PRINCIPES FONDAMENTAUX DE L’APPRENTISSAGE PROFOND

Pascal Maryniak

Principes fondamentaux de l'apprentissage profond

Table des matières

1 Introduction

Ce document est destiné à tous ceux qui désirent explorer et comprendre les concepts de l'apprentissage profond en un temps minimum. Il existe en effet énormément d'ouvrages, de documents en ligne et de tutoriels qui les expliquent, pour la plupart à l'aide de listings informatiques en Python qui est le langage utilisé pour leur implémentation. Apprendre le fonctionnement des réseaux de neurones profonds est donc d'un abord difficile pour les ingénieurs qui n'ont pas l'opportunité d'acquérir les bases nécessaires sur cet outil informatique, même s'il est puissant et facile à apprendre.

En fait, la majorité des ingénieurs qui n'exercent pas un métier dans ce domaine ignore que le fonctionnement des réseaux de neurones repose fondamentalement et simplement sur un algorithme mathématique de descente de gradient qui est accessible à tout ingénieur, qu'il soit débutant ou expérimenté.

Le but de ce document est donc de présenter de manière abordable les principes de fonctionnement des réseaux de neurones profonds en faisant abstraction du code et des aspects informatiques.

Le chapitre 2 énonce les définitions et principes de base de l'apprentissage profond.

Le chapitre 3 retrace succinctement l'évolution des réseaux de neurones et expose leur architecture et leur fonctionnement.

Le chapitre 4 décrit les méthodes d'entraînement des réseaux de neurones profonds, qui reposent fondamentalement sur l'algorithme de rétropropagation du gradient.

Le chapitre 5 est une introduction aux réseaux de neurones convolutifs, qui ont révolutionné le domaine de l'intelligence artificielle et la vision par ordinateur depuis une dizaine d'années.

Enfin, ce document fera très prochainement l'objet de versions ultérieures pour aborder l'IA générative et notamment les modèles de langage comme ChatGPT.

2 Qu'est-ce que l'apprentissage profond ?

2.1 De l'IA symbolique à l'apprentissage automatique (machine learning)

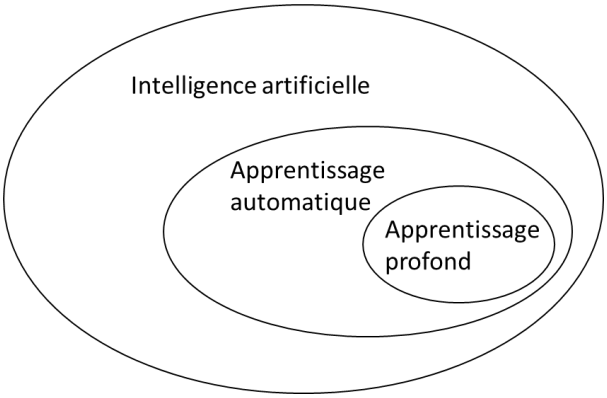
L'approche symbolique a dominé l'IA depuis les années 1950 jusqu'à la fin des années 1980. Ce paradigme repose sur la conception d'un ensemble suffisamment large de règles explicites faisant l'objet d'une traduction dans des programmes informatiques pour résoudre des problèmes relevant de l'intelligence humaine. Cette méthode, qui permettait pourtant de résoudre des problèmes logiques bien définis, s'est avérée incapable d'effectuer des tâches d'une nature plus complexe ou plus floue telles que la reconnaissance d'images, le traitement de la parole, ou la traduction linguistique avec des performances s'approchant de celles d'un humain. Une nouvelle approche est venue détrôner l'IA symbolique dans ces domaines, c'est l'apprentissage automatique (machine learning).

2.2 L'apprentissage profond (deep learning)

Les progrès considérables effectués en intelligence artificielle au cours de la dernière décennie résultent en grande partie de la contribution des réseaux de neurones profonds au techniques d'apprentissage machine. L'adjectif « profond » est relatif à la solution consistant à empiler de

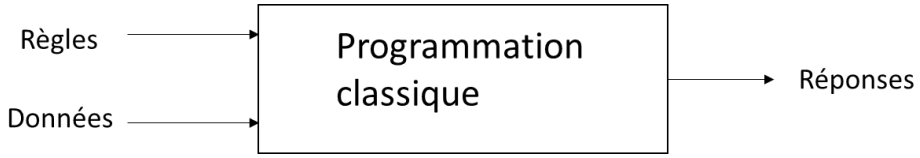
nombreuses couches de neurones afin d'automatiser des tâches de plus en plus compliquées. L'apprentissage profond, ou « deep learning » consiste en l'application de techniques d'apprentissage machine en faisant appel à des réseaux de neurones profonds. L'apprentissage profond forme donc un sous-domaine de l'apprentissage automatique (machine learning).

Aujourd'hui, le deep learning (apprentissage profond) surpasse ainsi largement les systèmes de traduction automatique à base de règles développés à partir des années 1970 en offrant des performances sensationnelles à un niveau quasi humain. Les résultats obtenus en matière de reconnaissance d'images sont également très spectaculaires, pouvant même surpasser les capacités humaines.



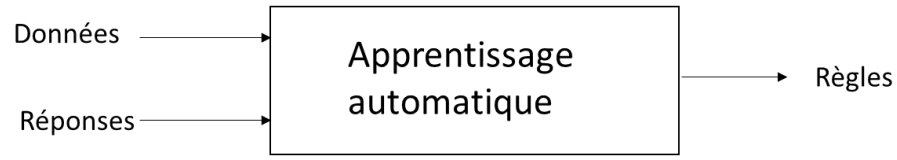
2.3 Qu'est-ce que l'apprentissage automatique ?

En programmation classique, on entre des données relatives à la question posée, ainsi qu'un programme qui sera appliqué sur ces données d'entrée pour établir les réponses en sortie.



En apprentissage automatique, on dispose au préalable d'un ensemble de données pour lesquelles les réponses à la question sont connues à l'avance, sans aucune connaissance des règles permettant de déduire les réponses à partir des données d'entrée. Par exemple, si les données d'entrée sont des images de chiens ou de chats et que le but est de classer les images dans une de ces deux catégories, la machine apprenante reçoit en entrée un échantillon d'images avec une étiquette associée à chacune d'entre elles portant le nom de l'animal représenté dans l'image, qui est la sortie attendue, « chien » ou « chat ».

L'apprentissage automatique va consister à entraîner le système à partir des exemples qui lui sont présentés et des réponses attendues afin qu'il acquière la capacité à prédire la sortie sur des nouvelles données dont il ne connait alors pas la réponse.

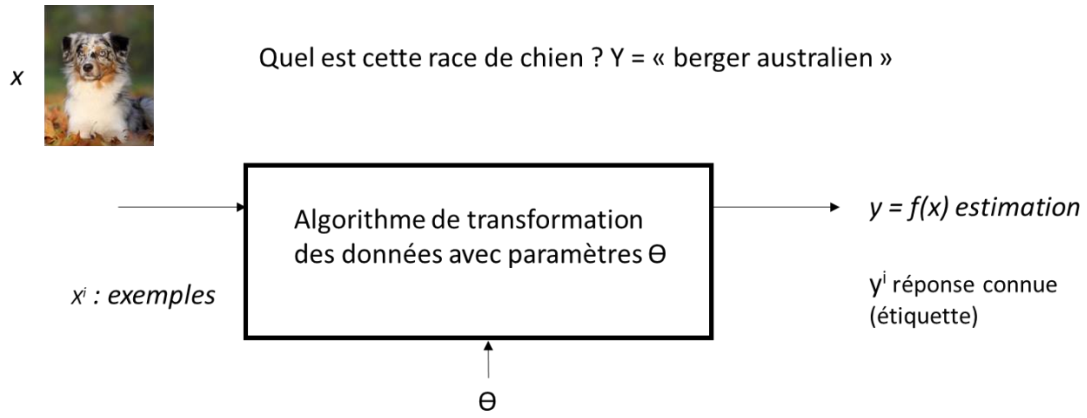


L'apprentissage automatique passe donc par une phase d'entraînement qui nécessite de disposer d'un grand nombre d'exemples avec des réponses connues à l'avance. Par exemple, pour répondre à la question « est-ce un chien ou un chat ? », il sera nécessaire de disposer de quelques centaines d'exemples d'images de chien ou de chat, chacune d'entre elle pouvant par exemple comporter un million de pixels. Si l'on a affaire à une application de reconnaissance de la parole ou de traitement du langage naturel, Il faudra plusieurs millions d'échantillons pour le son ou des millions de lettres dans un livre.

2.4 L'algorithme général d'apprentissage

La machine dispose d'un ensemble d'exemples (x^i), i étant un indice sur les exemples considérés, et x^i un vecteur dont les composantes représentent par exemple dans le cas d'une image les valeurs de l'intensité de ses pixels. La machine possède également en entrée les étiquettes (y^i) des réponses attendues à la question posée, par exemple « quelle est la race du chien dans cette image ?».

Le système applique au le vecteur d'entrée x un algorithme de transformation des données pour fournir une estimation y = f(x) de la réponse à la question posée. L'idée de base est que cet algorithme est paramétré avec potentiellement un grand nombre de paramètres qui sont par exemple les poids d'un réseau de neurones.



