Support de cours
Apprentissage profond (Deep Learning)

Cycle Ingénieur
Filière INDIA (3ème année)

Pr. Abderrahim EL QADI

Département Mathématique appliquée et Génie Informatique
ENSAM-Université Mohammed V de Rabat

A.U. 2023/2024

Plan

1. Introduction
2. Les enjeux de l'apprentissage profond
3. Réseaux de neurones perceptron multicouches (PMC)
4. Réseaux de neurones à convolution (CNN)
5. Réseaux de neurones récurrents (RNN-LSTM)
6. Réseaux auto-encoders

Références

Ouvrages:

- Neural Networks and Deep Learning. Charu C. Aggarwal. A Textbook, 2018.
- Principles of Artificial Neural Networks, 3rd Edition, Daniel Graupe
- Les Réseaux de Neurones Artificiels. Y. Djeriri. 2017
- Des Réseaux de neurones. Eric Davalo, Patrick Naim, Edition Eyroles
- Deep Learning with Python A Hands-on Introduction. Nikhil Ketkar
- Data Science and Predictive Analytics, Biomedical and Health Applications using R. Ivo D. Dinov
- Hands-On Machine Learning with Scikit-Learn and TensorFlow Concepts, Tools, and Techniques to Build Intelligent Systems. Aurélien Géron
- Long Short-Term Memory Networks With Python Develop Sequence Prediction Models With Deep Learning. Jason Brownlee

Supports de cours:

- Réseaux de neurones formels appliques à l'intelligence artificielle et au jeu. F.TSCHIRHART
- Fonction d'activation, comment ça marche ? – Une explication simple. Tom Keldenich, 8 février 2021
- Interprétation d'images basée sur la technologie des réseaux de neurones. Flávio Barreiro Lindo. 2018
- Systèmes intelligents, chapitre 6: Réseau de neurones. Gabriel Cormier
- https://openclassrooms.com/fr/courses/5801891-initiez-vous-au-deep-learning/5814621-initiez-vous-aux-autoencodeurs (consulter 20/10/2022)

Introduction

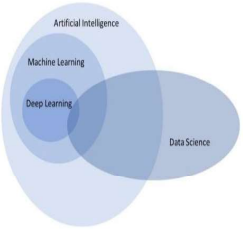
Intelligence Artificielle

Ensemble des théories et des techniques qui cherchent à développer des systèmes capables de simuler ce que les êtres humains font.

Systèmes qui conduisent la machine à prendre une décision ou à avoir le comportement attendu.

Intelligence Artificielle, un domaine très vaste et hétérogène :

- Raisonnement à base de règles, systèmes experts
- Algorithmes de jeu (échecs, Go, etc..)
- Multi-agents, émergence comportement collectif
- Optimisation, recherche opérationnelles, programmation dynamique,
- Planification (de trajectoire, de tâches, etc...)
- Apprentissage machine (statistical Machine-Learning)
- Apprentissage profond
- …



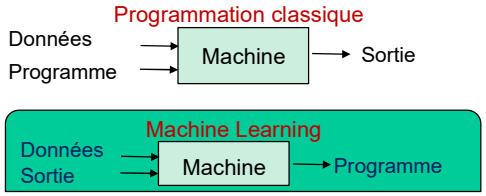
Les algorithmes de Google, Youtube, Facebook, Amazon, ou Netflix, fonctionnent tous grâce à l'IA.

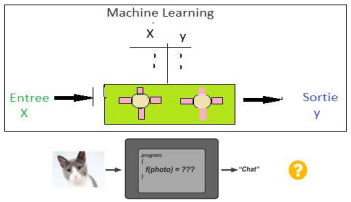
Apprentissage automatique (Machine Learning), ou apprentissage statistique

- est une discipline de l'informatique
- est une technologie d'IA basée sur le fait que la machine peut apprendre toute seule en se basant sur des modèles statistiques permettant d'effectuer des analyses prédictives.

Le ***Machine Learning***, ça consiste à laisser l'ordinateur **apprendre quel calcul effectuer, plutôt que de lui donner ce calcul** (définition de Arthur Samuel, mathématicien américain qui a développé un programme pouvant apprendre tout seul comment jouer aux Dames en 1959) (**Laisser la Machine apprendre à partir d'expériences)**

- Il consiste à écrire un programme qui au début ne sait rien faire, mais qui va **apprendre** à faire quelque chose avec le **temps** et **l'expérience**…
- Contrairement à la programmation qui consiste en l'exécution de règles prédéterminées



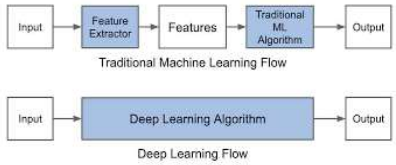


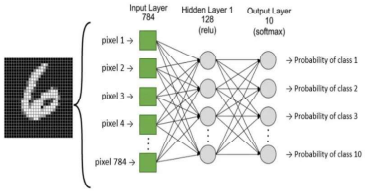
Deep Learning (Apprentissage profond)

est une discipline de l'IA capable d'analyser des données non structurées comme des vidéos, des images, des textes,…

C'est une technique de ML reposant sur le modèle des réseaux neurones: des dizaines voire des centaines de couches de neurones, chacune recevant et interprétant les informations de la couche précédente.

- Chaque couche reçoit en entrée des données et les renvoie transformées. Pour cela, elle calcule une combinaison linéaire puis applique éventuellement une fonction non-linéaire, appelée fonction d'activation. Les coefficients de la combinaison linéaire définissent les paramètres (ou poids) de la couche.
- La dernière couche calcule les probabilités finales en utilisant pour fonction d'activation la fonction logistique (classification binaire) ou la fonction *softmax* (classification multi-classes)



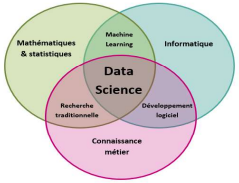


Data Science (science des données)

est un domaine qui incorpore tout ce qui est lié au nettoyage, à la préparation et à l'analyse de l'information.

C'est la discipline qui permet à une entreprise d'explorer et d'analyser les données brutes pour les transformer en informations précieuses permettant de résoudre les problèmes de l'entreprise.

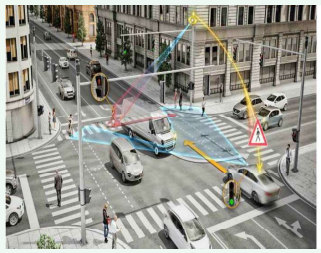
C'est est un mélange entre trois grands domaines : l'expertise mathématique, la technologie, et le business.



Les cas d'usage:

- La Data Science se cache derrière les technologies de reconnaissance faciale, vocale ou textuelle.
- Elle alimente aussi les moteurs de recommandations capables de suggérer des produits ou du contenu en fonction de préférences des utilisateurs.
- Elle permet aussi la prédiction, par exemple pour les ventes ou les revenus.

L'IA, quelques exemples d'usage

Domaine	Usage
Véhicule autonome  https://dataanalyticspost.com/le-vehicule-du-futur-verra-aussi-la-route-a-travers-les-yeux-de-linfrastructure-et-des-autres-vehicules/	l'IA motorise les véhicules autonomes via des modèles de Deep Learning. Véhicule doté des capteurs, des caméras et des radars avec une IA pour rouler sans aucun humain aux commandes. Grâce aux réseaux de neurones artificiels profonds, le véhicule « apprend » à identifier ces objets Le deep Learning permet à la voiture de s'adapter à son environnement, Par exemple le régulateur de vitesse intelligent. Cette fonctionnalité adapte l'allure de la voiture lorsqu'elle détecte un panneau de limitation

L'IA, quelques exemples d'usage

Domaine	Usage
Industrie	IA permet d'anticiper les pannes des équipements (que ce soit pour un robot sur une chaîne de montage, un serveur informatique, un ascenseur...) avant même qu'elles se produisent. Objectif : réaliser les opérations de maintenance de manière préventive. L'IA, l'IoT et l'acquisition des différentes données offrent maintenant la possibilité de mettre en place une maintenance prédictive: • Prendre des décisions immédiates et pertinentes • Mieux gérer les plannings d'interventions des techniciens • Réaliser rapidement des calculs mathématiques complexes • Prévoir à l'avance les défaillances techniques

https://www.usinenouvelle.com/article/l-industrialisation-des-donnees-est-essentielle-au-succes-de-l-industrie-4-0.N808840

L'IA, quelques exemples d'usage

Domaine	Usage
Finance-Banque	L'IA pour les banques et les institutions financières peut automatiser des tâches répétitives, améliorer l'efficacité des processus ML, le deep Learning, les analyses prédictives et le traitement du langage naturel permet d'alimenter des fonctionnalités plus robustes telles que des chabots et des robots-conseillers. Ajouter l'intelligence artificielle aider les banques et les institutions financières à améliorer leur expérience client, réduire les coûts et augmenter les revenus.

https://www.virtualagentstudio.com/fr/blog/dans-la-tete-un-chatbot-intelligence-artificielle-et-machine-learning/

