Généralités réseaux de neurones et deep-learning

June 2021

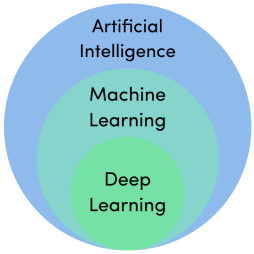
Durée : 25 mn

1 Prérequis

- Notions de programmation et d'algorithmique primaires

2 Acquis d'apprentissage

- Notions de construction de neurones et réseaux denses



https://levity.ai/blog/difference-machine-learning-deep-learning

Le terme d'**intelligence artificielle** définit l'ensemble des théories et des techniques mises en œuvre en vue de réaliser des machines capables de mimer le fonctionnement du cerveau humain, ou du moins sa logique lorsqu'il s'agit de prendre des décisions.

Le terme de **machine learning** désigne une sous-branche de l'intelligence artificielle. Il inclut des techniques qui permettent aux machines d'améliorer leurs performances par apprentissage. Il existe deux types d'apprentissage : l'*apprentissage supervisé* et *non-supervisé*.

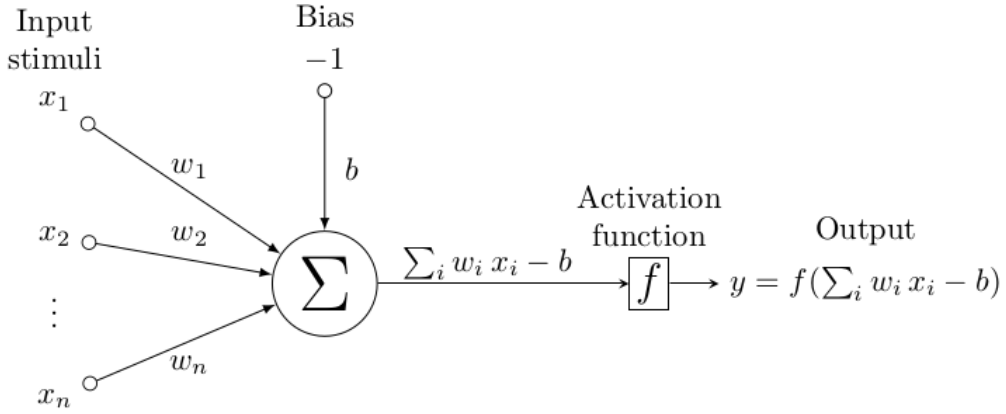
- L'*apprentissage supervisé* correspond à un réseau où chaque données d'entraînements est attachée à un label. Ce dernier correspond à la réponse que l'algorithme doit trouver.
- L'*apprentissage non supervisé* correspond à un réseau où les données d'entraînements ne sont pas étiquetées. L'algorithme ne peut donc pas vérifier l'exactitude de ses réponses.

Le terme de **deep learning** désigne une sous-branche de machine learning. Il est basé sur des réseaux de neurones qui permettent à la machine de s'entraîner pour réaliser une certaines tâche. Les réseaux profonds utilisent plusieurs couches de neurones (couches cachées).

3 Réseaux de neurones

3.1 Le neurone artificiel

C'est une unité de traitement informatique qui reçoit des entrées x_i affectées chacune d'un poids ω_i. La sortie du neurone est donnée par la valeur de sa **fonction d'activation** en un point défini par la **combinaison linéaire de ses entrées** $\sum_i \omega_i x_i - b$:



JLC

L'entrée *Bias* (b) reçoit le stimuli "-1" affecté du poids b : il permet de décaler le point où la fonction d'activation est calculée.

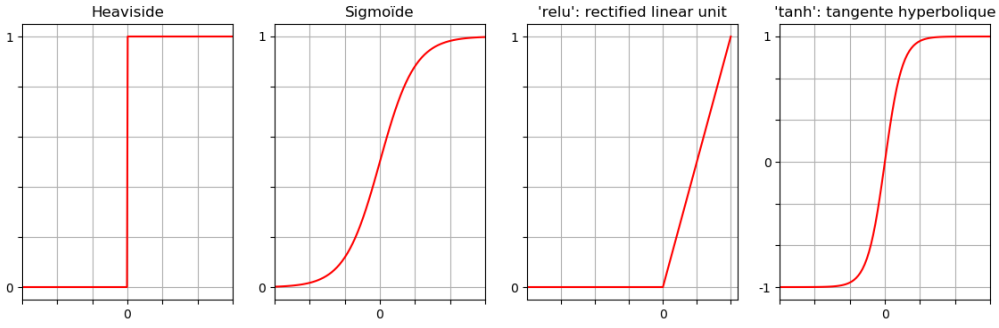
3.2 Fonction d'activation

La fonction d'activation permet de calculer la sortie d'un neurone artificiel.