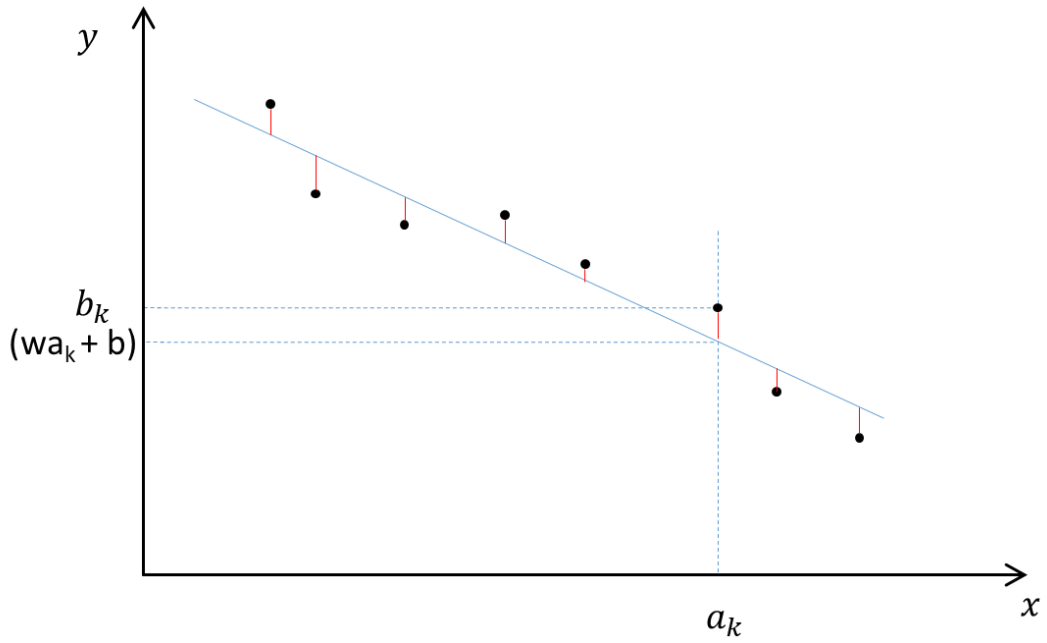
L'apprentissage consiste à optimiser les paramètres de l'algorithme de telle sorte que sur les exemples (x^i), la réponse qu'il fournit soit la plus proche possible des valeurs attendues (y^i). Les paramètres seront ainsi adaptés pour que l'algorithme fournisse quasiment la bonne réponse f(x^i) ~ y^i sur les exemples d'entraînement.

L'enjeu d'un tel algorithme réside bien évidemment dans sa capacité de généraliser, c'est-à-dire de prédire la réponse juste y pour un nouvel x qui n'est pas dans l'ensemble d'entraînement.

2.5 Un exemple simple et représentatif d'un algorithme d'apprentissage : la régression linéaire

Il est très intéressant au plan pédagogique de revenir à la présentation de cette méthode mise à contribution dans un très grand nombre d'activités pour prédire une variable à partir d'un ensemble de données connues. Dans son principe de base connu par tout ingénieur, cette méthode permet, à partir d'un nuage de points, de tracer une droite qui en soit la plus proche possible. L'étude de sa formulation mathématique et de sa démarche résolution constituent en fait une véritable introduction aux principes du machine learning et du fonctionnement des réseaux de neurone.

La régression linéaire permet de prédire une variable y en fonction d'une variable x, en partant de la connaissance de valeurs de y relevées ou mesurées sur le terrain correspondant à un échantillon de valeurs de x. Par exemple, pour prédire la température y en fonction de l'altitude x dans un environnement montagneux, on effectue divers relevés de température b_k en différents points d'altitude a_k . Le nuage de points est formé de l'ensemble des p points de coordonnées (a_1,b_1), (a_2,b_2),…(a_p,b_p) dans le plan xy, comme cela est représenté sur la figure ci-dessous :



La recherche de la droite qui passe au plus près des points de ce nuage est équivalente à une approche de machine learning visant à prédire la température y en fonction de x, à partir de l'ensemble des données (a_k,b_k) :

- Considérant l'équation générale d'une droite $y = w \times x + b$, on se trouve en présence d'un algorithme de prédiction de y en fonction de l'entrée x paramétré selon les deux paramètres w et b
- l'erreur commise par l'algorithme sur la température pour l'altitude a_k est la différence entre la valeur prédite par l'algorithme et la valeur relevée : $(w \times a_k + b) - b_k$, matérialisée par

les traits verticaux rouges sur la figure ci-dessus et fournit une mesure de la distance entre la droite d'équation $y = w \times x + b$ et le point (a_k,b_k)

- l'erreur globale, notée L sur l'ensemble d'entraînement (a_k,b_k) est la somme des carrés des erreurs sur chacune des altitudes a_k : $L = \sum_k ((w \times a_k + b) - b_k)^2$
- l'apprentissage consiste à rechercher les valeurs de w et b qui rendent minimum l'erreur L : c'est précisément l'algorithme de la régression linéaire par la méthode des moindres carrés.

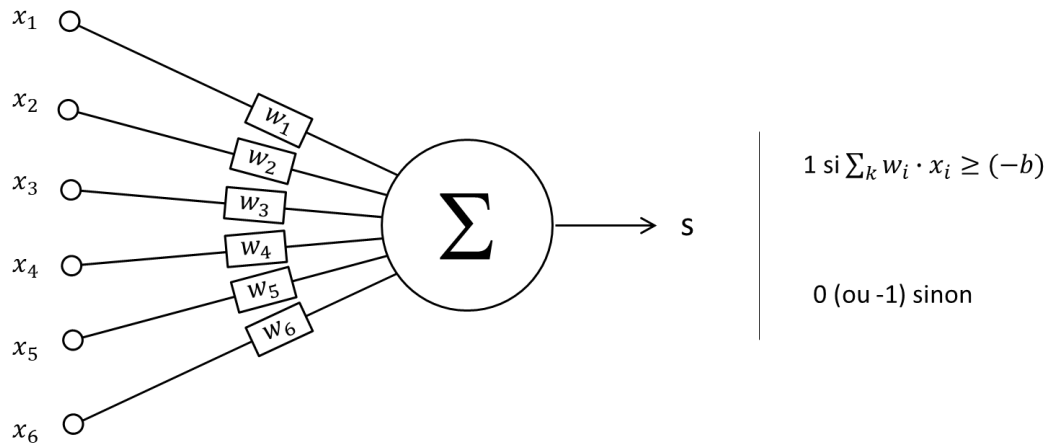
La méthode classique pour déterminer les valeurs de w et b est de calculer les dérivées partielles de la fonction d'erreur L(w,b) en fonction des variables w et b, les valeurs a_k et b_k étant fixées. Les valeurs w et b recherchées doivent annuler ces dérivées partielles ce qui conduit à un système d'équations linéaires à deux inconnues dont la résolution est très simple.

L'apprentissage profond reprend ce même principe de recherche d'un minimum d'une fonction d'erreur sur un ensemble de paramètres qui sont dans le cas d'un réseau de neurones les différents poids du réseau. En apprentissage profond, le nombre de paramètres peut être considérable et la recherche d'un tel minimum est impossible par le calcul analytique. Dans les réseaux de neurones profonds, la recherche d'un minimum s'appuie sur un algorithme de descente du gradient qui sera expliqué dans la suite.

3 Structure et fonctionnement des réseaux de neurones

3.1 Début des réseaux de neurones

En 1943, Warren McCulloch, et Walter Pitts sont les premiers à proposer un modèle très simplifié d'un neurone biologique. Les dendrites sont représentées par les entrées du modèle, et la sortie correspond au point de départ de l'axone (cône d'émergence). Le neurone effectue une somme pondérée des entrées reçues et applique un seuil : si la valeur de la somme est supérieure au seuil (-b), le neurone est activé et la sortie vaut 1, et si cette somme est inférieure au seuil, le neurone renvoie une sortie nulle.



En 1957, le psychologue Frank Rosenblatt reprend cette idée pour concevoir une machine apprenante, appelée le Perceptron. Cette machine consistait dans sa forme la plus simple en un neurone unique,

apprenant en modifiant ses poids afin de reconnaître des formes ou des lettres. Elle reste encore aujourd'hui un modèle de référence de l'apprentissage machine.

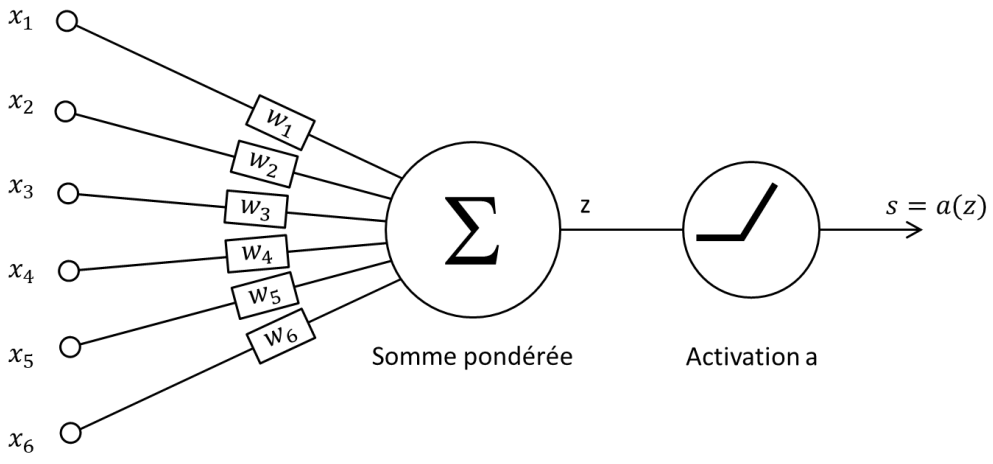
3.2 Le neurone informatique moderne

D'un point de vue formel, le modèle de neurone utilisé dans les réseaux de neurones modernes reste assez proche de celui de McCulloch, et Pitts. Les entrées reçues sont un ensemble de n valeurs numériques $(x_1, \dots, x_n)$ regroupées dans un vecteur X de dimension n qui correspondent aux données d'entrée du problème à résoudre. Chaque entrée étant plus ou moins importante pour le neurone, ce dernier lui applique un poids appelé coefficient synaptique selon l'analogie avec le neurone cérébral. Le neurone effectue la somme pondérée de ces entrées selon ces n coefficients synaptiques notés $w_1,\dots,w_n$ et lui ajoute en général un biais constant b. Différents types de fonctions d'activation sont ensuite appliqués à cette somme, dont l'étude sera présentée dans la suite. Pour le cas particulier du neurone de McCulloch, et Pitts qui est un neurone binaire, la fonction d'activation est la fonction de Heaviside qui vaut 1 si la sortie est positive et 0 sinon. La fonction d'activation la plus célèbre est la fonction ReLU (Rectified Linear Unit) qui prend pour une entrée x le maximum des deux valeurs 0 et x.