L'IA, quelques exemples d'usage

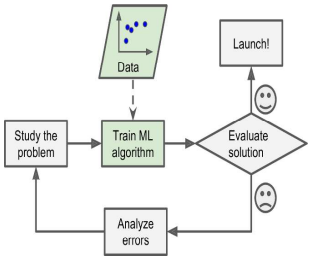
Domaine	Usage
Sport	Le deep Learning est en train de rendre tout le monde fou. Pendant longtemps la collecte de statistiques sur les matchs (nombres de passes, kilomètres parcourus, nombres de ballons touchés, etc.) étaient calculés par des humains qui suivaient les matchs chaque weekend et envoyaient les résultats à différents acteurs du foot. Aujourd'hui des systèmes de computer vision très avancés permettent de faire ce travail instantanément et avec une plus grande précision (YOLO). Le système YOLO est devenu incontournable notamment grâce à ses performances et sa rapidité. Il permet l'analyse d'images instantanément.

https://www.science-et-vie.com/technos-et-futur/euro-2020-lia-peut-elle-aider-a-gagner-une-competition-68509.html

Méthodologie pour l'apprentissage

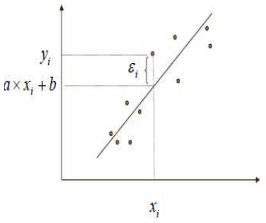
1-Données $\{x_1, ..., x_N\}$:
Comprendre le domaine d'application;
Nettoyage et prétraitement des données
2-Machine ou modèle F(x), paramétrer le modèle
3-Critère C (apprentissage et évaluation):
déterminer les paramètres généraux d'apprentissage:
- renforcement : trouver une fonction d'évaluation
- fixer les critères de réussite et de performance (taux de validation,...)

4- faire apprendre: ça peut être long ...
5- tester et valider: souvent, repartir en 3 ou en 1



Exemple d'apprentissage

Données : domaine d'application
Modèle: modèle linéaire $f(x)=ax+b$
Critère : fonction coût:
$erreur(i)= (f(x(i)) - y(i))2$
Evaluation : Trouver les paramètres a et b qui minimisent la fonction Coût



Types d'apprentissage

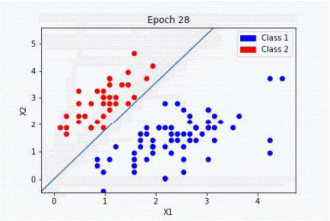
Apprentissage supervisé

Montrer à la machine ce qu'elle doit faire
C'est un Processus en deux étapes

- construction d'un modèle sur les données dont la classe est connue (training data set)
- utilisation pour classification des nouveaux arrivants

$$\begin{bmatrix} x_{11} & x_{12} \dots x_{1D} \\ x_{21} & x_{22} \dots x_{2D} \\ x_{31} & x_{32} \dots x_{3D} \\ & \dots \\ x_{N1} & x_{N2} \dots x_{ND} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \dots \\ y_N \end{bmatrix}$$

Le modèle ajuste ses paramètres de façon à **diminuer l'écart entre les résultats obtenus et les résultats attendus**

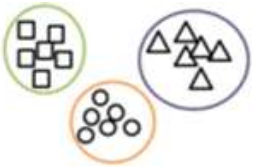


Apprentissage non supervisé

Laisser la machine apprendre toute seule
n'utilise pas de données étiquetées
Il s'agit de diviser un ensemble de données en clusters, des paquets homogènes.

- On ne connaît pas les classes à priori: uniquement des données d'entrées
- $x_1 \dots , x_n$, pas de sortie désirée associée.

$$feature\ matrix : X = \begin{bmatrix} x_{11} & x_{12} \dots x_{1D} \\ x_{21} & x_{22} \dots x_{2D} \\ x_{31} & x_{32} \dots x_{3D} \\ & \dots \\ x_{N1} & x_{N2} \dots x_{ND} \end{bmatrix}$$

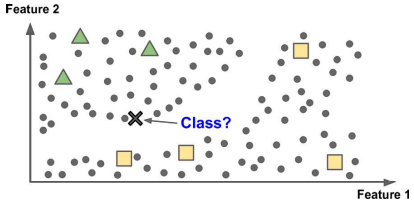


Apprentissage semi-supervisé

décrit un flux de travail spécifique dans lequel des algorithmes d'apprentissage non supervisé sont utilisés pour générer automatiquement des étiquettes, qui peuvent être introduites dans les algorithmes d'apprentissage supervisé.

Exemple de la reconnaissance d'images: les humains étiquettent manuellement certaines images, l'apprentissage non supervisé devine les étiquettes pour d'autres, puis toutes ces étiquettes et images sont introduites dans les algorithmes d'apprentissage supervisé pour créer un modèle d'IA.

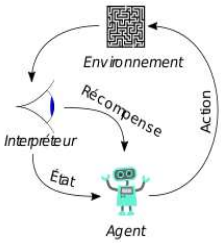
Utilisation: grandes masses de données où l'étiquetage est possible mais coûteux



Apprentissage par Renforcement

Laisser la machine génére sa propre expérience, utilisé pour apprendre à une machine à exécuter une séquence d'étapes.

- Le système d'apprentissage, appelé **agent** dans ce contexte, peut observer l'environnement, sélectionner et effectuer des actions, et obtenir des récompenses en retour (ou des pénalités sous forme de récompenses négatives)
- Il doit alors apprendre par lui-même quelle est la meilleure stratégie, appelée politique, pour tirer le meilleur parti récompense dans le temps.



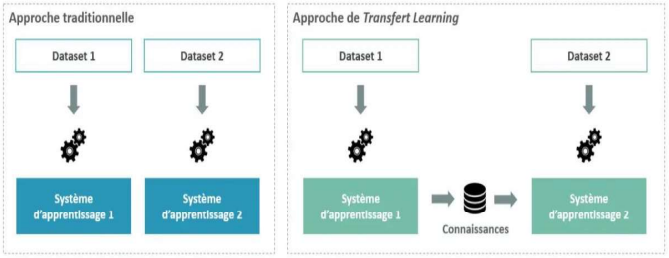
Utilisation: robotique, jeux à 2 joueurs, jeux collectifs, les voitures autonomes, les drones, la programmation dynamique, applications web ou sociales, …

De nombreux robots implémentent des algorithmes d'apprentissage par renforcement pour apprendre à marcher.

Le programme AlphaGo de DeepMind est également un bon exemple de Reinforcement Learning

Apprentissage par Transfer (Transfer Learning)

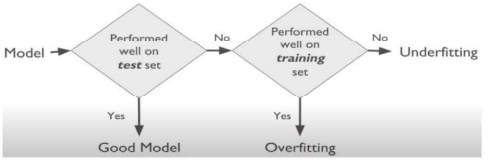
désigne l'ensemble des méthodes qui permettent de transférer les connaissances acquises à partir de la résolution de problèmes donnés pour traiter un autre problème.



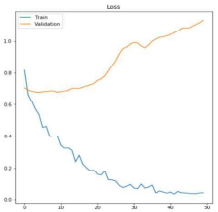
Le *Transfer Learning* a connu un grand succès avec l'essor du *Deep Learning*.
Les modèles utilisés dans ce domaine nécessitent des **temps de calcul très élevés** et des **ressources importantes**.
Or, en utilisant des modèles pré-entraînés comme point de départ, le *Transfer Learning* permet de développer rapidement des modèles performants et résoudre efficacement des problèmes complexes en *Computer Vision* ou *Natural Language Processing*, *NLP*

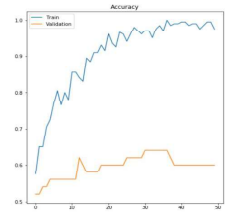
Problème de sur-ajustement (overfitting)

Il fait référence au fait que l'ajustement d'un modèle à un ensemble de données d'apprentissage particulier ne garantit pas qu'il fournira de bonnes performances de prédiction sur des données de test, même si le modèle prédit parfaitement les cibles sur les données d'apprentissage.

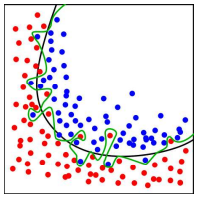


La variance élevée des performances du modèle est un indicateur d'un problème de sur-ajustement.





Exemple



La ligne verte classifie trop parfaitement les données d'entrainement, elle généralise mal et donnera de mauvaises prévisions futures avec de nouvelles données. **Le modèle vert est donc au final moins bon que le noir.**

Problème de sur-ajustement (overfitting)

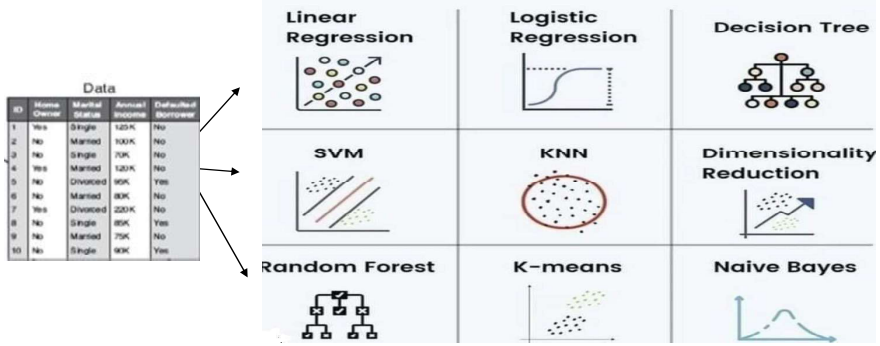
Le sur-ajustement se produit lorsque :

- Les données d'apprentissage ne sont pas nettoyées et contiennent des valeurs parasites.
- Le modèle capture le bruit dans les données d'apprentissage et ne parvient pas à généraliser l'apprentissage du modèle.
- Le modèle a une variance élevée.
- La taille des données d'apprentissage n'est pas suffisante et le modèle s'entraîne sur les données d'apprentissage limitées pendant plusieurs époques.
- L'architecture du modèle comporte plusieurs couches neuronales empilées. Les réseaux de neurones profonds sont complexes et nécessitent beaucoup de temps pour s'entraîner, et conduisent souvent à un sur-ajustement de l'ensemble d'apprentissage.

