

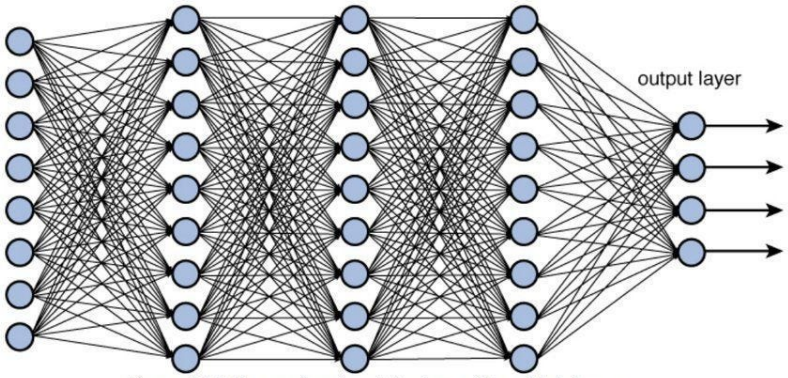
Hands-on Introduction to Deep Learning

Artificial Neural Networks

cnrs IDRIS

Goals of this section

- understanding the origin and development of neural networks
- mastering the fundamentals of neural networks



Artificial Intelligence

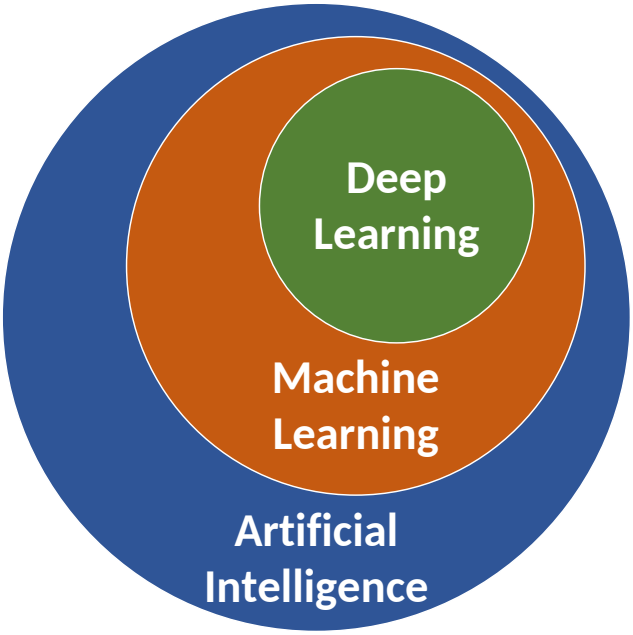
Systems capable of reproducing human actions or decisions

Machine Learning

Algorithms mainly based on statistical methods, often with an iterative aspect from which comes the term « Learning »

Deep Learning

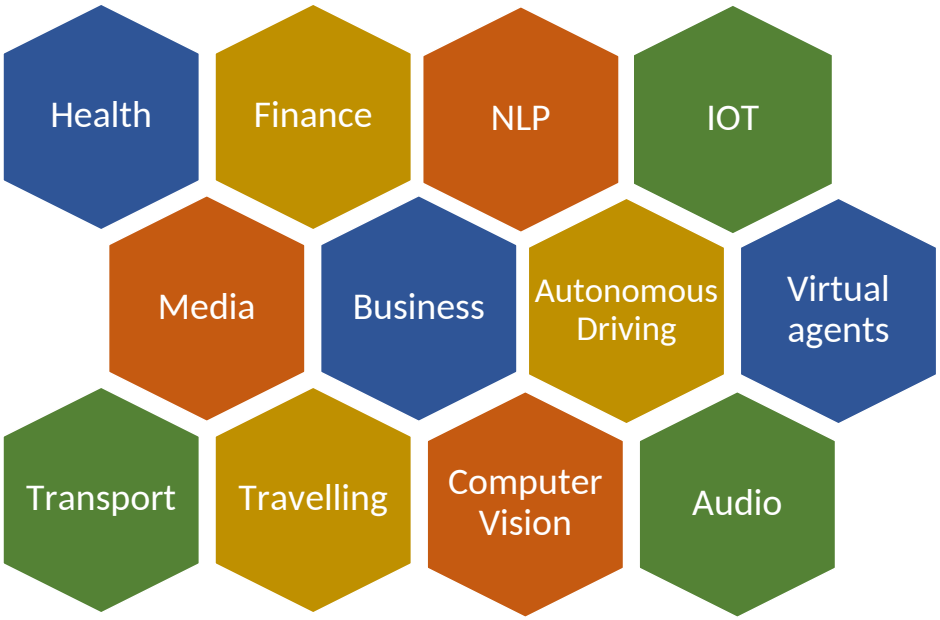
A group of models based on logical units called neurons and distributed in layers