La sortie s du neurone s'exprime donc de la manière suivante :

$s = a(\sum_i w_i \times x_i + b)$

où a est la fonction d'activation appliquée. En général, une fonction dérivable est choisie pour une raison que nous verrons dans la suite.



3.3 Le perceptron et ses limites

Le perceptron, fabriqué pour reconnaitre des formes et des lettres, présentait encore des capacités restreintes, s'avérant incapable de distinguer certains types de formes. Cette limitation vient du fait qu'il repose sur un modèle linéaire. Si la tâche consiste à distinguer un C d'un D, l'entrée de la machine va être un vecteur X de dimension n dont les coordonnées x_i seront les intensités des pixels de l'image. Dans sa forme la plus simple consistant en un seul neurone, le perceptron calcule une somme pondérée $\sum_i w_i \times x_i + b$ et produit une sortie binaire valant 1 si la somme est positive (ou ce qui revient au même, supérieure à un seuil -b) et 0 sinon. Le perceptron sépare ainsi l'espace à n dimensions des entrées X en deux moitiés dont la frontière est l'hyperplan à n-1 dimensions d'équation :

$\sum_i w_i \times x_i + b = 0.$

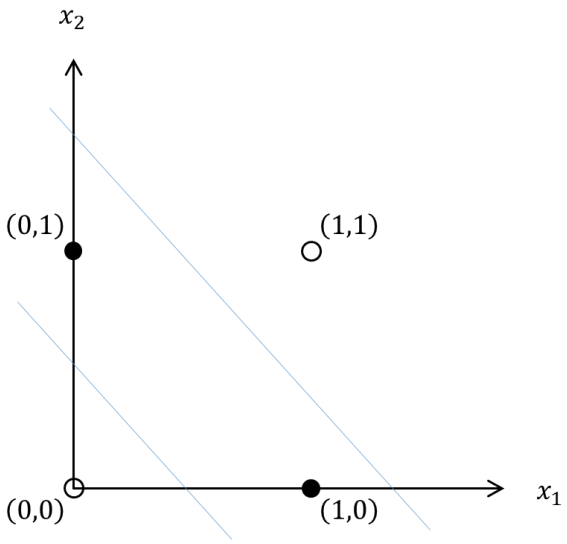
Avec un perceptron à trois entrées, l'espace des entrées possède trois dimensions et la frontière de séparation est un plan. Dans le cas d'un perceptron à deux entrées $x_1$et x_2, la sortie est 1 si la somme pondérée $w_1 \times x_1 + w_2 \times x_2$ + b est positive et 0 sinon. La frontière de séparation vérifie donc l'équation $w_1 \times x_1 + w_2 \times x_2$ + b = 0, qui, remise sous la forme :

$$x_2 = -\frac{1}{w_2}(w_1 \times x_1 + b)$$

est bien l'équation d'une droite.

Pour que le perceptron puisse classer les points d'entrées (x_1, x_2) dans deux catégories, il est nécessaire que les deux ensembles de points représentés par chacune de ces catégories soient situés de part et d'autre d'une droite, les points appartenant à une même catégorie étant du même côté de ladite droite. D'un côté de la droite, on doit avoir tous les points qui activent la sortie du perceptron, tandis que de l'autre côté, la sortie doit être nulle. Le perceptron ne peut donc fonctionner de manière efficace que si les exemples d'apprentissage sont linéairement séparables, c'est-à-dire pouvant être séparés par une droite. Or, il est fréquent de rencontrer des situations dans lesquelles les points ne sont pas linéairement séparables, notamment et surtout en reconnaissance de formes, où les entrées sont par exemples constituées de la valeur des pixels d'une image avec des tailles importantes (rien qu'une image de 5 x 5 pixels est représentée par un vecteur d'entrée de dimension 25).

Si l'on prend l'exemple des fonctions logiques (« et », « ou », etc.) à deux entrées binaires, le nombre de fonctions est de 2^4 puisqu'il y a quatre entrées possibles (2 x 2) et à chaque fois deux possibilités pour chacune de ces 4 entrées. Deux fonctions sur ces 16 ne sont pas séparables linéairement, dont le OU exclusif. Pour s'en convaincre, il suffit en effet de représenter les quatre entrées possibles (0,0) (0,1) (1,0) (1,1) sur un plan dans la figure ci-dessous et de se rendre compte qu'il est impossible de tracer une droite qui sépare les deux points (0,0) et (1,1) des deux points (0,1) et (1,0). La fonction OU exclusif vaut en effet 0 pour les deux entrées (0,0) et (1,1) et 1 pour les deux autres entrées (0,1) et (1,0).



Plus la dimension de l'espace des entrées est élevée, plus le risque est grand de rencontrer des cas où les catégories ne sont pas linéairement séparables.

3.4 Du perceptron aux réseaux de neurones profonds

Confrontée aux limites du perceptron, la communauté scientifique se tourne alors vers la solution consistant à empiler plusieurs couches successives de neurones pour résoudre des tâches plus difficiles. Cependant, aucune solution n'est d'abord trouvée pour entraîner de tels réseaux de bout en bout, si bien que cette voie de recherche est abandonnée à la fin des années 1960. Un tel algorithme a émergé au début des années 1980, connu sous le nom de rétro-propagation du gradient. Il repose d'une part sur la méthode de descente du gradient développée et publiée par le mathématicien Augustin-Louis Cauchy en 1847 pour optimiser les poids du réseau, et d'autre part, pour le calcul des gradients, sur les règles de composition en calcul différentiel qui sont issues des travaux antérieurs de Leibniz en 1676. La méthode est publiée en octobre 1986 par trois auteurs, David E. Rumelhart, Geoffrey E. Hilton, et Ronald J. Wiliams, sous le titre ***«Learning representations by back-propagating errors »***

Malheureusement, il faut encore attendre les années 2010 pour que les réseaux profonds fassent leur apparition, grâce aux processeurs graphiques GPU (Graphic Processing Units) et la disponibilité de très grandes quantités de données pour les entraîner.

3.5 Les règles de construction des réseaux de neurones profonds

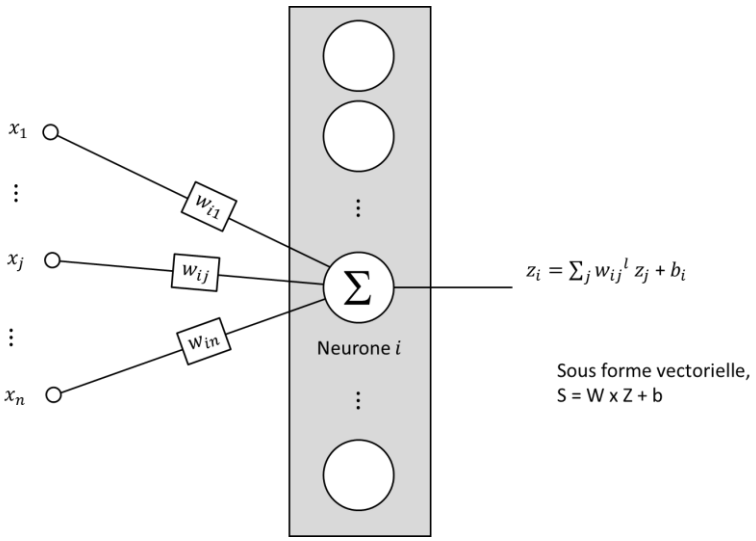
3.5.1 L'assemblage en couches

Dans un réseau comportant plusieurs couches empilées les unes sur les autres, chaque couche peut comporter un nombre quelconque de neurones. Considérons le cas général d'un modèle de dimension quelconque avec des données en entrée sous la forme d'un vecteur de nombres $(x_1, \ldots, x_n)$. Si la première couche du réseau comporte un nombre p de neurones, chacune des entrées x_j est connectée

à chaque neurone de cette première couche par l'intermédiaire d'un poids w_{ij}, i étant l'indice du neurone concerné dans la couche (voir figure ci-dessous). La sortie de ce neurone d'indice i vaut donc la somme pondérée des entrées $(x_1, \dots, x_n)$: $\sum_j w_{ij} \times x_j + b_i$ à laquelle on a ajouté le biais du neurone b_i.

Les poids du réseau sont ainsi représentés par une matrice W constituée des poids w_{ij}, et on peut aisément vérifier que si X symbolise le vecteur d'entrée $(x_1, \dots, x_n)$, la sortie d'un tel réseau à une couche est l'addition du produit matriciel W× X et du vecteur de biais B, qui est un vecteur de dimension p correspondant au nombre de neurones de la couche :

S = W × X + B



Les couches sont empilées de telle sorte que les sorties d'une couche deviennent les entrées de la couche suivante, la sortie du réseau étant celle de la dernière couche. La taille des sorties d'une couche est fixée par le nombre de neurones de la couche. Une couche est dite dense si les neurones qu'elle comporte sont entièrement connectées aux sorties de la couche inférieure. Nous verrons que ce n'est pas toujours le cas, notamment pour les réseaux de neurones convolutifs, qui font l'objet du chapitre 5.