- La validation croisée K-fold est l'une des techniques les plus populaires couramment utilisées pour détecter le sur-ajustement.

Le modèle est formé sur un échantillon limité pour estimer comment le modèle devrait fonctionner en général lorsqu'il est utilisé pour faire des prédictions sur des données non utilisées pendant la formation du modèle.
Un fold agit comme un ensemble de validation à chaque étape.

Taxonomie des algorithmes d'apprentissage



Les algorithmes de régression (quand la valeur à prédire est continue)

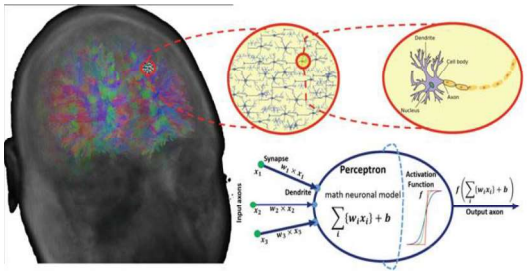
Les algorithmes de classification (quand la valeur à prédire est catégorielle).

Les enjeux de l'apprentissage profond
Réseau de neurones

Un **réseau de neurones** est un modèle de raisonnement basé sur le cerveau humain.
Le cerveau est constitué d'un ensemble de cellules nerveuses, ou unités de traitement d'information, appelés neurones.

Le cerveau contient environ 100 milliards de neurones, 60 trillions de connexions, des synapses, entre eux.

Un neurone est **une cellule particulière**. Elle possède des extensions par lesquelles elle peut distribuer des signaux (**axones**) ou en recevoir (**dendrites**).



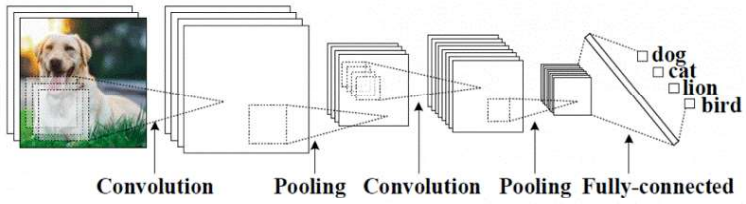
Les réseaux de neurones sont des **imitations simples des fonctions d'un neurone dans le cerveau humain** pour résoudre des problématiques d'apprentissage de la machine (Machine Learning)

Le neurone est une unité qui est exprimée généralement par une fonction sigmoïde: $f(x) = \frac{1}{1+e^{-x}}$

Domaines d'application

Les domaines d'application des réseaux neuronaux sont souvent caractérisés par une relation entrée-sortie de la donnée d'information :

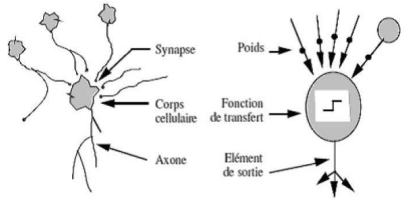
- La reconnaissance d'image
- Les classifications de textes ou d'images
- Identification d'objets
- Prédiction de données
- Filtrage d'un set de données



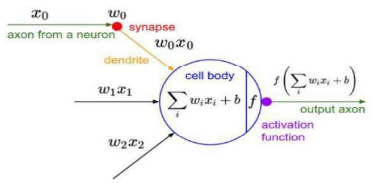
Exemple : Réseau de neurones classique pour la reconnaissance d'image.

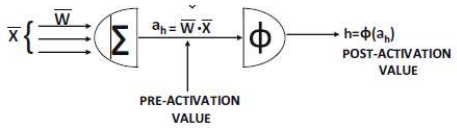
Neurone formel

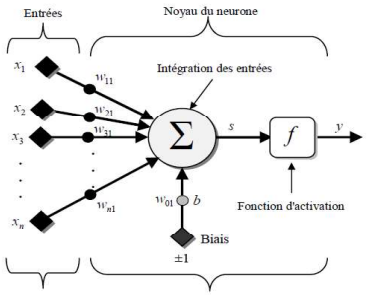
Un **"neurone formel"** (ou simplement "neurone") est une **fonction algébrique,** fait une somme pondérée des potentiels d'actions qui lui parviennent, puis s'active suivant la valeur de cette sommation pondérée.
Si cette somme dépasse un seuil, le neurone est activé.



Neurone biologique	Neurone formel
Synapses	Poids des connexions
Axones	Signal de sortie
Dendrites	Signal d'entrée
Noyau ou Somma	Fonction d'activation







$x = [x_1, x_2, x_3 \ldots x_n]^T$

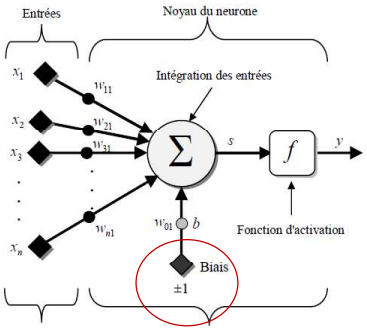
$w = [w_{11}, w_{21}, w_{31}, \ldots w_{n1}]^T$

$s = \sum_{i=1}^{n} w_{i1} x_i \pm b$

x_i représentent les vecteurs d'entrées, proviennent soit des sorties d'autres neurones, soit de stimuli sensoriels (capteur visuel, sonore…) ;
Elle doivent être normalisées;

w_{ij} sont les poids synaptiques du neurone j;
$w_{ij} > 0$: synapse excitatrice;
$w_{ij} < 0$: synapse inhibitrice.
Ces poids pondèrent les entrées et peuvent être modifiés par apprentissage ;

Noyau : intègre toutes les entrées et le biais et calcul la sortie du neurone selon une fonction d'activation



Biais : prend souvent les valeurs -1 ou +1 ; permet d'ajouter de la flexibilité au réseau en permettant de varier le seuil de déclenchement du neurone par l'ajustement des poids et du biais lors de l'apprentissage;

Pourquoi ajoutons-nous un biais?

Supposons que si nous avons des valeurs d'entrée (0,0), la somme des produits des nœuds d'entrée et des poids sera toujours nulle.
Dans ce cas, la sortie sera toujours nulle, peu importe combien nous entraînons notre modèle.

Pour résoudre ce problème et faire des prédictions fiables, nous utilisons le terme de biais.

Le biais est nécessaire pour faire un réseau de neurones robuste.

$s = w^T x \pm b$

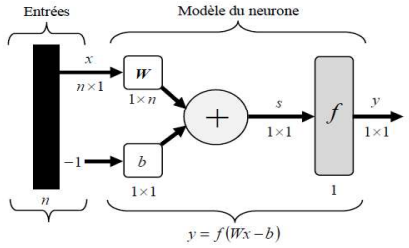
- La **sortie s** correspond à une somme pondérée des poids et des entrées plus ce qu'on nomme **le biais *b* du neurone.**
- ***Le résultat s*** *de la somme pondérée s'appelle le niveau d'activation du* neurone.
- Le biais *b s'appelle aussi* ***le seuil d'activation du neurone.***
- *Lorsque le niveau* d'activation atteint ou dépasse le seuil *b , alors l'argument de f devient positif (ou nul). Sinon, il* est négatif.

$y = f(s) = f(w^T x \pm b)$

En remplaçant w^T *par une matrice* $W = w^T$ *d'une seule ligne, on obtient :* $y = f(Wx \pm b)$

Un réseau de neurone ne se programme pas, il est entraîné grâce à un mécanisme d'apprentissage.

L'information est stockée de manière distribuée dans le réseau sous forme de coefficients synaptiques ou de fonctions d'activation

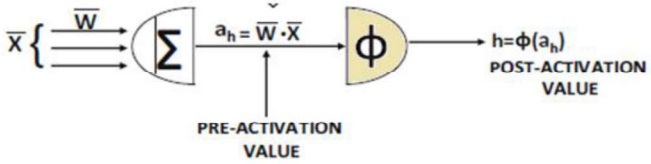


Fonctions d'activation

se sont des fonctions mathématiques, et sont souvent **bornées** i.e. elles ne peuvent pas dépasser une certaine valeur : cela évite les grands calculs et la mémoire consommée.
Par exemple, si on donne la valeur 100 à la fonction d'activation tanh on obtient la nouvelle valeur 0.999 : on a, pour tout réel x, -1 < tanh(x) < 1.