Principaux rôles :

- introduire dans le neurone un comportement **non linéaire** (comme des mécanismes de seuil, de saturation...)
- fixer la plage de sortie de la valeur calculée par le neurone, par exemple dans l'intervalle $[-1;1]$, $[0;1]$ ou encore $[0;+\infty[$

Exemples de fonctions d'activations couramment utilisées :



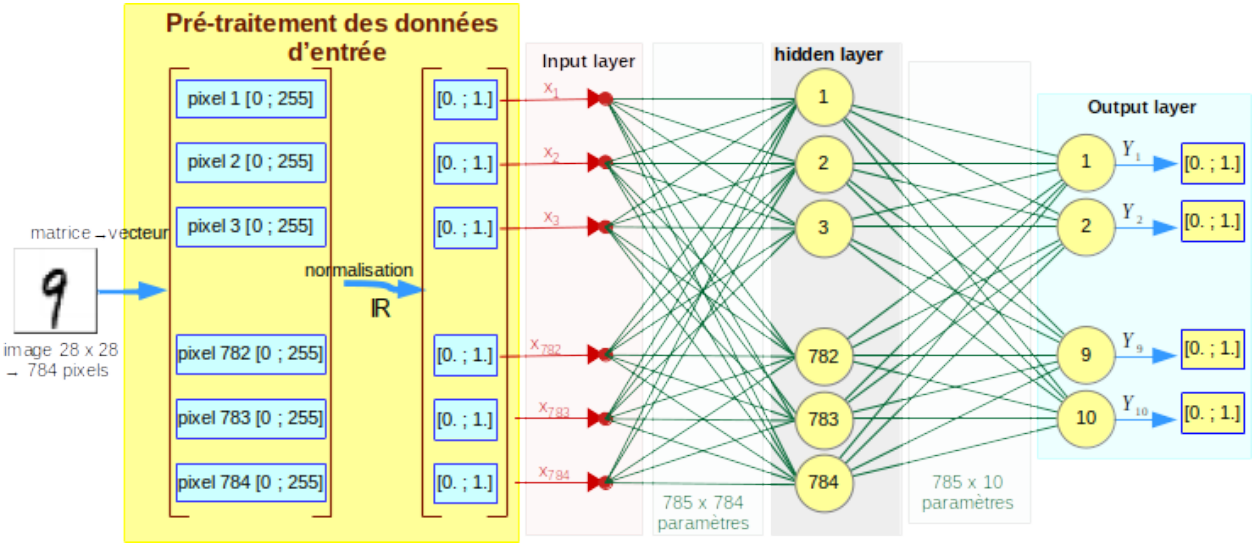
JLC

3.3 Réseau dense séquentiel

Pour créer un réseau de neurones dense séquentiel on connecte les sorties des neurones d'une couche sur les entrées des neurones de la couche suivante. Chaque neurone possède ses propres poids et son propre biais. On peut voir chaque neuronne comme une fonction : il prend toutes les sorties de la couche précédente puis il calcule et renvoit la valeur de sa fonction d'activation.

L'image ci-dessous présente un **réseau dense séquentiel**, avec :

- une **couche d'entrée** de 784 valeurs comprises entre 0 et 1 (les pixels des images MNIST 28 × 28 mis sous forme d'un vecteur de 784 nombres 'float'),
- une **couche cachée** de 784 neurones utilisant la fonction d'activation 'relu',
- une **couche de sortie** à 10 neurones, pour la classification des images en 10 classes associées aux chiffres 0,1,2...9, utilisant la fonction d'activation 'softmax' adaptée aux problèmes de classification.



JLC

Remarques :

- Chaque neurone de la première couche cachée reçoit 785 entrées : les 784 valeurs x_i des pixels de l'image plus le biais (l'entrée '-1').
- Il y a donc 785 inconnues pour chaque neurone : les 784 poids w_i affectés à chaque entrée x_i, plus le poids b affecté au biais.
- On compte donc 785 × 784 inconnues pour la couche cachée et 785 × 10 inconnues pour la couche de sortie : soit un total de 623290 inconnues dont la valeur doit être optimisée par l'algorithme d'apprentissage du réseau.