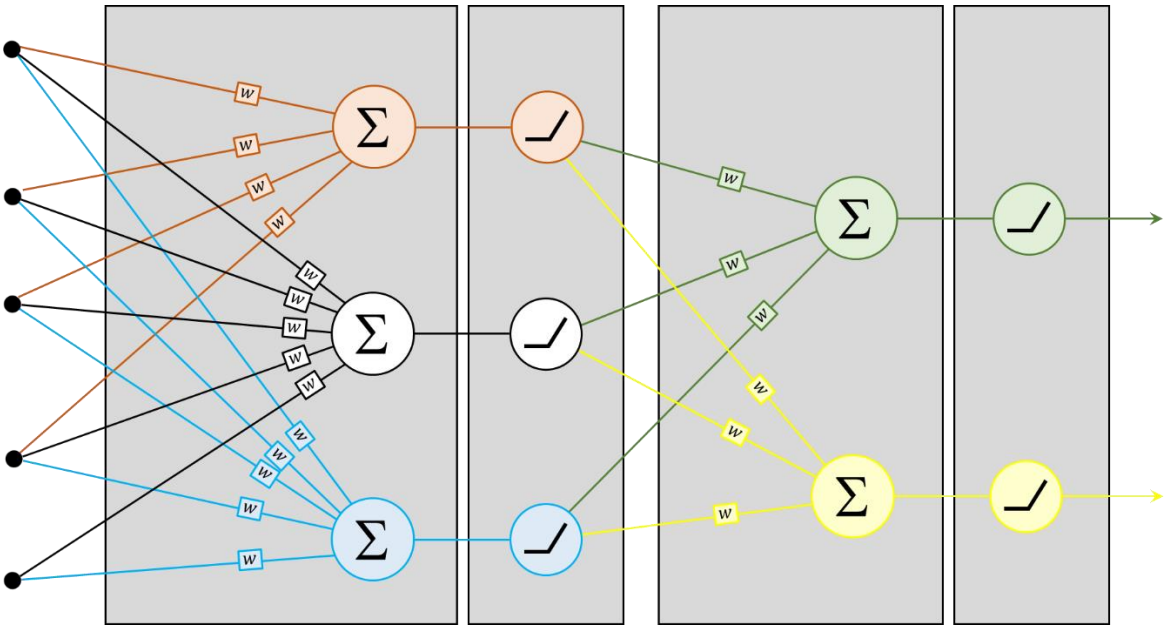
Les poids du réseau, qui sont les paramètres des transformations à appliquer, sont regroupés dans une suite de matrices W_{ij}^loù W_{ij}^lest la matrice de poids de la couche l du réseau. La dernière couche du réseau est appelée couche de sortie, et les différentes couches intermédiaires avant cette couche de sortie sont dites couches cachées.

3.5.2 Les fonctions d'activation des couches intermédiaires

Un problème subsiste encore, en dépit de l'introduction des réseaux profonds, et de la découverte d'une méthode permettant de les entraîner. Il a été montré plus haut que les transformations linéaires qui fondent le fonctionnement du perceptron sont insuffisantes pour représenter des fonctions complexes, un neurone élémentaire étant incapable de modéliser une fonction comme le OU exclusif. Or, si l'on réalise un empilement de couches linéaires, la combinaison des fonctions qui en résulte sera linéaire et on retomberait alors sur le même problème, sans avoir apporté aucune amélioration de capacité du réseau. La solution pour sortir de cette impasse est d'introduire des non-linéarités en intercalant des fonctions non linéaires entre les couches de neurone. Ces fonctions qui ont été introduites au paragraphe 3.2 sont dites fonctions d'activation, la plus célèbre étant la fonction ReLU

(Rectified Linear Unit) qui prend pour une entrée x le maximum des deux valeurs 0 et x. Il en existe quelques autres, aux noms tous aussi bizarres tels que PRelU, eLU ou encore SeLU.

Le modèle général d'empilement des couches avec insertion de fonctions d'activation est schématisé sur la figure suivante :

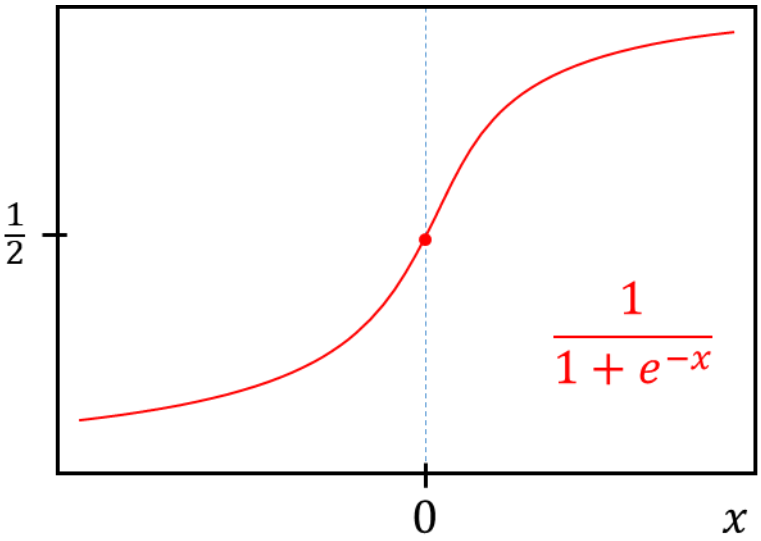


3.5.3 Structure et fonctions d'activation de la dernière couche

La dernière couche du réseau est adaptée en fonction de la tâche cible à réaliser, qui peut être de quatre natures différentes :

- Une classification binaire, appelée aussi classification en deux classes, dans laquelle chaque exemple d'entrée doit être classé entre deux catégories exclusives ; un exemple en est la distinction entre chien ou chat dans des images comportant exclusivement un de ces animaux,
- Une classification en plusieurs classes, consistant à classer chaque entrée dans une catégorie unique parmi un ensemble donné de plus de deux catégories ; un exemple en est la classification d'images de chiffres manuscrits dans dix catégories (de 0 à 9)
- La régression scalaire, ou la cible est une valeur continue
- La régression vectorielle, ou la cible est un ensemble de valeurs continues.

Pour une tâche de classification binaire, comme par exemple le classement de critiques de film en critiques positives ou négatives, la couche finale comportera une sortie unique, donc une couche à un seul neurone, à laquelle sera appliquée une fonction d'activation bien particulière afin d'obtenir une valeur de sortie comprise entre 0 et 1 représentant la probabilité que l'échantillon soit classé dans une des deux catégorie (par exemple la probabilité que la critique de film soit positive). Cette fonction est la fonction sigmoide qui peut être vue comme une approximation continue et dérivable de la fonction de Heaviside qui vaut 1 pour x positif ou est nulle pour x négatif. La fonction sigmoide (voir graphe ci-dessous) « écrase » les valeurs dans l'intervalle [0,1] si bien que sa valeur peut être interprétée comme une probabilité.



La fonction sigmoïde est notée σ et elle est définie par $\sigma(x) = \frac{1}{1+e^{-x}}$

Elle est dérivable et sa dérivée est égale à $\sigma(1-\sigma)$

Pour un problème de classification à plusieurs classes, si p est ce nombre de classes, la dernière couche sera une couche dense constituée de p neurones entièrement connectés qui retournera un vecteur de sortie de dimension p contenant les scores de probabilité que l'entrée appartienne aux différentes classes. Une fonction mathématique particulière nommée softmax sera utilisée en tant que fonction d'activation pour obtenir l'équivalent de telles probabilités pour chaque classe avec des valeurs comprises entre 0 et 1 et une somme égale à 1.

Avec une sortie $(y_1, \dots, y_{p)}$ on remplace chaque y_i par l'exponentielle e^{y_i} pour obtenir un nombre positif, puis on normalise le résultat en le divisant par la somme de ces exponentielles. Ainsi, softmax $(y_1, \dots, y_{p)}$ est le vecteur $(z_1, \dots, z_{p)}$ où $z_i = \frac{e^{y_i}}{\sum_k e^{y_k}}$

Pour un problème de régression scalaire, la dernière couche est une couche à une seule sortie, sans aucune fonction d'activation spécifique.

Pour un problème de régression vectorielle, la sortie est multidimensionnelle et fournit directement les valeurs cibles, sans fonction d'activation spécifique non plus, afin que le réseau soit capable d'apprendre des valeurs quelconques non bornées.

3.5.4 Les fonctions de perte à utiliser pour l'apprentissage

Pour un problème de régression vectorielle, la fonction de perte à minimiser est l'erreur quadratique moyenne : si Y'^i est la sortie du réseau pour l'entrée X^i (i étant l'indice sur l''ensemble des données d'entrainement) et Y^i l'étiquette associée à cette entrée qui correspond à la valeur attendue pour cette entrée, la fonction de perte est l'erreur quadratique moyenne :

$$L = \frac{1}{N}\sum_i [Y'^i - Y^i]^2$$

La régression linéaire est un cas particulier d'un tel modèle, avec une couche unique comportant un seul neurone avec une seule entrée pondérée par w à laquelle un biais b est ajouté. La fonction de perte à minimiser est l'erreur quadratique moyenne $L = \sum_k ((w \times a_k + b) - b_k)^2$; les paramètres w