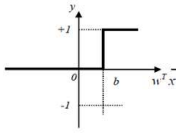
Selon le problème à résoudre (classification, régression, …) on utilise des fonctions d'activations différentes.

Pour choisir la bonne fonction d'activation il faut à la fois considérer la transformation direct qu'elle applique aux données mais aussi sa dérivé qui sera utilisé pour ajuster les poids lors de la back propagation.

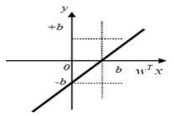


Fonction (f)	Relation entrée/sortie	Intervalle de définition
Seuil (Sign) (Heaviside)	$s = w^T x \pm b$) y =f(s) y = 1 si s ≥ 0	[0,+1[
Linéaire (Identity)	y = s	(-∞;+∞)
Sigmoïde (Sigmoid (logistic)	y=1 / (1 + exp(-s))	(0,1)
Tangente hyperbolique (TanH)	y = (e^s – e^{-s}) / (e^s + e^{-s})	(-1,+1)
Exponentielle normalisée (Softmax)	y=exp(s) / sum(exp(s_i))	(-∞;+∞)
ReLU (Rectified Linear Unit)	y=max(s, 0)	(0;+∞)
Exponential Linear Unit (ELU)	y=s si s > 0 y= alpha*(exp(s) – 1) si s < 0 (alpha > 0)	(-∞;+∞)

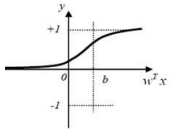
La fonction seuil (hard limit) applique un seuil sur son entrée.
- une entrée négative ne passe pas le seuil, la fonction retourne alors la valeur 0 (on peut interpréter ce 0 comme signifiant faux),
- une entrée positive ou nulle dépasse le seuil, et la fonction retourne à 1 (vrai).



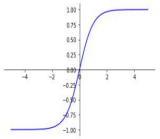
La fonction linéaire affecte directement son entrée à sa sortie: y = s



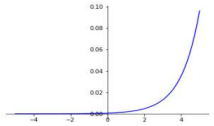
La fonction **de transfert sigmoïde** (fonction non linéaire) ressemble soit à la fonction seuil, soit à la fonction linéaire, selon que l'on est loin ou près de *b , respectivement*



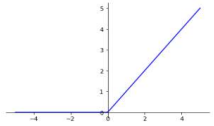
La fonction **Tanh** (fonction non linéaire) comme Sigmoïde, utilisée dans la classification binaire. Il s'agit en fait d'une version mathématiquement décalée de la fonction sigmoïde :
sigmoïde donne un résultat entre 0 et 1
tanh donne un résultat entre -1 et 1



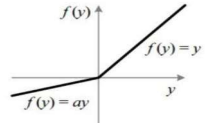
La fonction exponentielle normalisée (Softmax) permet de transformer un vecteur réel en vecteur de probabilité.
On l'utilise souvent dans la couche finale d'un modèle de classification, notamment pour les problèmes multi classes.



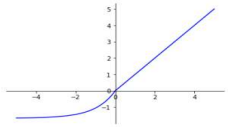
La fonction **ReLU** (Rectified Linear Unit) est la fonction d'activation la plus simple et la plus utilisée (Filtre les valeurs négatives).
Très utilisé pour les CNN, RBM, et les réseaux de multi perceptron.



La fonction Leaky ReLU permet d'ajouter une variante pour les nombres négatifs, ainsi les neurones ne meurent jamais. Ils entrent dans un long coma mais on toujours la chance de se réveiller à un moment donné.



La fonction ELU (Exponential Linear Unit) dérivée de la ReLU.
- Celle-ci va approcher les valeurs moyenne proche de 0, ce qui va avoir comme impact d'améliorer les performances d'entrainements.
- Elle utilise exponentiel pour la partie négative. Elle parait plus performante en expérimentation que les autres Relu.



Architecture des réseaux de neurones

Les réseaux de neurones diffèrent par plusieurs paramètres dont les principaux sont :
- La topologie des connexions entre les neurones du réseau.
- La fonction d'activation des neurones.
- Le mode d'apprentissage (supervisé ou non).
- L'algorithme d'apprentissage.

Les architectures ont leurs forces et faiblesses et peuvent être combinées pour optimiser les résultats. Le choix de l'architecture s'avère ainsi crucial et il est déterminé principalement par l'objectif.

Exemples de réseaux de neurones:

• Les Perceptron multicouches (multilayer perceptron *MLP*) sont utilisé pour prédire une variable cible continue ou discrète; Bon pour la classification, et très bon pour l'approximation de fonctions.
• Les réseaux neuronaux convolutifs (Convolutional Neural Networks CNN): ensemble de réseaux de neurones distincts dédiés à traiter chacun une partie de l'information (par exemple le traitement d'image, de vidéos, de textes).
•Les réseaux de Neurones récurrents (RRN) traitent l'information en cycle. Ces cycles permettent au réseau de traiter l'information plusieurs fois en la renvoyant à chaque fois au sein du réseau.

Caractéristiques fonctionnelles	Type de RNA
Traitement d'images	CNN, Hopefield (le réseau de neurones récurrent d'une seule couche)
Classification et clustering	MLP, Kohonen, RBF, ART, PNN
Reconnaissance de formes	MLP, Hopefield, Kohonen, PNN
Mémoires associatives	Hopefield, MLP recurents, Kohonen
Modélisation et control	MLP, MLP recurent

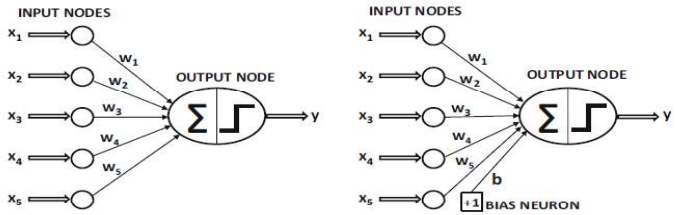
- Traitement d'images : reconnaissance de caractères et de signatures, compression, reconnaissance de forme, classification, etc.
- Traitement du signal : filtrage, classification, identification de source, traitement de la parole…etc.
- Contrôle : commande de processus, diagnostic, contrôle qualité, asservissement des robots, systèmes de guidage automatique des automobiles et des avions…etc.
- Optimisation : planification, allocation de ressource, gestion et finances, etc.

Modèle de Perceptron : Réseau à apprentissage supervisé sans rétro propagation

• Le perceptron inventé **en 1957 par Franck Rosenblatt**, est considéré comme étant le réseau de neurones le plus simple.

• C'est un réseau de neurones sans aucune couche cachée. Un perceptron n'a qu'une couche d'entrée et une couche de sortie.

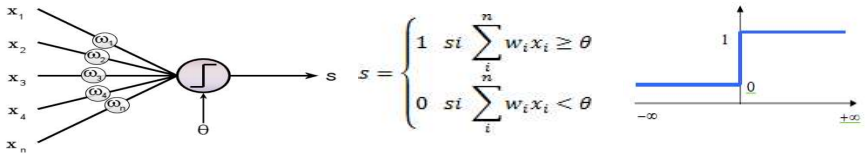
• C'est un algorithme d'apprentissage automatique supervisé de classification binaire.



(a) Perceptron without bias (b) Perceptron with bias

Perceptron monocouche

• Le Perceptron utilise la fonction de Heaviside (seuil) comme fonction d'activation, il calcule sa sortie de la façon suivante:



$$s = \begin{cases} 1 & si \sum_i^n w_i x_i \geq \theta \\ 0 & si \sum_i^n w_i x_i < \theta \end{cases}$$

Dans le cas du perceptron, le choix de la fonction d'activation de signe est motivé par le fait **qu'une étiquette de classe binaire** doit être prédite.

$$\hat{y} = sign\{\bar{W}.\bar{X} + b\} = sign\{\sum_{j=1}^{n} w_j x_j + b\}$$

Si la **variable cible à prédire est réelle**, il est logique d'utiliser la fonction d'activation d'identité et l'algorithme résultant est le même que la régression des moindres carrés.

S'il est souhaitable de prédire une probabilité d'une **classe binaire (P(y/X)**, il est logique d'utiliser une fonction sigmoïde pour activer le nœud de sortie, de sorte que la prédiction ˆy indique la probabilité que la valeur observée, y, de la dépendance est 1.

y

Apprentissage du Perceptron

L'apprentissage d'un neurone se fat par l'optimisation d'une fonction de perte.
On cherche les paramètres W et b qui minimisent la fonction de perte.