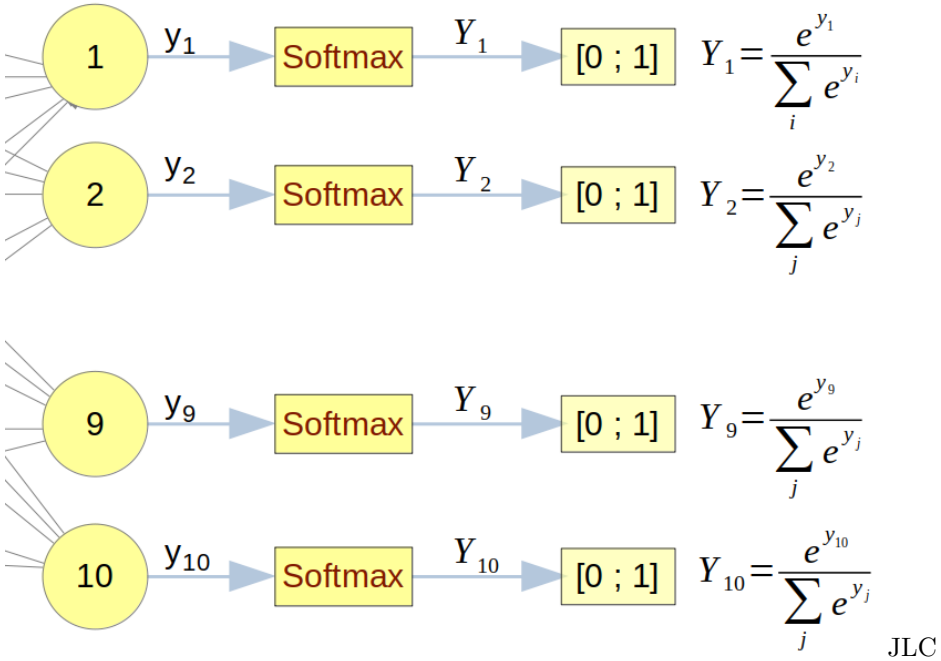
3.4 Intérêt de la fonction d'activation softmax pour les problèmes de classification

Dans les problèmes de classification, on a besoin que les sorties de la dernière couche puissent servir à décider quelle est la classe reconnue par le réseau pour une enrée donnée. Une solution est de transformer les sorties

des neurones de la dernière couche en probabilités, en utilisant la fonction d'activation **Softmax**. On obtient alors une distribution de probabilité : la somme de toutes les sorties est égale à 1.

La fonction 'softmax' calcule pour chaque neurone de sortie k la valeur $Y_k = \frac{e^{y_k}}{\sum_i e^{y_i}}$, où y_k désigne la combinaison linéaire $\sum_i \omega_i x_i - b$ calculée par le neurone k.

'softmax' associe ainsi à chacune des sorties y_k une valeur $Y_k \in [0, 1]$ qui peut être interprétée comme la probablité de la sortie k : on obtient une valeur proche de 1 pour le neurone fournissant la valeur y_i la plus grande, et quasiment 0 pour tous les autres.



JLC

3.5 Catégorisation des labels (one-hot coding)

Dans les problèmes d'entraînement supervisé, les classes à reconnaître sont identifiées par une étiquette "lisible" par l'être humain ("chien", "vélo", "personne"...), présentée au réseau de neurones sous la forme d'un entier (le numéro de classe: "chien" $\rightarrow$ 1, "vélo $\rightarrow$ 2...). La comparaison des sorties du réseau de neurones (liste de probabilités) avec l'entier désignant la classe n'est pas directe... Pour faciliter la comparaison entre l'entier désignant une clase à reconnaître et les probabilités calculées par le réseau sur ses neurones de sortie, on utilise la technique de **catégorisation** qui associe à un entier un vecteur dont les éléments sont tous nuls sauf un (one-hot coding).

Par exemple pour un problème de classification à 10 classes la représentation one-hot coded donne :

