position du rectangle occultant, il est possible de voir si le réseau s'intéresse effectivement aux parties de l'image spécifiques de la classe, ou à des autres zones (le fond par exemple) ([Fig. 16](#)).



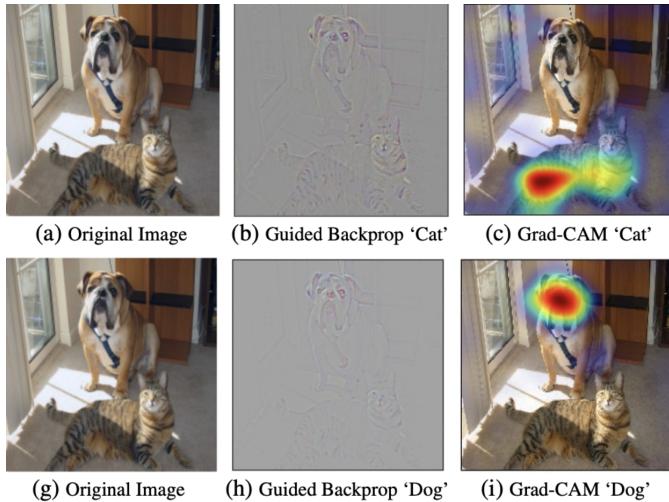
**Fig. 16** Occlusion d'une image (à gauche). Le rectangle noir est déplacé dans l'image et pour chaque position la probabilité de la classe de l'image (ici un loulou de Poméranie) est enregistrée. Ces probabilités sont ensuite représentées sous forme d'une carte 2D (à droite). La probabilité de la classe s'effondre lorsque le rectangle couvre une partie de la face du chien. Cela suggère que cette face est grandement responsable de la forte probabilité de classement de l'image comme un loulou. A l'inverse, l'occlusion du fond n'affecte pas la forte valeur de probabilité de la classe (source :[\[5\]](#))

## Méthodes fondées sur le gradient

Pour comprendre quelle(s) partie(s) de l'image est (sont) utilisée(s) par le réseau pour effectuer une prédiction, il est possible de calculer des cartes de saillance (saliency maps). L'idée est relativement simple : calculer le gradient de la classe de sortie par rapport à l'image d'entrée. Cela indique à quel point une petite variation dans l'image induit un changement de prédiction. En visualisant les gradients, on observe alors par exemple leurs fortes valeurs, indiquant qu'une petite variation du pixel correspondant augmente la valeur de sortie.

Il est également possible d'utiliser le gradient par rapport à la dernière couche de convolution (approche Grad-CAM), ce qui permet de récupérer des informations de localisation spatiale des régions importantes pour la prédiction ([Fig. 17](#) droite).

Plus généralement, en choisissant un neurone intermédiaire du réseau (d'une couche de convolution), la méthode de rétropropagation guidée calcule le gradient de sa valeur par rapport aux pixels de l'image d'entrée, ce qui permet de souligner les parties de l'image auxquelles ce neurone répond ([Fig. 17](#) milieu).



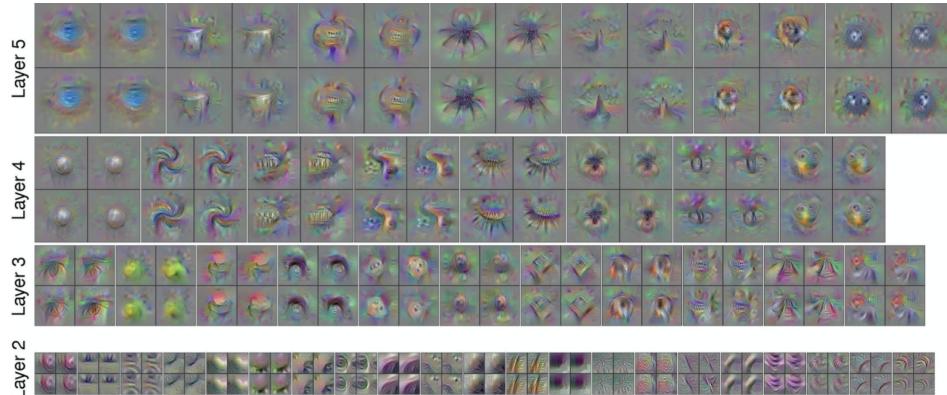
**Fig. 17** Approches par gradient de visualisation du fonctionnement d'un réseau

convolutif. Comparaison de la méthode de rétropropagation guidée et de Grad-CAM

(source :[\[6\]](#))

Ces gradients peuvent également être utilisés dans la méthode de montée de gradient (gradient Ascent), dont l'objectif est de générer une image qui active de manière maximale un neurone donné du réseau. Le principe est d'itérativement passer l'image d'entrée  $\mathbf{I}$  dans le réseau pour obtenir les valeurs des sorties  $\mathbf{Y}_i$ , de rétropropager pour obtenir le gradient d'un neurone par rapport aux pixels de  $\mathbf{I}$  et d'opérer une petite modification de ces pixels.

Outre son aspect informatif sur la structure interne du réseau étudié (visualisation des cartes \(\{Y\_i\}\) intermédiaires), cette méthode produit des images parfois très artistiques ((Fig. 18)).



**Fig. 18** Visualisation des 4 premières couches de convolution d'un réseau convolutif

par montée de gradient (source : [7])

- [1] V. Nair and G. Hinton. Rectified linear units improve restricted boltzmann machines. In Johannes Fürnkranz and Thorsten Joachims, editors, *ICML*, 807–814. Omnipress, 2010. URL: <http://dblp.uni-trier.de/db/conf/icml/icml2010.html#NairH10>.
- [2] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey J. Gordon, David B. Dunson, and Miroslav Dudík, editors, *AISTATS*, volume 15 of *JMLR Proceedings*, 315–323. JMLR.org, 2011. URL: <http://dblp.uni-trier.de/db/journals/jmlr/jmlrp15.html#GlorotBB11>.
- [3] Sergey Ioffe and Christian Szegedy. Batch normalization: accelerating deep network training by reducing internal covariate shift. In Francis R. Bach and David M. Blei, editors, *ICML*, volume 37 of *JMLR Workshop and Conference Proceedings*, 448–456. JMLR.org, 2015. URL: <http://dblp.uni-trier.de/db/conf/icml/2015.html#IoffeS15>.
- [4] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 00, 580–587. June 2014. URL: <https://ieeexplore.ieee.org/abstract/document/6909475/>, doi:10.1109/CVPR.2014.81.
- [5] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. *CoRR*, 2013. URL: <http://dblp.uni-trier.de/db/journals/corr/corr1311.html#ZeilerF13>.
- [6] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: visual explanations from deep networks via gradient-based localization. In *ICCV*, 618–626. IEEE Computer Society, 2017. URL: <http://dblp.uni-trier.de/db/conf/iccv/iccv2017.html#SelvarajuCDVPB17>.
- [7] Jason Yosinski, Jeff Clune, Anh Mai Nguyen, Thomas J. Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. *CoRR*, 2015. URL: <http://dblp.uni-trier.de/db/journals/corr/corr1506.html#YosinskiCNFL15>.