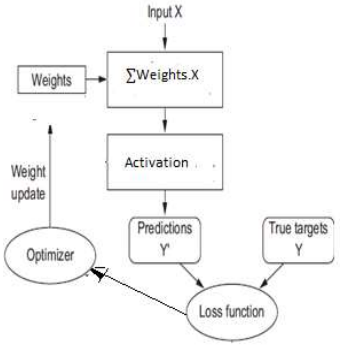
Règle d'apprentissage:

Modifier les poids : $\omega_i(t+1) = \omega_i(t) + \alpha X_i(t) * e(t)$

$\omega_i(t+1) \leftarrow \omega_i(t) + \Delta\omega_i$

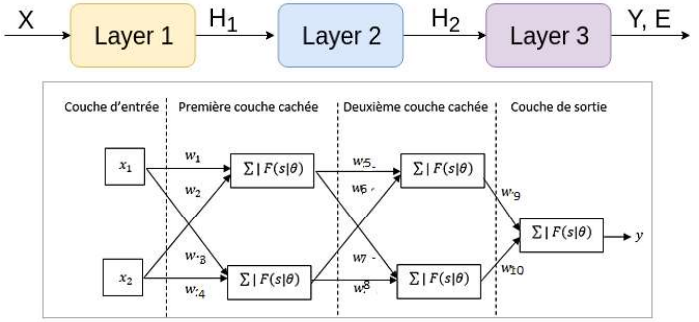
- X représente les entrées du réseau
- e(t)= y' - y ou t=1,2,3…
- y' la sortie désirée
- y la sortie calculée par le Perceptron à l'itération t
- α est le taux d'apprentissage, (0<α<1), détermine l'amplitude de l'apprentissage;

Quelle est la bonne valeur ?
Trop petit →lenteur de convergence
Trop grand →oscillation
en général autour de 0.05



Les réseaux à couches cachées

On augmente le pouvoir de prédiction en ajoutant une ou plusieurs couches cachées entre les couches d'entrée et de sortie



Le pouvoir de prédiction augmente avec le nombre de nœuds des couches cachées
• le nombre de couches cachées est très généralement 1 ou 2
• lorsque ce nombre = 0, le réseau effectue une régression linéaire ou logistique

Les réseaux à couches cachées

Il existe des termes utilisés pour décrire la forme et la capacité d'un réseau de neurones :

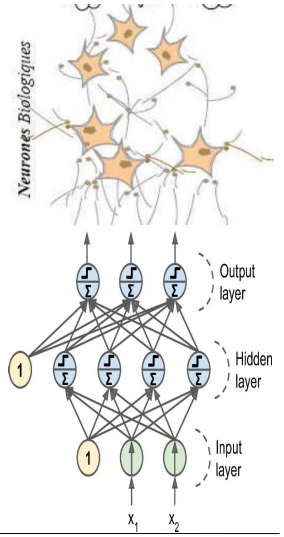
- Taille : nombre de nœuds dans le modèle.
- Largeur : le nombre de nœuds dans une couche spécifique.
- Profondeur : le nombre de couches dans un réseau de neurones.
- Capacité : Le type ou la structure des fonctions qui peuvent être apprises par une configuration réseau. Parfois appelée «capacité de représentation»
- Architecture : La disposition spécifique des couches et des nœuds dans le réseau.

Perceptron multicouche : Rumelhart et McClelland (1986): Réseau à apprentissage supervisé avec rétro propagation

Les *Perceptrons multicouches* (*PMC)* constituent l'architecture la plus fréquemment utilisée actuellement.

Chaque neurone va en fait calculer une somme pondérée de ses entrées qu'il va transmettre à une fonction de transfert *f* afin de produire ses sorties.

Un Perceptron multicouche peut posséder un nombre de couches quelconque et chaque couche peut comporter un nombre de neurones (ou d'entrées, si il s'agit de la couche d'entrée) également quelconque.



La dernière couche calcule les probabilités finales en utilisant pour fonction d'activation la fonction logistique (classification binaire) ou la fonction *softmax* (classification multi-classes).

Une combinaison de séparateurs linéaires permet de produire un séparateur global non-linéaire (Rumelhart, 1986).

Apprentissage de PMC

Un réseau neuronal s'exécute en 2 étapes:

1- Feed-forward (Réseaux de Neurones Non-Bouclés):
Propagation avant (Feed-fowarded) signifie tout simplement que la donnée traverse le réseau d'entrée à la sortie sans retour en arrière de l'information.

2- La technique de **rétro propagation** (*Back propagation*) *est une méthode qui permet de calculer le* ***gradient de l'erreur*** *pour chacun des neurones du réseau, de la dernière couche vers la première.*

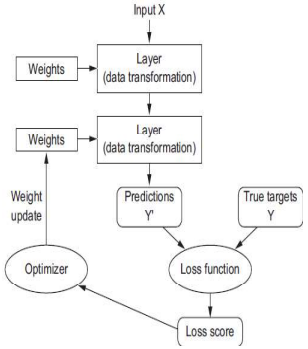
Apprentissage de PMC

Une fonction de perte (***loss function***) est associée à la couche finale pour calculer l'erreur de classification.
Il s'agit en général de l'**entropie croisée.**
On cherche les paramètres W et b qui minimisent la fonction de perte.
Dans le cas de la classification, on utilise la log-vraisemblance négative:

$$L(\hat{y} - y) = -y.\log(\widehat{y}) + (1-y).\log(1-\hat{y})$$

Quand tout est dérivable, on peut utiliser la descente de gradient.
A chaque étape, les paramètres sont déplacés de la manière suivante:

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y}\acute{f}(< w, x > + b)x_i$$



Le score de perte est utilisé comme signal de rétroaction pour ajuster les poids

Gradient Descent (la descente de gradient)

est un algorithme d'apprentissage automatique qui fonctionne de manière itérative pour trouver les valeurs optimales de ses paramètres.
Il prend en compte le taux d'apprentissage défini par l'utilisateur et les valeurs initiales des paramètres.

Fonctionnement: (itératif)
1. Initialiser les paramètres w
2. Calculer le coût.
3. Mettre à jour les valeurs de w: $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{w}}$
4. Convergence jusqu'a $\|\Delta_w\| \approx 0$

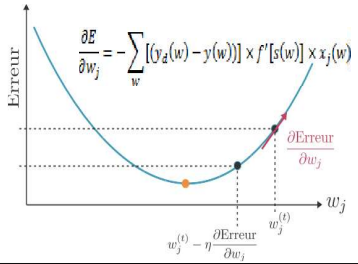
le gradient donne la direction de plus grande variation d'une fonction (fonction d'erreur), pour trouver le minimum de cette fonction il faut se déplacer dans la direction opposée au gradient. (Lorsque la fonction est minimisée localement, son gradient est égal à 0.)

Critère à optimiser: critère des moindres carrés $E = \frac{1}{2}\sum_w (y_d(w) - y(w))^2$

Optimisation: dérivation de la fonction objectif

$$w_j \leftarrow w_j - \alpha(y_d - y_c) f'(s) x_j$$

Gradient: maj des poids dans la direction qui minimise E



Fonction de perte (Loss Function)

Le choix de la fonction de perte est essentiel pour définir les sorties d'une manière sensible à l'application en cours.

Par exemple, la régression des moindres carrés avec des sorties numériques nécessite une simple perte au carré de la forme $(y - y')^2$ pour une seule instance d'apprentissage avec la cible y et la prédiction y'.

Pour les prédictions probabilistes, deux types différents de fonctions de perte sont utilisés, selon que la prédiction est binaire ou multidirectionnelle :

- Binary targets (régression logistique) : la fonction de perte pour une seule instance avec la valeur observée y et la prédiction à valeur réelle y' (avec activation d'identité) est définie comme suit :
  $L = \log(1 + \exp(-y \cdot y'))$
- Categorical targets: Dans ce cas, si $y'_1 \ldots y'_k$ sont les probabilités des k classes (en utilisant l'activation softmax), alors la fonction de perte pour une seule instance est définie comme suit :
  $L = -\log(y'_r)$
  Ce type de fonction de perte implémente une régression logistique multinomiale, et on l'appelle la **perte d'entropie croisée**.

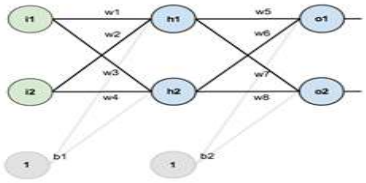
Rétro-propagation

On a un ensemble de fonctionnalités d'entrées et des pondérations aléatoires que nous optimiserons en utilisant la propagation vers l'arrière (back-propagation).