Ici : DL *–fait partie de->* ML *–fait partie de->* IA

Aujourd'hui les gens le voient parfois comme : IA *–fait partie de->* DL *–fait partie de->* ML

Application fields



Le domaine de l'IA est porté par de nombreux succès, aussi bien dans l'industrie que dans la recherche.
On le retrouve donc dans une très grande variété de secteurs d'activité, qui ne se ressemblent pas forcément :

- les données sont de types très différents,
- les jeux de données n'ont pas les mêmes propriétés,
- et les tâches à accomplir varient d'un cas à l'autre.

Application fields

IOT



- Personal assistant
- Smart connected object

Healthcare



- Medical Imaging
- Drug research

Finance



- Fraud detection
- Prediction of financial flows

Autonomous driving



- Route optimization
- Autonomous driving

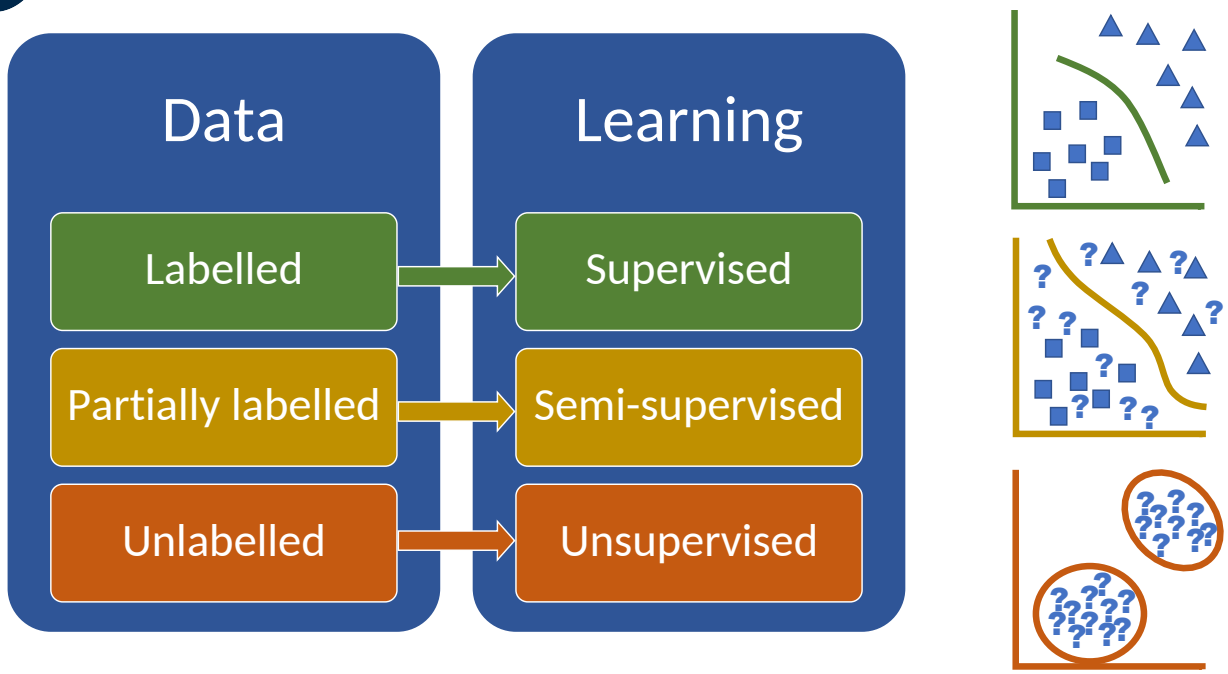
Différents domaines =/= différentes modalités

Il existe des architectures/concepts réutilisables entre différents domaines grâce aux :

- Similarités dans les données et les problèmes
- Similarités de tâches dans différents domaines

—> par exemple pour plusieurs secteurs : traitement d'images

Type of problems



Les mêmes outils reviennent d'un secteur à l'autre parce que les problèmes se ressemblent. Pour s'y retrouver, on les classe selon une première question :

est-ce que mes données sont étiquetées ou non ? Prenons un exemple simple, avec des fruits :

EXEMPLE **PANIER DE FRUITS** :

- tous les fruits du panier sont déjà étiquetés "pomme" / "banane" —> Données étiquetées —> apprentissage supervisé
- quelques fruits sont étiquetés, la majorité ne l'est pas —> apprentissage semi-supervisé
- Aucun fruit n'est étiqueté —> apprentissage non supervisé