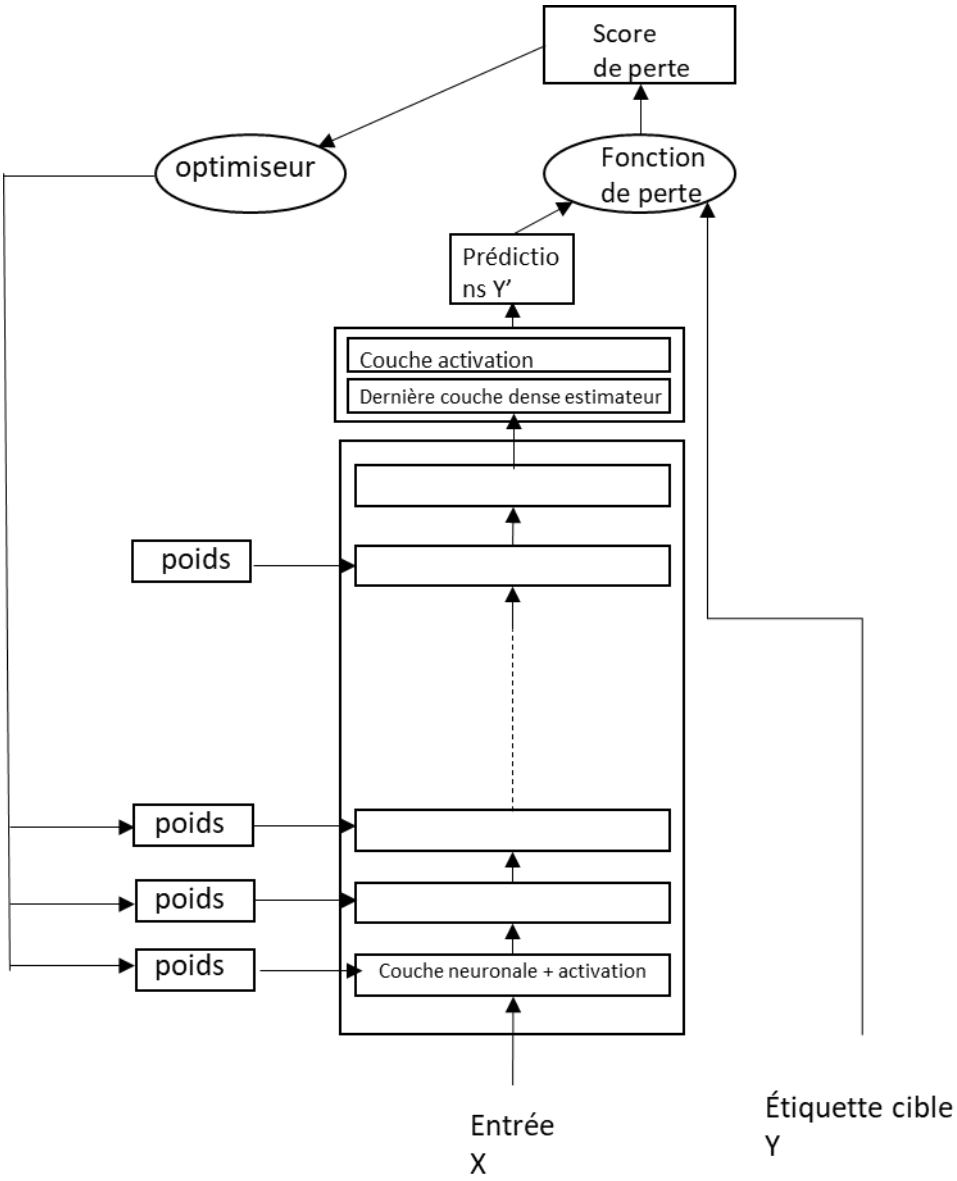
et b qui minimisent cette fonction peuvent être déterminés en calculant les dérivées partielles de cette fonction par rapport à w et b qui doivent s'annuler pour ces valeurs.

Pour un problème de classification à plusieurs classes, la sortie du réseau représente une distribution de probabilité relative aux différentes classes pour l'entrée considérée. Plutôt que l'erreur quadratique moyenne, la fonction de perte utilisée sera l'entropie croisée qui permet de mesurer une différence entre les deux distributions de probabilités. L'exposé des principes théoriques de cette fonction qui relèvent du champ de la théorie de l'information nous entrainerait beaucoup trop loin dans l'analyse, d'autant plus qu'il est difficile de trouver un document concis et clair en ligne sur ce sujet. Contentons-nous d'énoncer simplement que pour deux distributions de probabilités p_j et q_j, l'entropie croisée est définie comme étant la somme $-\sum_j p_j \log q_j$.

Ainsi, l'entropie croisée entre la sortie Y'^i et la valeur attendue Y^i est : - $\sum_k Y_k^i \log Y'^i_k$. Le vecteur étiquette Y^i a toutes ses coordonnées nulles sauf Y_k^i, qui vaut 1 pour l'indice k' correspondant à la classe attendue, si bien que l'entropie croisée s'écrit simplement - $\log Y'^i_k$,

3.5.5 Architecture récapitulative d'un réseau de neurones

L'architecture générale d'un réseau de neurones est présentée dans la figure ci-dessous :



L'architecture d'un réseau de neurones comporte un empilement de nombreuses couches adapté au problème à résoudre.

Le réseau alterne des couches linéaires (où chaque sortie est une somme pondérée des entrées) et des couches non-linéaires possédant le même nombre d'entrées et de sortie et qui consistent le plus souvent en la fonction ReLU (Rectified Linear Unit), prenant pour une entrée x la valeur maximale de x et de 0. Ces couches non-linéaires placées après chaque couche neuronale sont dites couches d'activation de la couche neuronale qui la précède.

Le réseau se termine par une couche dense entièrement connectée (c'est-à-dire que toutes ses entrées sont connectées à tous ses neurones).

La fonction de perte mesure l'erreur entre la valeur calculée par le réseau Y' pour l'entrée X et la valeur attendue Y (étiquette).

L'optimiseur ajuste les poids des couches pour que le score de perte soit le plus petit possible.

La régression linéaire est un cas particulier d'un tel modèle, avec une couche unique comportant un seul neurone avec une seule entrée pondérée par w à laquelle un biais b est ajouté. La fonction de perte à minimiser est l'erreur quadratique moyenne $L = \sum_k((w \times a_k + b) - b_k)^2$; les paramètres w et b qui minimisent cette fonction peuvent être déterminés en calculant les dérivées partielles de cette fonction par rapport à w et b qui doivent s'annuler pour ces valeurs.

4 L'entrainement des réseaux de neurones

4.1 La minimisation de la fonction de perte

Abordons maintenant l'algorithme de rétro-propagation du gradient qui est la fondation même du deep learning. Le principe fondamental de tout apprentissage reste toujours le même : il consiste à ajuster les paramètres d'une fonction de transformation des entrées de telle sorte que l'erreur commise par le système soit la plus petite possible sur les exemples d'entraînement.

Les paramètres de l'algorithme sont les différents poids et biais du réseau, et la fonction de perte, qui mesure l'erreur entre les sorties du réseau et les valeurs attendues, est choisie en fonction du type de problème à traiter comme cela a été vu plus haut.

Dans le cas d'un réseau de neurones comportant un très grand nombre de paramètres, la fonction de perte est vue comme une fonction dépendant de ces différents poids regroupés dans une variable Θ

et de l'ensemble des données d'entrainement $(x^i)_{1\leq i\leq N}$: L = L(Θ, (x^i))

Les données de l'ensemble d'entraînement étant fixées, il est possible de mettre en œuvre un algorithme de descente du gradient pour trouver un minimum Θ_0 de la fonction de coût L.

4.2 L'algorithme général de descente du gradient.

En dimension 1, pour une fonction f de variable réelle x, la méthode consiste, en partant d'une valeur initiale x_0, à déterminer d'une manière itérative une suite de valeurs qui s'approche de plus en plus de la valeur x qui minimise la fonction f.

A partir d'une valeur x_n, on détermine x_{n+1} en effectuant un déplacement dans le sens opposé à la dérivée $f'(x_n)$ de façon à faire diminuer f(x) : $x_{n+1} = x_n - \delta f'(x_n)$

Si la dérivée $f'(x_n)$ est positive, on diminue alors la valeur de x_n pour diminuer f et se rapprocher du minimum; si la dérivée $f'(x_n)$ est négative, on augmente alors légèrement x_n pour diminuer f et donc

se rapprocher du minimum. Le déplacement est d'autant plus grand que la valeur absolue de la dérivée est grande.

Le pas de gradient δ doit être choisi suffisamment grand pour converger vite et assez petit pour éviter que la suite de valeurs x_n oscille autour du minimum.

L'itération se termine quand l'écart entre deux valeurs successives de x_n devient inférieur à une valeur minimale fixée.

La descente de gradient peut être généralisée à des fonctions de plusieurs variables. En dimension 2, la fonction F à minimiser C dépend de deux variables x et y et peut être représentée par une surface où x est la longitude, y la latitude, et F(x,y) l'altitude du point considéré sur la surface. En partant d'un point quelconque, on effectue de proche en proche un déplacement dans le sens de la plus grande pente, dans la direction opposée à celle du gradient ∇F défini par ses deux dérivées partielles $\frac{\partial F}{\partial x}$ et $\frac{\partial F}{\partial y}$, jusqu'à atteindre le fond de la vallée.

Pour une fonction F dépendant de P variables $X = (x_1, \dots, x_P)$, la suite de valeurs est déterminée de proche en proche par :

$X^{n+1} = X^n - \delta * \nabla F$

où ∇F est le gradient de la fonction C au point X^n, vecteur de coordonnées $\left(\frac{\partial F}{\partial x_1}(X^n), \dots, \frac{\partial F}{\partial x_P}(X^n)\right)$.

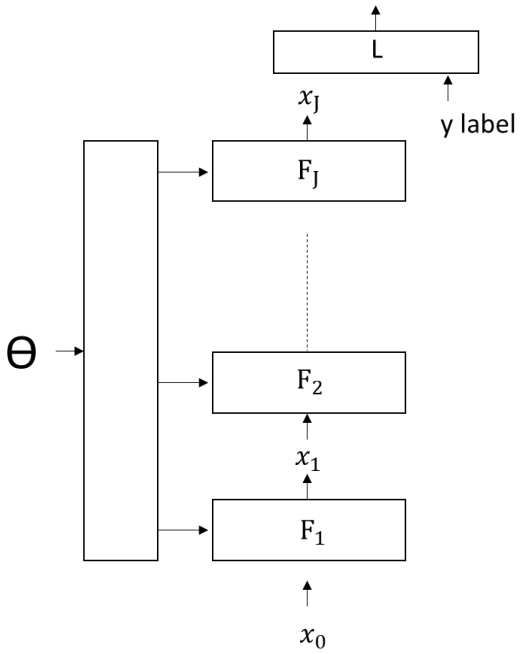
4.3 la rétropropagation du gradient

4.3.1 méthode de calcul du gradient

Pour être en mesure d'appliquer une descente de gradient afin de minimiser la fonction de perte d'un réseau de neurones, ll est indispensable de disposer d'une méthode de calcul du gradient d'une telle fonction L(Θ, (x_0^i)) par rapport à la variable Θ. La fonction de perte est la moyenne des pertes sur les différents exemples d'entrainement $L = \frac{1}{N}\sum_i L(\Theta, x^i)$. Aussi, nous allons calculer les gradients de la fonction de perte d'un exemple quelconque de l'ensemble d'apprentissage, et il suffira alors d'additionner tous les gradients obtenus pour obtenir le gradient de la fonction de cout sur l'ensemble de l'échantillon.