Il y a 2 problèmes majeurs rencontrés lors de la rétropagation de l'erreur :

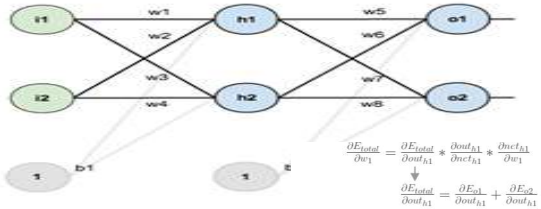
- Il se peut que les gradients d'erreurs disparaissent, c'est-à-dire qu'au fur et à mesure que l'information progresse dans les couches du réseau, les gradients deviennent très petits et par conséquent, la mise à jour des poids de connexion n'est presque plus effectuée. Alors, le modèle ne converge jamais vers une bonne solution.
- Ou le contraire, les gradients d'erreurs deviennent de plus en plus grands et explosent. Alors, les poids de connexion deviennent très grands à leur tour et le modèle diverge.

$$w \leftarrow w - \alpha \frac{\partial E}{\partial w}$$



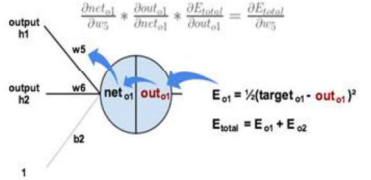
Rétro-propagation

$$w \leftarrow w - \alpha \frac{\partial E}{\partial w}$$



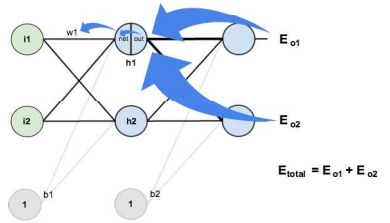
$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$

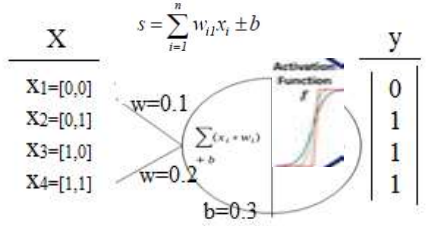
$$\frac{\partial E_{total}}{\partial w_5} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * out_{h1}$$



$$\frac{\partial E_{total}}{\partial w_1} = \left(\sum_o \frac{\partial E_{total}}{\partial out_o} * \frac{\partial out_o}{\partial net_o} * \frac{\partial net_o}{\partial out_{h1}}\right) * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = \left(\sum_o \delta_o * w_{ho}\right) * out_{h1}(1 - out_{h1}) * i_1$$

Rétro-propagation : Exemple 1



f(s)=1 / (1 + exp(-s))
δf(s)=f(s)*(1-f(s))
erreur=f(s)-y
lr=0.05
deriv = δf(s) * error
deriv_final = np.dot(inputsT,deriv)
w = w - lr * deriv_final
bias = bias - lr * i (for i in deriv)

$y_1=f(s_1)=f((0*0.1+0*0.2)+0.3)=f(0.3)=0.574 \rightarrow error1=y_{1d}-y_1=0.574$

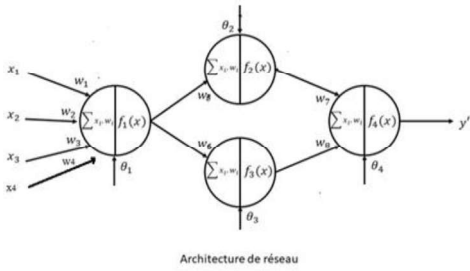
$$w_j \leftarrow w_j - \alpha(y_d - y_c)f'(s)x_j$$

Pré-activation (s)	Activation-formule (f(s))	Back-propogation (Erreur)	Dérivé (δf(s))
0.3	0.574	0.574	0.230
0.5	0.622	-0.377	0.227
0.4	0.598	-0.401	0.228
0.6	0.645	-0.354	0.225

Poids	
0.085	0.185
0.108	0.208
0.108	0.208
0.108	0.208

Bias
0.272
0.317
0.317
0.317

Rétro-propagation: Exemple 2



$$w \leftarrow w - \alpha\frac{\partial E}{\partial w}$$ f'(x)=f(x)*(1-f(x)

$$w_j \leftarrow w_j - \alpha(y_d - y_c)f'(s)x_j$$

$$w_7 = w_7 - (\eta.\frac{\partial E}{\partial w_7})$$

$$\frac{\partial E}{\partial w_7} = \frac{\partial E}{\partial y'}.\frac{\partial y'}{\partial s_4}.\frac{\partial s_4}{\partial w_7}$$
$$\frac{\partial E}{\partial w_7} = (y' - y).(\frac{1}{1+e^{-s4}}.(1 - \frac{1}{1+e^{-s4}})).(f_4(x))$$

$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial y'}.\frac{\partial y'}{\partial s_4}.\frac{\partial s_4}{\partial f_2(x)}.\frac{\partial f_2(x)}{\partial s_2}.\frac{\partial s_2}{\partial f_1(x)}.\frac{\partial f_1(x)}{\partial s_1}.\frac{\partial s_1}{\partial w_1}$$

Le taux d'apprentissage α=0.05, et la sortie désirée y=1

θ_1	θ_2	θ_3	θ_4
1	1	1	1

Entrée	Poids								Sorties				
x_1=1, x_2=1, x_3=0, x_4=0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$	$f_5(x)$= y'
Propagation avant	0,5	-0,5	0,2	-0,5	0,2	0,5	-0,2	0,5	0.731	0.758	0.796	0.776	0.776
Rétro-propagation	0.502	-0.497	0,2	-0,5	0,2	0,501	-0,2	0,5					

Rétro-propagation : Codage en python

```
# Sigmoid function:
def sigmoid(x):
    return 1/(1+np.exp(-x))
 # Derivative of sigmoid function:
def sigmoid_der(x):
    return sigmoid(x)*(1-sigmoid(x))
```

f(s)=1 / (1 + exp(-s))
δf(s)=f(s)*(1-f(s))
erreur=f(s)-y
lr=0.05
deriv = δf(s) * error
deriv_final = np.dot(inputsT,deriv)
w = w - lr * deriv_final
bias = bias - lr * i (for i in deriv)

```
for epoch in range(10000):
    inputs = X
    s = np.dot(inputs, weights) + bias          #Feedforward input
    fs = sigmoid(s)                             #Feedforward output
   #Backpropogation
    error = fs – y                              #Calcul de l'erreur
    deriv = error * sigmoid_der(s)  #Multiplication individual derivation
    #Multiplication avec le 3rd individual derivation:
    inputs = X^T                                #le transposé de X
    deriv_final = np.dot(inputs, deriv)
    weights -= lr * deriv_final                 #Modification des valuers de poids
    for i in deriv:                             # Modification la valuer de bias
        bias -= lr * i
```


PMC : Codage avec modèle MLPClassifier

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer ( )
# PMC avec trois couches cachées, chacune comportant 10 neurones.
from sklearn.neural_network import MLPClassifier
mlp = MLPClassifier (hidden_layer_sizes =(30 ,30 ,30) )
mlp.fit (X_train , y_train )
```

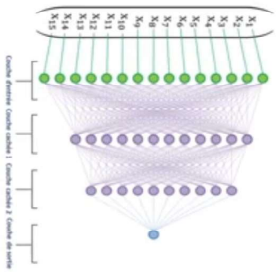
```
MLPClassifier(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9,
       beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=(30, 30, 30), learning_rate='constant',
       learning_rate_init=0.001, max_iter=200, momentum=0.9,
       n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
       random_state=None, shuffle=True, solver='adam', tol=0.0001,
       validation_fraction=0.1, verbose=False, warm_start=False)
```

Chacun des paramètres apparaissant dans le retour de l'appel de la fonction mlp.fit peut être précisé lors de l'appel de la fonction MLPClassifier.

PMC : Codage avec Tensorflow et Keras

Keras permet de créer un réseau de neurones vide à l'aide de la méthode Sequential et d'ajouter ensuite des couches à l'aide de la méthode add.

Python
- Tensorflow 2.0
- Keras



```
import numpy as np
import matplotlib.pyplot as plt

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# charger le jeu de données
dataset=np.loadtxt("pair_impair.csv", delimiter=",")
x=dataset[:,0:15]
y=dataset[:,15]

# définir le modèle keras
model = Sequential()
```

ajouter une couche cachée Fully Connected de 12 neurones qui reçoit 15 entrées et une fonction d'activation ReLU.

Couche cachée ReLU avec 10 neurones .

Le modèle se termine par une couche sigmoid. On a un seul neurone en sortie.

```
# définir le modèle keras
model = Sequential()
model.add(Dense(12, input_dim=15, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# compiler le modèle keras
model.compile(loss='binary_crossentropy', optimizer='adam',
              metrics=['accuracy'])

# fit le modèle keras sur le dataset
history = model.fit(x, y, validation_split=0.25, epochs=60, batch_size=1)
```

la fréquence d'ajustement des poids du réseau

Exemples
Apprentissage : 75%, Test : 25%