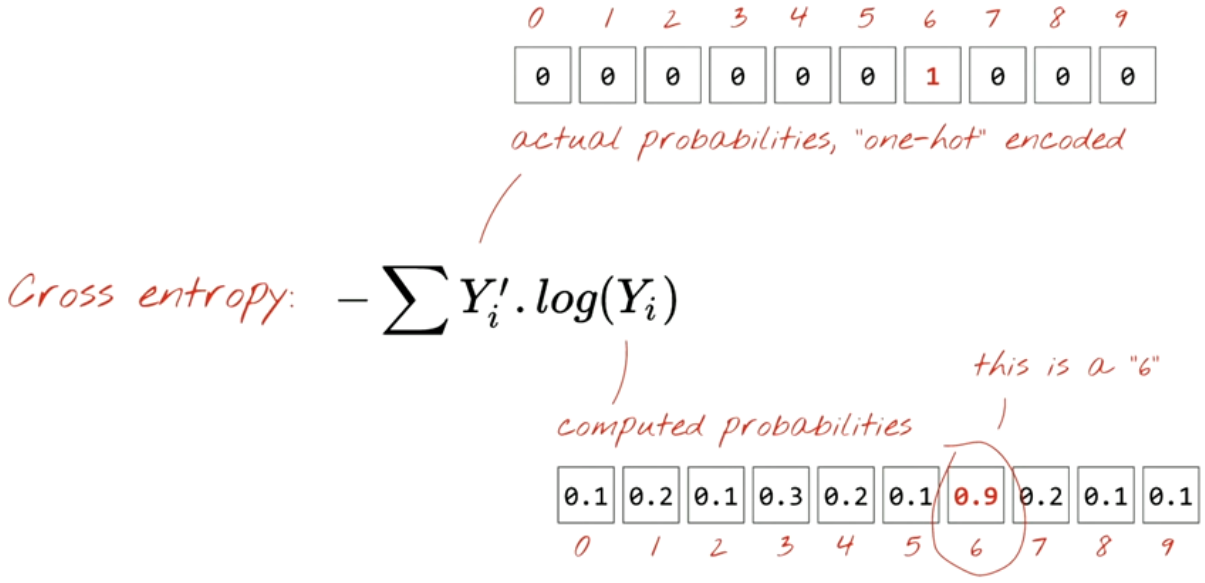
chiffre	Y'_i : vecteur *one-hot*
0	[1 0 0 0 0 0 0 0 0 0]
1	[0 1 0 0 0 0 0 0 0 0]
2	[0 0 1 0 0 0 0 0 0 0]
3	[0 0 0 1 0 0 0 0 0 0]
4	[0 0 0 0 1 0 0 0 0 0]
5	[0 0 0 0 0 1 0 0 0 0]
6	[0 0 0 0 0 0 1 0 0 0]
7	[0 0 0 0 0 0 0 1 0 0]
8	[0 0 0 0 0 0 0 0 1 0]
9	[0 0 0 0 0 0 0 0 0 1]

3.6 Calcul de l'erreur d'inférence du réseau

Dans les problèmes d'**entraînement supervisé**, on doit calculer l'erreur d'inférence du réseau lorsqu'on l'entraîne avec des données labellisées. Cette erreur sert à modifier les poids du réseau grâce à l'algorithme de *back propagation*.

On définit une fonction de perte (*loss function*) qui permet de calculer l'écart entre l'inférence du réseau de neurones et le résultat attendu. Le but de l'apprentissage est de modifier les poids du réseau de neurones pour minimiser cette fonction de perte.

Dans les problèmes de **classification** où l'inférebce du réseau se présente sous la forme d'un vectuer de probabiltés, on peut utiliser l'erreur *cross entropy* ($-\sum_i Y'_i . \log Y_i$) qui mesure l'écart entre la représentation *one-hot* du label et la réponse du réseau :



vidéo "Deep Learning TensorFlow" de Martin Gorner

3.7 Backpropagation error

L'algorithme de **rétropropagation de l'erreur** permet de modifier les poids des neurones du réseau pour minimiser l'erreur entre valeur attendue et valeur prédite par le réseau. Au fur et à mesure des apprentissages

successifs, les poids du réseau convergent vers un état qui minimize l'erreur d'inférence du réseau et qui constitue l'état du réseau entraîné.

On initialise le réseau avec des poids et des biais aléatoires pour chaque neurone et on les ajuste au fur et à mesure de l'entraînement. Lorsque le réseau est peu ou pas entrainé, les sorties du réseau sont loin des sorties attendues.

Les données d'entrée du réseau sont regroupées en paquets (**batch**) : chaque paquet fait l'objet d'entraînements successifs qui permettent d'améliorer la minimisation de la fonction d'erreur. Le nombre d'entraînements du réseau sur un même batch est appelé "nombre d'époques" (**n_epochs**).

3.8 Optimiseur

La recherche du minimum de la fonction d'erreur est confié à un **optimiseur** : un des algorithmes les plus simples est la *descente de gradient* (GD), qui consiste à se déplacer dans le sens de la pente la plus forte à chaque itération d'entraînement.

Adam est un optimiseur plus complexe que **GD**. Sur l'animation ci-dessous on voit plusieurs optimiseurs se déplacer sur une fonction de coût ne comportant que 2 paramètres (au lieu de plusieurs milliers...) à la recherche d'un minimum.

(Numbers in figure legend indicate learning rate, specific to each Optimizer.) Tu peux télécharger l'animation en cliquant ici.

Une autre caracteristique de l'algorithme GD, est que le pas effectué à chaque itération est fixe. L'image suivante montre Adam et GD dans un cas ou la pente devient très forte :

Tu peux télécharger l'animation en cliquant ici.