L'algorithme de rétro propagation propose un calcul analytique du gradient de la fonction de coût par rapport aux poids du réseau. Comme on va le voir, les gradients sont calculés de proche en proche en partant de la dernière couche du réseau et en propageant les calculs à rebours de couche en couche, d'où l'origine de cette appellation de rétro propagation. L'algorithme effectue seulement deux passes dans le réseau, une passe avant pour calculer les valeurs des sorties, et une passe arrière pour calculer les gradients de la fonction de perte par rapport aux poids des différentes couches.

Comme nous l'avons vu, tout réseau de neurones est constitué de couches empilées les unes sur les autres, selon le schéma ci-dessous :



Le réseau comporte J couches, j est l'indice de la couche j, qui a pour sortie le vecteur x_j. Les entrées de la couche j sont par construction les sorties x_{j-1}de la couche d'indice j-1. La sortie du réseau est x_J.

Le but est de calculer le gradient de la fonction L par rapport aux poids du réseau, soit les dérivées partielles $\frac{\partial L}{\partial w}$ pour chacun des poids élémentaires du réseau.

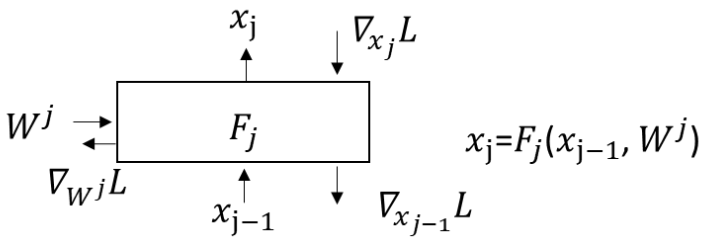
La compréhension des équations de rétropropagation du gradient nécessite beaucoup de patience et il ne faut pas s'attendre à les assimiler de manière instantanée. En revanche, la stratégie de l'algorithme est assez simple à comprendre.

L'approche consiste à considérer la fonction d'un réseau de neurones comme un enchaînement de fonctions composées sur lequel on va pouvoir appliquer les règles de la dérivation des fonctions composées. Dans l'exemple ci-dessus avec trois couches,

$$x_3 = F_3(x_2) = F_3(F_2(x_1)) = F_3(F_2(F_1(x_0)))$$

Cette approche pouvant suffire en première lecture, nous allons présenter dans la suite la procédure pas à pas de calcul des gradients, en énonçant simplement les équations sans les démontrer. Les démonstrations détaillées seront enfin présentées au paragraphe 4.3.2 pour les lecteurs désireux d'approfondir l'étude.

Pour calculer la dérivée partielle $\frac{\partial L}{\partial W_{mp}^j}$ par rapport à un poids quelconque du réseau, il suffit de se placer au niveau de la couche comportant le neurone auquel ce poids est affecté. Si j est l'indice de cette couche, la règle de dérivation des fonctions composées permet de relier le gradient de la perte $\nabla_{W^j} L$ par rapport aux poids de la couche j au gradient de la perte $\nabla_{x_j} L$, cette fois par rapport aux sorties de la couche j.



En effet, on la relation suivante :

$\frac{\partial L}{\partial W_{mp}^j} = \frac{\partial L}{\partial x_j(m)} \times x_{j-1}(p)$

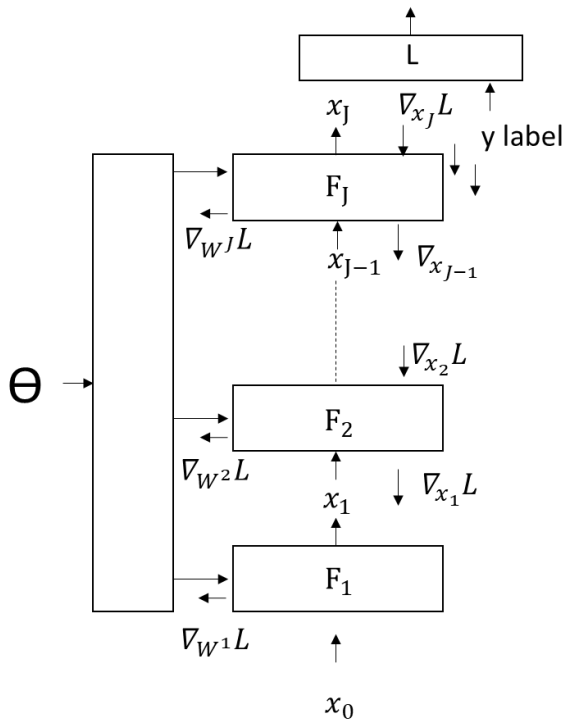
$x_{j-1}(p)$, qui est la p[ième] sortie de la couche j-1 est bien un terme connu qui résulte du calcul des sorties intermédiaires de la première couche à la couche j-1 tandis que les termes $\frac{\partial L}{\partial x_j(m)}$ représentent les composantes du gradient $\nabla_{x_j} L$ de la perte par rapport aux sorties de la couche j du réseau.

Une deuxième application de la règle de composition des fonctions permet de déduire le gradient $\nabla_{x_{j-1}} L$ de la perte par rapport aux sorties de la couche j-1 du gradient $\nabla_{x_j} L$ de la perte par rapport aux sorties de la couche j.

Les gradients $\nabla_{x_j} L$ par rapport aux sorties des couches sont ainsi calculés de proche en proche jusqu'à la sortie de la première couche, d'où le terme de rétropropagation du gradient.

Les gradients recherchés se déduisent des termes $\nabla_{x_j} L$ selon la formule $\frac{\partial L}{\partial W_{mp}^j} = \frac{\partial L}{\partial x_j(m)} \times x_{j-1}(p)$ vue plus haut.

Le schéma ci-dessous résume la démarche de déduction des gradients de proche en proche :



Pour exprimer la relation entre deux gradients $\nabla_{x_{j-1}} L$ et $\nabla_{x_j} L$, deux cas sont à considérer selon que l'on est en présence d'une couche neuronale ou d'une couche d'activation.

a) Cas d'une couche neuronale :

W^j étant la matrice des poids de la couche j, le gradient $\nabla_{x_{j-1}} L$ de la fonction de perte par rapport aux sorties de la couche j-1 s'obtient en effectuant le produit matriciel de la matrice transposée de W^j par le gradient $\nabla_{x_j} L$ de la perte par rapport aux sorties de la couche j-1 :

$$\nabla_{x_{j-1}} L = (W^j)^T \times \nabla_{x_j} L$$

Cette relation peut s'écrire pour chaque composante m du gradient $\nabla_{x_{j-1}} L$, $\frac{\partial L}{\partial x_{j-1}(m)}$

$$\frac{\partial L}{\partial x_{j-1}(m)} = \sum_p W^j_{pm} \times \frac{\partial L}{\partial x_j(p)}$$

Le gradient de la perte par rapport aux sorties d'une couche neuronale s'obtient donc en appliquant une transformation neuronale au gradient de la perte par rapport aux sorties de la couche immédiatement supérieure selon la matrice de poids $(W^j)^T$ d'où la notion de passe vers l'arrière pour le calcul de ces gradients de proche en proche, en utilisant les matrices de poids transposées.

b) Cas d'une couche d'activation :

Si a est la fonction d'activation, on a pour chaque composante m de x_j :

$$x_j(m) = a(x_{j-1}(m))$$

On en déduit donc que $\frac{\partial L}{\partial x_{j-1}(m)} = \frac{\partial L}{\partial x_j(m)} \times\ a'(x_{j-1}(m))$

Il ne reste plus qu'à initialiser la démarche en calculant le gradient de la perte par rapport aux sorties finales x_J du réseau :

Pour le cas de la régression linéaire, La fonction de perte est : $l(x_J, y) = \frac{1}{2}(x_J - y)^2$ donc

$\frac{\partial L}{\partial x_J} = x_J - y$

Dans le cas de la classification, la fonction Softmax est appliquée à la sortie x_J du réseau avant de calculer l'entropie croisée si bien que :

$l(x_J, y) = -\sum_1^N y(k) \times \log(\text{Softmax}(x_J(k))$ où Y est un vecteur contenant des 0 partout sauf pour l'indice correspondant à la probabilité de la bonne classe, qui est la valeur 1.

On montre facilement que : $\nabla_{x_J} l = \text{Softmax}(x_J) - y$

4.3.2 Démonstration des équations de la rétropropagation

La méthode de calcul du gradient par rétropropagation utilise les règles du calcul différentiel sur la dérivation des fonctions composées. Un premier résultat utilisé est la règle de dérivation de la composition de deux fonctions de variables réelles qui pose que la dérivée de la fonction $g \circ f$ qui associe à x la valeur g(f(x)) est la fonction $(g' \circ f) * f'$ qui à x, associe g'(f(x)) $* f'$(x).