```
# evaluate the keras model
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title("Model's Training & Validation loss across epochs")
plt.ylabel('Loss')
```

Les couches cachées

Le nombre de couches cachées à utiliser dépend du problème à traiter.
Si le jeu de données à traiter est simple, deux couches cachées ou moins, sont souvent suffisantes.

Le nombre de neurones d'une couche cachée doit rester entre le nombre de neurones de la couche d'entrée et du nombre de neurones de la couche de sortie.

Utiliser une expérimentation systématique pour découvrir ce qui fonctionne le mieux pour votre ensemble de données spécifique.

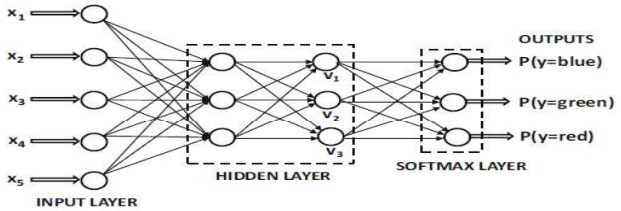
Certaines stratégies de recherche populaires incluent :

- Aléatoire : essayer des configurations aléatoires de couches et de nœuds par couche.
- Grille : essayer une recherche systématique sur le nombre de couches et de nœuds par couche.
- Heuristique : essayer une recherche dirigée dans des configurations telles qu'un algorithme génétique ou une optimisation bayésienne.
- Exhaustif : essayer toutes les combinaisons de couches et le nombre de nœuds ; cela pourrait être faisable pour les petits réseaux et ensembles de données.

Les couches cachées

Le choix et le nombre de nœuds de sortie sont également liés à la fonction d'activation, qui à son tour dépend de l'application en cours.

Par exemple, si une classification k-way est prévue, k valeurs de sortie peuvent être utilisées, avec une fonction d'activation softmax par rapport aux sorties $v = [v_1, \ldots, v_k]$ aux nœuds d'une couche donnée.



Les couches cachées

Le nombre de couches et le nombre de nœuds dans chaque couche sont des hyperparamètres du modèle qui devront être spécifier.

Keras Tuner : Il s'agit d'un cadre d'optimisation d'hyperparamètres qui aide à sélectionner les hyperparamètres optimaux.
On passe une gamme de couches (comme 2 à 20) au réseau de neurones avec les unités avec une taille de pas.
On peut également passer différents taux d'apprentissage.
L'algorithme RandomSearch entraînera le modèle sur toutes les combinaisons possibles et donnera le meilleur résultat.

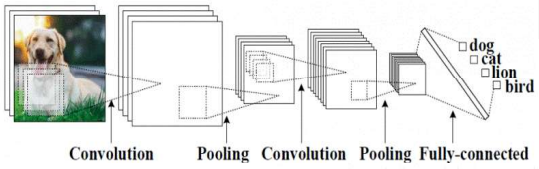
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Dropout
from tensorflow.keras.optimizers import Adam
import keras_tuner
def build_model(hp):
    model = Sequential()
    model.add(Dense(
        units=hp.Int("units", min_value=8, max_value=32, step=4),
        # hp.Choice('units', [8, 12, 32]), activation='relu',input_dim=8)
        )
    model.add(Dense(2, activation='softmax'))
    model.compile(optimizer='adam',loss="binary_crossentropy",
                  metrics=["accuracy"])
    return model
tuner=keras_tuner.RandomSearch( hypermodel=build_model,
        objective='val_acc', max_trials=5, directory='my_dir',
        executions_per_trial=2, overwrite=True  )
tuner.search(X_train, y_train, epochs=5,
validation_data=(X_test,y_test))
best_model=tuner.get_best_models()[0]
best_model.summary()
tuner.results_summary()
```

Réseau de neurones à convolution (CNN)

Les réseaux de neurones à convolution ont été créés dans les années 80 (ujjwalkarn, 2016).
L'idée qui a mené à sa création est le fait d'essayer de faire ressortir certaines parties de l'image qui pourraient être intéressantes et les analyser, indépendamment de leur position dans l'image.

Les CNN ont une méthodologie similaire à celle des méthodes traditionnelles d'apprentissage supervisé : ils reçoivent des images en entrée, détectent les *features* de chacune d'entre elles, puis entraînent un classifieur dessus.

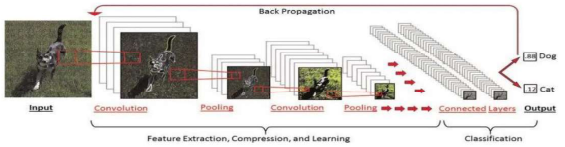
Un réseau de neurones à convolution est essentiellement composé de 4 parties :
• **Convolution** : opère une convolution sur le signal d'entrée en associant une réduction de dimension
• **Non-linéarité (ReLu)**
• **Pooling** : réduction de dimension en remplaçant un sous-ensemble des entrées par une valeur
• **Couche Fully Connected (FC)**



Les **CNN** sont particulièrement utilisés pour le **traitement d'image, de vidéo et la reconnaissance/détection d'objet** mais ils peuvent aussi s'appliquer à d'autres types de données.

Dans le cas d'une classification d'images:

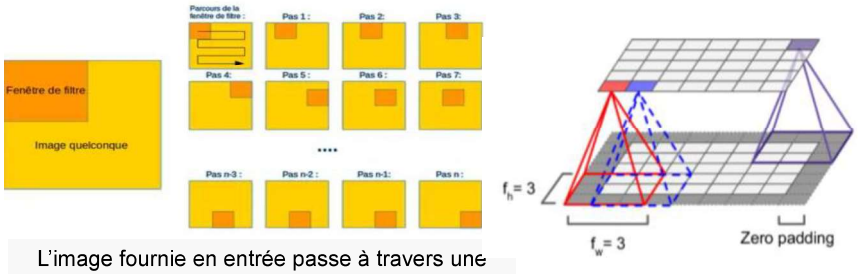
- l'utilisateur fournit en entrée **une image sous la forme d'une matrice de pixels**. Celle-ci dispose de 3 dimensions :
  - **Deux dimensions** pour une image en niveaux de gris.
  - Une **troisième dimension**, de profondeur 3 pour représenter les couleurs fondamentales (**Rouge, Vert, Bleu**).
- L'extraction des caractéristiques et la segmentation de l'image en séquence d'entrée RGB se fait automatiquement à travers les couches de convolution et de pooling (Partie Convolution)
- L'application de la *fonction softmax* aux sorties de la couche fully connection, nous donne la probabilité d'une classe (Partie Classification).



Convolution

- D'un point de vue simpliste, est le fait d'appliquer un filtre mathématique à une image.
- D'un point de vue plus technique, il s'agit de faire glisser une matrice par-dessus une image, et pour chaque pixel, utiliser la somme de la multiplication de ce pixel par la valeur de la matrice.

Cette technique permet de trouver des parties de l'image qui pourraient être intéressantes.



L'image fournie en entrée passe à travers une **succession de filtres de convolution**.

Convolution : Exemple

Exemple de valeurs des pixels d'une image 5x5

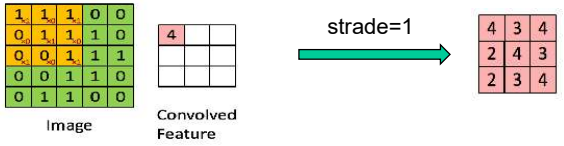
1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

×

1	0	1
0	1	0
1	0	1

Exemple de valeurs d'une matrice utilisée comme filtre de taille 3×3

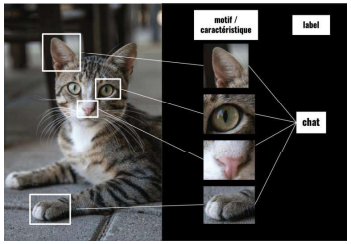
chaque valeur des pixels de l'image est multipliée par chaque valeur correspondante du filtre



4	3	4
2	4	3
2	3	4

Convolution : Exemple

Les couches de convolution analyse l'image par zone.
Elles se focalisent sur **chaque partie** de la donnée.
Elles **extraient les motifs**, **les tendances d'une donnée.**



La **première couche** apprendra **les petits motifs**, une **deuxième** apprendra **des motifs plus importants** constitués des caractéristiques des premières couches, etc.

Exemple : On veut **repérer** si **un chat** se trouve **sur une image**.
Avec **des couches de convolution**, le modèle de Deep Learning détecte les **zones de l'image** qui ressemble à un chat. Ainsi il peut **repérer les endroits** où se trouve:
- des oreilles
- un museau
- des pattes

Le **modèle garde ces caractéristiques en mémoire** et comprend qu'ils **représentent le label** "chat".
Par la suite, il **comprend facilement** si un chat se trouve sur une image **en recherchant ces caractéristiques**.

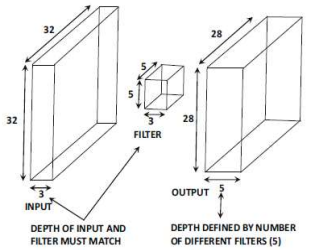
Convolution : Filtre (noyau)

Un réseau de neurones à convolution contient de multiples filtres et ces filtres sont appliqués sur l'image d'origine.

Parmi les filtres les plus connus, on retrouve notamment :
- le **filtre moyenneur** (calcule pour chaque pixel la moyenne du pixel avec ses 8 proches voisins)
- le **filtre gaussien** permettant de réduire le bruit d'une image fournie en entrée

Taille du filtre: appliquer K filtres de taille F×F engendre un *feature map* de sortie de taille O×O×K

La convolution entre une couche d'entrée de taille 32 × 32 × 3 et un filtre de taille 5 × 5 × 3 produit une couche de sortie de dimensions spatiales 28 × 28



Convolution : Paramètres des filtres

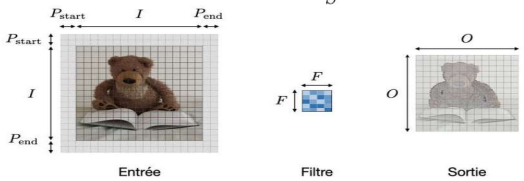
Pas (strides) : est un paramètre qui dénote le nombre de pixels par lesquels la fenêtre se déplace après chaque opération.

Zero-padding : est une technique consistant à ajouter P zeros à chaque côté des frontières de l'entrée.
Cette valeur peut être spécifiée soit manuellement, soit automatiquement

0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

En notant :
I le côté du volume d'entrée, F la taille du filtre, P (Pstart+Pend) la quantité de zero-padding, S la stride, la taille O de la feature map de sortie suivant cette dimension est telle que :

$$O = \frac{I - F + P_{start} + P_{end}}{S} + 1$$



Convolution : librairie Keras

Pour chaque couche de convolution on s'intéresse à :
- **le nombre de neurones,** chaque neurone d'une couche appliquant une convolution différente sur l'ensemble de l'image
- **la taille** du motif, le nombre de pixel qu'il contiendra
- **le pas**, la distance entre les motifs (elle peut être nulle)

```
tf.keras.layers.Conv2D(
    filters, kernel_size,
    strides=(1, 1),
    padding='valid',
    activation=None
)
```

Exemple:

```
from keras import layers model.add(
        layers.Conv2D(64, (3, 3), strides=(1, 1),activation='relu'))
```

Ici on a une **couche de convolution** avec plusieurs paramètres :
- 64 filtres
- un motif de taille (3, 3), aussi appelé kernel – 3×3 pixels
- un pas de (1,1), aussi appelé strides (par défaut il est égal à (1,1)),
- une fonction d'activation relu

Cette couche passe sur **chaque pixel de l'image** (strides = (1,1)) pour en **extraire des motifs** de taille 3×3 pixels (kernel = (3,3)). La couche exécute cette action **64 fois** (filtres = 64).
Cette couche produit ce qu'on appelle des feature-map **64 feature-maps possédant chacune des caractéristiques différentes de l'image**.

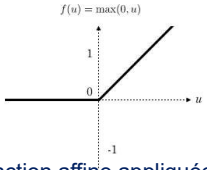
Couche Non-linéarité de correction (ReLu)

La couche de correction ou d'activation est l'application d'une fonction non-linéaire aux cartes de caractéristiques en sortie de la couche de convolution.

En rendant les données non-linéaires, elle facilite l'extraction des caractéristiques complexes qui ne peuvent pas être modélisées par une combinaison linéaire d'un algorithme de régression.

Les fonctions non-linéaires les plus utilisées sont :

- Sigmoïde ou logistique $f(x) = \frac{1}{1+e^{-x}}$
- tangente hyperbolique $f(x) = \frac{2}{1+e^{-2x}} - 1$
- Unité de rectification linéaire (ReLU) $f(x) = \begin{cases} 0 \text{ si } x < 0 \\ x \text{ si } x \geq 0 \end{cases}$



La fonction **ReLU** est choisie, car elle maximise la décision de la fonction affine appliquée par convolution. Elle remplace chaque valeur négative par un 0

$f(x) = max(0,x)$.

Couche Pooling

est utilisée afin de réduire la taille d'une couche tout en s'assurant que les éléments importants d'une image sont gardés.

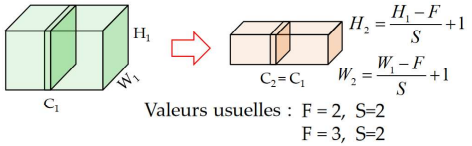
- Elle permet de réduire l'utilisation de la mémoire, le nombre de paramètres à apprendre et la quantité de calculs effectués dans le réseau.
- Elle insérée régulièrement entre les couches de correction et de convolution.

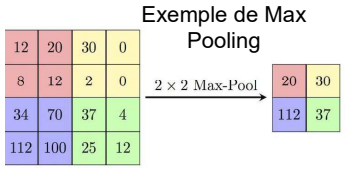
La couche prend en entrée les **feature-maps** créée par le *Conv2D* et applique le MaxPooling pour **réduire la dimension** de chacune de ces feature-map

En pratique, il existe deux méthodes courantes de Pooling :
- Max Pooling : la valeur la plus haute choisit parmi celles présentes dans la fenêtre
- Average Pooling: Calcule la valeur moyenne de chaque fenêtre

Couche Pooling

- Consiste à imaginer une fenêtre de 2 ou 3 pixels qui glisse au-dessus d'une image, comme pour la convolution.
- Avec pas de 2 pour une fenêtre de taille 2, et des pas de 3 pour 3 pixels.
- La taille de la fenêtre est appelée « kernel size » et les pas s'appellent « strides »






```
from keras import layers model.add(
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2), strides=None))
```

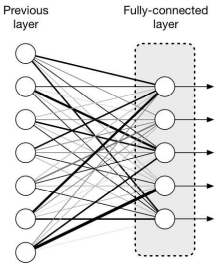
La couche **MaxPooling** a deux paramètres :
- un **kernel** de taille (2, 2)
- un **stride** égal à None (None est la valeur par défaut)

Couche Fully Connected (FC)

Après avoir extrait les caractéristiques des entrées, on attache à la fin du réseau un perceptron ou bien un MLP (Multi Layer perceptron) communément appelé **Fully connected.**

Ces couches sont placées en fin d'architecture de CNN et sont **entièrement connectées à tous les neurones de sorties.**

Après avoir reçu un vecteur en entrée, la couche FC applique successivement une **combinaison linéaire** puis une **fonction d'activation** dans le but final de classifier l'input image



Fonction de perte (LOSS)

Elle calcule l'erreur entre la prévision du réseau et la valeur réelle.
Lors d'une tâche de classification, la variable aléatoire est discrète, car elle peut prendre uniquement les valeurs 0 ou 1, représentant l'appartenance (1) ou non (0) à une classe.

La fonction de perte la plus courante et la plus adaptée est la fonction **d'entropie croisée (en anglais cross-entropy).**

issue du domaine de la théorie de l'information, et mesure la différence globale entre deux distributions de probabilité (celle de la prévision du modèle, celle du réel) pour une variable aléatoire ou un ensemble d'évènements

$$loss(x,\ class) = -\sum_{class\ =\ 1}^{C} y_{x,\ class} \log(p_{x,\ class})$$

y est la probabilité estimée d'appartenance de *x* à la classe *i*, p la probabilité réelle d'appartenance de *x* à la classe *i*, sachant qu'il y a *C* classes.

Apprentissage d'un réseau de neurones à convolution

La phase d'apprentissage, de test et validation du réseau de neurones à convolution reste très similaire à celui du réseau de neurones multicouches, et les filtres sont adaptés à chaque itération également .

Technique *DropOut*

- L'idée de cette technique est de désactiver des neurones de manière aléatoire dans le réseau afin que celui-ci soit redondant et qu'il puisse trouver de nouveaux moyens de résoudre un même problème.
- Cette technique n'est utilisée que pendant la phase d'apprentissage.
- Il faut que la probabilité qu'un neurone ne soit pas désactivé soit entre 0.5 et 0.8 pour obtenir les meilleurs résultats.

CNN : Exemple reconnaître les chiffres écrits à la main 4 7 …

```
def small_model():
    # create model
    model = Sequential()
    model.add(Conv2D(64, (3, 3), input_shape=(1, 28, 28), activation='relu'))
    model.add(Conv2D(32, (3, 3), activation='relu'))
    model.add(Flatten())
    model.add(Dense(nbre_classes, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

- 1ere convolution de 64 filtres en 3×3 suivie d'une couche d'activation ReLU
- 2eme convolution de 32 filtres en 3×3 suivie d'une couche d'activation ReLU
- **Un flatten qui va créer le vecteur final à envoyer au réseau de neurones artificiels (alias dense) (**permet d'**aplatir le tenseur**, de réduire sa dimension. Elle prend **en entrée un 3D-tensor et retourne un 1D-tensor)**.
- Un dense, réseau de neurones artificiels qui aura 10 neurones et sera suivi d'un softmax

Paramètres de Conv2D : le nombre de filtres, les dimensions du noyau (3×3), **input_shape** : la taille des données d'entrée (uniquement sur la 1ère couche du CNN) et enfin la fonction d'activation

CNN : Exemple reconnaître les chiffres écrits à la main 4 7 …

```
def medium_model():
    # create model
    model = Sequential()
    model.add(Conv2D(32, (5, 5), input_shape=(1, 28, 28), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(nbre_classes, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

- Le max-pooling est ajouté grâce à MaxPooling2D, qui prend en paramètre les dimensions du pooling (2,2) dans le paramètre pool_size.
- Le dropout pour sa part est ajouté via Dropout, qui reçoit directement le taux de dropout (i.e. 0.2). Cela signifie que 20% des neurones seront ignorés

Avantages et inconvénients de CNN

- Contrairement aux méthodes traditionnelles, les *features* ne sont pas pré-définies selon un formalisme particulier (par exemple SIFT), mais apprises par le réseau lors la phase d'apprentissage

Les noyaux des filtres désignent les poids de la couche de convolution. Ils sont initialisés puis mis à jour par **rétro propagation du gradient**.

Les CNN sont capables de déterminer tout seul les éléments discriminants d'une image, en s'adaptant au problème posé.

Par exemple, si la question est de distinguer les chats des chiens, les f*eatures* automatiquement définies peuvent décrire la forme des oreilles ou des pattes.

- ***La couche de pooling*** *permet de réduire le nombre de paramètres et de calculs dans le réseau. On améliore ainsi l'efficacité du réseau et on évite le sur-apprentissage.*