Une autre règle, moins connue des très anciens étudiants de classes préparatoires est la généralisation de la règle de dérivation des fonctions composées aux fonctions de plusieurs variables : la matrice jacobienne d'une composée de fonctions de plusieurs variables est le produit des matrices jacobiennes. Si F est une fonction de N variables (x_i) prenant ses valeurs dans un espace à P dimensions et si F^j sont les P composantes de F, la matrice jacobienne notée $\frac{\partial F}{\partial x}$ désigne la matrice dont chaque ligne d'indice i est la suite des dérivées partielles des différentes composantes de F par rapport à la variable x_i , soit la matrice $\begin{bmatrix} \frac{\partial F^1}{\partial x_1} & \cdots & \frac{\partial F^P}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial F^1}{\partial x_N} & \cdots & \frac{\partial F^P}{\partial x_{1N}} \end{bmatrix}$

Ainsi, $\frac{\partial G \circ F}{\partial x} = \frac{\partial F}{\partial x} \times \frac{\partial G}{\partial y}$

Ce résultat permet par exemple de calculer la dérivée d'une fonction s'écrivant sous la forme :

$f(t) = F\big(u(t), v(t)\big)$ où F est une fonction différentiable de deux variables x et y et u et v des fonctions d'une variable réelle t. La dérivée de f est la fonction :

$$f'(t) = \frac{\partial F}{\partial x}\big(u(t), v(t)\big) * u'(t) + \frac{\partial F}{\partial y}\big(u(t), v(t)\big) * v'(t)$$

qui est le produit de la matrice ligne $\lfloor u'(t), v'(t) \rfloor$ par la matrice colonne $\begin{bmatrix} \frac{\partial F}{\partial x} \\ \frac{\partial F}{\partial y} \end{bmatrix}$

a) <u>preuve de l'équation de propagation des gradients de la perte par rapport aux sorties :</u>

il suffit de considérer la fonction de perte par rapport aux variables $x_{\mathrm{j}-1}$ comme la composition de deux fonctions :

- La fonction F_j de la couche j qui à $x_{\mathrm{j}-1}$ associe x_{j}
- La fonction qui à x_{j} associe la perte L

En appliquant la règle de composition des fonctions à plusieurs variables, on peut écrire :

$$\frac{\partial L}{\partial x_{j-1}} = \frac{\partial x_{\mathrm{j}}}{\partial x_{\mathrm{j}-1}} \times \frac{\partial L}{\partial x_{\mathrm{j}}}$$

Pour une couche neuronale, le jacobien $\frac{\partial F_j}{\partial x_{\mathrm{j}-1}} = \frac{\partial x_j}{\partial x_{\mathrm{j}-1}}$de la fonction F_j est la matrice $(W^j)^T$; en effet :

$x_j(p) = \sum_m W_{pm}^j \times x_{j-1}(m)$ car $x_j = W^j \times x_{j-1}$ et la dérivée partielle $\frac{\partial x_j(p)}{\partial x_{j-1}(m)}$ est donc le terme W_{pm}^j ce qui vérifie bien que $\frac{\partial F_j}{\partial x_{j-1}} = (W^j)^T$

Les jacobiens $\frac{\partial L}{\partial x_{j-1}}$ et $\frac{\partial L}{\partial x_j}$ n'étant autres que les gradients de la fonction de perte $\nabla_{x_{j-1}} L$ et $\nabla_{x_j} L$, la règle de composition donne la relation suivante entre ces deux quantités :

$$\nabla_{x_{j-1}} L = (W^j)^T \times \nabla_{x_j} L$$

Pour une couche d'activation, le jacobien de la fonction F_j est la matrice diagonale constituée des valeurs dérivées de la fonction d'activation a sur les différentes composantes du vecteur x_{j-1}, et la multiplication de cette matrice diagonale par le gradient $\nabla_{x_j} L$ fournit bien le résultat recherché :

$$\frac{\partial L}{\partial x_{j-1}(m)} = \frac{\partial L}{\partial x_j(m)} \times a'(x_{j-1}(m))$$

b) <u>Preuve de la relation entre le gradient de la perte par rapport aux poids et son gradient par rapport aux sorties x_j :</u>

Cette fois, il s'agit de considérer la fonction de perte comme la fonction composée des deux fonctions :

- La fonction F_j qui, à W^j associe la sortie x_j, x_{j-1} étant fixé (la variation du seul poids W_{mp}^j de la couche j n'affectant pas la sortie des couches inférieures)
- La fonction qui à x_j associe la perte L

L'application de la règle de composition des fonctions par multiplication des jacobiens n'est pas directement possible, car la première fonction a ses variables indicées dans une matrice de dimension 2 qu'il faudrait alors aplatir. La bonne méthode consiste à calculer la dérivée partielle de la perte par rapport à un poids élémentaire W_{mp}^j dans cette matrice.

Vue sous l'angle d'une seule variable W_{mp}^j, la formule de dérivation est la suivante, en appliquant directement le résultat $f'(t) = \frac{\partial F}{\partial x}\big(u(t), v(t)\big) * u'(t) + \frac{\partial F}{\partial y}\big(u(t), v(t)\big) * v'(t)$ qui est généralisable à une fonction F de plus de deux variables :

$$\frac{\partial L}{\partial W_{mp}^j} = \sum_k \frac{\partial L}{\partial x_j(k)} \times \frac{\partial x_j(k)}{\partial W_{mp}^j}$$

Ainsi, il ne reste plus à calculer que le terme $\frac{\partial x_j(k)}{\partial W_{mp}^j}$

Or, $x_j(k)$ ne dépend que des poids W_{kl}^j du neurone de rang k donnant la sortie $x_j(k)$:

$x_j(k) = \sum_l W_{kl}^j \times x_{j-1}(l)$

Si k est différent de m, $x_j(k)$ ne dépend donc pas du poids W_{mp}^j et sa dérivée partielle par rapport à celui-ci est nulle.

On a donc :

$$\frac{\partial L}{\partial W_{mp}^{j}} = \frac{\partial L}{\partial x_j(m)} \times \frac{\partial x_j(m)}{\partial W_{mp}^{j}}$$

$\frac{\partial x_j(m)}{\partial W_{mp}^{j}} = x_{j-1}(p)$ donc en remplaçant dans l'égalité ci-dessus,

$$\frac{\partial L}{\partial W_{mp}^{j}} = \frac{\partial L}{\partial x_j(m)} \times x_{j-1}(p)$$

Il reste un dernier effort à accomplir pour calculer le gradient de la fonction de perte par rapport aux paramètres de biais b_j :

En intégrant le biais dans la couche neuronale sous la forme $x_{\mathrm{j}} = W^{j} \times x_{\mathrm{j}-1} + b_j$, ce qui ne change pas les résultats jusqu'présent obtenus, la règle de composition des fonctions permet d'écrire :

$$\frac{\partial L}{\partial b_j} = \frac{\partial x_{\mathrm{j}}}{\partial b_j} \times \frac{\partial L}{\partial x_{\mathrm{j}}}$$

Le jacobien $\frac{\partial x_{\mathrm{j}}}{\partial b_j}$ est la matrice identité I comportant des 1 sur la diagonale et des 0 partout ailleurs, car la dérivée partielle d'une composante de $x_{\mathrm{j}}(m)$ par rapport à un biais $b_j(p)$ est nulle pour m différent de p et vaut 1 pour m=p.

$$\frac{\partial L}{\partial b_j} = I \times \frac{\partial L}{\partial x_{\mathrm{j}}} = \frac{\partial L}{\partial x_{\mathrm{j}}}$$

4.3.3 Mise en œuvre de la descente de gradient

La descente de gradient est effectuée en itérant plusieurs fois sur l'ensemble des données d'apprentissage. Chaque itération sur l'ensemble des données est appelée un « epoch ». Un choix aléatoire d'un vecteur de poids est effectué avant le lancement du processus, et donc de la première epoch. Puis, à chaque itération ou epoch, une descente de gradient est effectuée avec une boucle de mise à jours des poids en effectuant un pas de gradient à chaque passage de la boucle. Plusieurs méthodes sont possibles pour le calcul du gradient à chaque pas. On pourrait utiliser l'ensemble des données disponibles quoique cette méthode soit couteuse en temps de calcul. La méthode généralement employée est la descente de gradient stochastique qui consiste à calculer un gradient moyen estimé sur un sous-ensemble de l'échantillon tiré aléatoirement. A chaque epoch, l'ensemble des données est alors découpé de manière aléatoire en sous-ensembles appelés « mini-batch » de taille égale fixée par un paramètre en entrée du processus global. Un mini-batch est utilisé pour le calcul du gradient à chaque mise à jour des poids selon la formule w_i += w_i - $\delta * \frac{\partial L}{\partial w_i}$

Au bout d'une epoch, l'ensemble des données a été parcouru, avec autant de mises à jour des poids que de batchs.

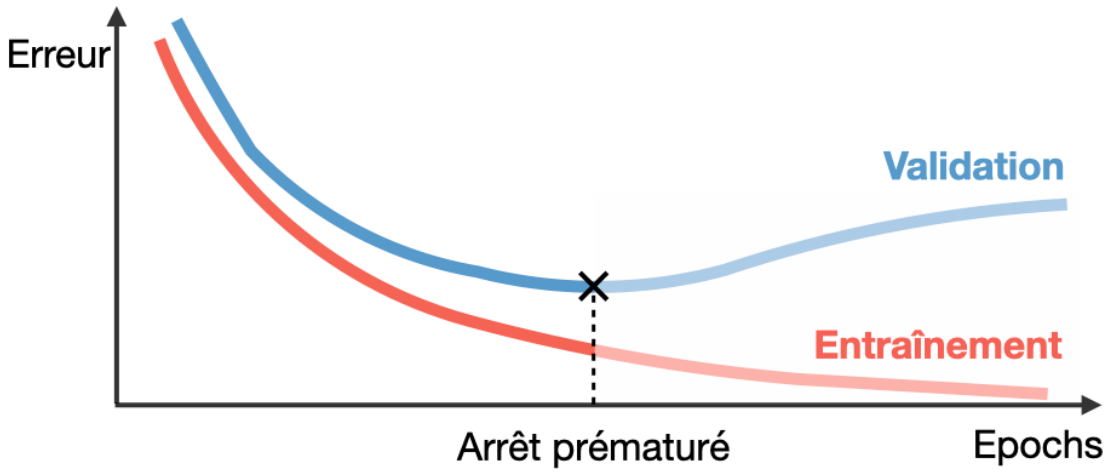
L'algorithme s'arrête après avoir effectué le nombre d'epoch fixé en paramètre d'entrée.

Par exemple, supposons que l'on dispose d'un échantillon de 10 000 exemples, et que l'on mette en œuvre une descente de gradient avec 5 epochs et une taille de mini-batchs de 100, ce qui fait 100 batches pour chaque epoch. A chaque epoch l'algorithme effectue ainsi 100 mises à jour de gradient, une par mini-batch. A l'issue des 5 epochs, le réseau aura donc effectué 500 mises à jour de gradient.

Cette manière de mettre en œuvre la descente de gradient est appelé la descente de gradient stochastique sur un mini-lot de données (mini-batch stochastic gradient descent – mini-batch SGD). Le terme « stochastique » est relatif au choix aléatoire des lots de données.

4.4 Test du modèle – lutte contre le sur-ajustement

Un réseau une fois entraîné doit pouvoir être testé sur des données qu'il n'a jamais rencontrées au cours de la phase d'apprentissage. Pour ce faire, une partie de l'échantillon sera séparé des données d'origine pour créer un ensemble de données de validation. L'évaluation du modèle sur les données de validation va même pouvoir être effectuée au cours de la phase d'entraînement. Ce test consiste à l'issue de chaque epoch à mesurer l'exactitude de prédiction du modèle et sa fonction de coût ou perte, non seulement sur l'ensemble des données d'entraînement, mais aussi sur les données de validation. Il est alors possible de suivre l'évolution des performances du modèle au fur et à mesure de l'avancée des itérations. D'une manière générale, on constate alors que la perte d'entrainement (c'est-à-dire mesurée sur les données d'entrainement) diminue à chaque époch, ce qui est bien le but visé de l'optimisation par descente de gradient ; de même l'exactitude d'entrainement augmente à chaque époque sur les données d'entraînement. En revanche, cette tendance s'inverse sur les données de validation : alors que la perte de validation commence par diminuer à chaque époque, et l'exactitude de validation à s'améliorer, les performances du modèle passent par un point culminant et se dégradent ensuite sur les données de validation au bout de seulement quelques époques. Ce phénomène est représentatif d'un sur-ajustement (overfitting) du modèle aux données d'entraînement : le modèle est en sur apprentissage sur les données d'entrainement si bien qu'il n'est plus capable de généraliser de manière pertinente sur des données en dehors de cet ensemble d'apprentissage.



Une des meilleures façons de réduire le sur-ajustement est d'augmenter la taille des données d'entraînement. Cependant, l'acquisition de données d'entraînement est en général un processus difficile et couteux. Une autre manière de réduire le sur-ajustement est de réduire la taille du réseau, mais du fait que les gros réseaux ont un potentiel de capacités plus important, on n'adoptera cette option qu'à contre-cœur.

Heureusement, il existe d'autres techniques permettant de réduire le sur-ajustement, même avec un réseau figé et un jeu fixé de données d'entraînement, comme la régularisation des poids ou l'ajout de drop out. La régularisation des poids consiste à ajouter à la fonction de perte un terme supplémentaire proportionnel au carré de la somme des poids (régularisation L2), ou à la somme de leurs valeurs absolues (régularisation L1), ce qui a pour effet de favoriser l'apprentissage de poids faibles. Le drop out est l'une des techniques de régularisation les plus efficaces et consiste à supprimer aléatoirement des neurones dans les couches cachées du réseau au cours de l'entraînement. Nous n'entrerons pas dans le détail de l'ensemble de ces techniques.

5 Les réseaux de neurones convolutifs

L'architecture du réseau est le paramètre clé pour obtenir de bons résultats. Empiler des couches de neurones qui sont entièrement connectées les unes avec les autres sans aucune idée de structuration du réseau ne permet pas d'atteindre des résultats spectaculaires. En fait, comme l'explique Stéphane Mallat dans son cours au Collège de France, l'apprentissage profond subit « la malédiction de la grande dimension ». On pourrait penser que si l'on dispose d'un nombre d'exemples suffisants, le réseau sera en quelque sorte capable d'interpoler le résultat à partir des valeurs de la fonction sur des points voisins. Le problème de la grande dimension est qu'il est absolument impossible de garantir l'existence de voisins proches pour effectuer une interpolation avec une bonne approximation. Supposons par exemple que toutes les variables soient comprises entre 0 et 1 ce qui signifie que les points sont situés dans un cube $[0,1]^d$. En dimension 1 (d=1), seulement 10 points suffisent pour garantir qu'il existe pour tout point un voisin plus proche que 1/10. En dimension 2, il faudra au moins $10 \times 10 = 100$ points, et en dimension quelconque d, il faut 10^d points et comme d est très grand, de l'ordre de 1 millions, ce chiffre est bien plus qu'astronomique, 10^{80} étant déjà une estimation du nombre d'atomes dans l'univers.

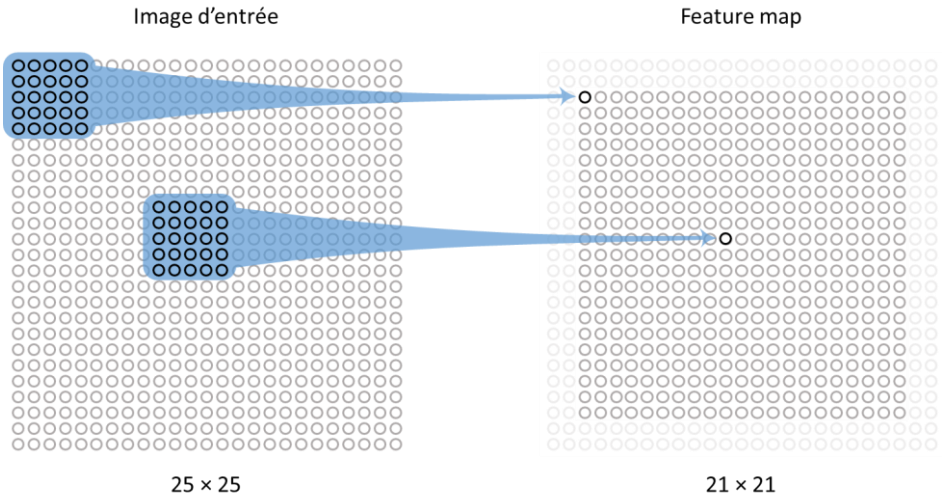
En grande dimension, les points de l'espace d'apprentissage sont fortement éloignés les uns des autres si bien qu'il va falloir se rendre capable de découvrir des formes de régularité qui permettront de réduire la complexité ou la dimension du problème, en structurant le réseau d'une manière particulière.

Une des premières applications de ce principe est l'introduction des réseaux de neurones convolutifs, dont la mise au point résulte en grande partie des travaux du chercheur français Yann Le Cun au milieu des années 1990. Leur principe de base est l'application de filtres permettant de détecter des motifs invariants par translation, et de répéter l'application de ces filtres sur des zones de plus en plus grandes de l'image pour trouver des motifs de plus en plus complexes et ainsi résoudre la tâche cible.

La classification d'une image doit pouvoir être effectuée à partir de certains motifs caractéristiques dans l'image et ne concernant qu'un sous-ensemble de pixels situés dans un même voisinage. Par ailleurs, un réseau ayant appris un motif dans une région particulière d'une image devra demeurer

capable de le reconnaître n'importe où dans l'image sans avoir à le réapprendre : c'est ce qui est couramment appelé l'invariance par translation.

Pour détecter des motifs locaux dans une image, l'idée de base des réseaux de neurones convolutifs est d'utiliser une fenêtre de taille réduite, par exemple de 5 par 5 pixels, soit 25 pixels en tout, qui parcourt l'image dans toutes les positions possibles en calculant pour chaque position de cette fenêtre une somme pondérée de ses pixels. Pour une position donnée de la fenêtre sur l'image, cette somme pondérée peut être vue comme le résultat de l'application d'un neurone sur les 25 entrées que constituent les pixels de la fenêtre. A chaque position de la fenêtre, correspond donc un neurone qui calcule la somme pondérée de ses pixels, chaque fois en utilisant les mêmes poids pour toutes les positions. Cette fenêtre agit comme un filtre qui permet de reconnaître la présence d'un motif particulier comme un bord, un trait droit, une courbe ou une forme pour la reconnaissance de caractères par exemple. En répétant cette opération pour toutes les positions sur l'image d'entrée, on obtient une image en sortie contenant les résultats aux différentes positions. Une telle image s'appelle une feature map, ou carte de caractéristiques, car elle permet de détecter la présence d'un motif bien particulier (bord ou autre type de forme) sur les différentes parties de l'image en entrée.



Au cours du passage de cette fenêtre, plusieurs cartes de caractéristiques sont calculées, qui correspondent chacune à des filtres différents ayant des poids spécifiques. Les différentes cartes de caractéristiques obtenues filtre par filtre sont empilées pour former la sortie de la couche, qui possède ainsi trois dimensions, hauteur, largeur, profondeur.

On répète ensuite le processus en appliquant au résultat de la première couche une seconde couche de convolution, et ainsi de suite, les filtres de convolution prenant chaque fois en compte les trois dimensions de la sortie de la couche précédente sur la fenêtre de convolution.

Les différentes couches sont ainsi capables d'apprendre des hiérarchies spatiales de motifs de plus en plus abstraits et complexes issus des caractéristiques des couches précédentes. A l'issue du processus, il va être possible d'effectuer une classification à l'aide d'une dernière couche dont chaque sortie estimera la probabilité que l'entrée concernée soit dans une classe.

Le schéma général est le suivant :