Supervisé :

- Le modèle apprend une règle pour prédire l'étiquette de nouveaux cas qui eux sont non étiquetés
- Difficulté : construire le jeu de données étiquetées (prend beaucoup de temps)
- EXEMPLE COURBES : tous les points sont déjà classés triangle / carré → on trace une frontière nette entre les deux

Non supervisé :

- Apprentissage « autonome » visant non pas à prédire des informations mais à les extraire en maximisant certains critères
- Difficulté : Évaluer le modèle et déterminer quelle règle utiliser pour l'optimisation
- EXEMPLE COURBES : aucun point n'a de forme connue → pas de frontière possible, seulement des regroupements de points proches (cercles)

Semi-supervisé :

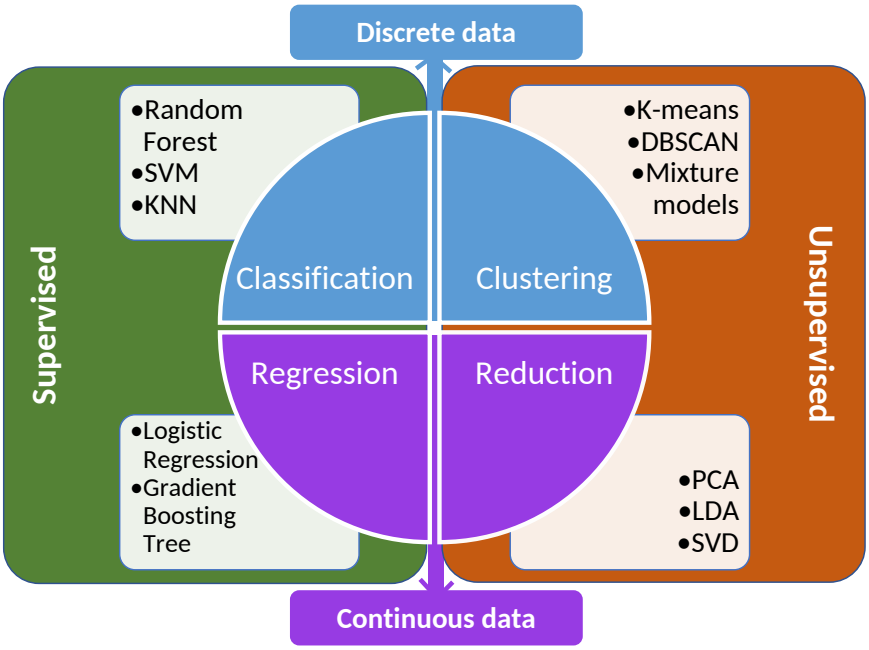
- Objectif : Réduire la quantité de données à étiqueter, tout en gardant de bonnes performances
- EXEMPLE COURBES : seuls quelques points ont une forme connue → la frontière est retrouvée avec moins d'information de départ

Discrete Data

data that can only take specific, separate values, usually whole numbers or categories you can count

Continuous Data

data that can take any value within a range, including fractions and decimals, usually something you measure



Pour choisir un algorithme, il faut aussi se demander quel type de résultat on veut :

une catégorie ou un nombre ? C'est ce qu'on va voir en croisant ces deux critères.

- **Plusieurs Types de données**
- **Discret** : des valeurs séparées, qu'on peut compter (ex : "chat" ou "chien", le nombre d'enfants dans une classe).
- **Continu** : des valeurs qui peuvent prendre n'importe quel nombre dans un intervalle, avec des décimales (ex : une taille, une température).

Tâches d'apprentissage

Apprentissage supervisé (données étiquetées) + on veut une **sortie discrète** —> **classification** | ex : chien ou chat

Apprentissage supervisé (données étiquetées) + on veut une **sortie continue** —> **regression** | ex : âge du chien à partir de ses données de santé (âges connus à l'entraînement)

Apprentissage non supervisé (données non étiquetées) + on veut une **sortie discrète** —> **clustering** | ex : regrouper des chiens selon plusieurs caractéristiques, sans catégories prédéfinies

Apprentissage non supervisé (données non étiquetées) + on veut une **sortie continue** —> **réduction** | ex : photo de chien (100x100px) réduite en 50px (le modèle trouve seul les quelques valeurs qui décrivent le mieux les données, sans qu'on lui dise quoi garder.)

Classification *(supervisé + discret)*

- Le modèle doit prédire **une catégorie parmi un ensemble fini et connu à l'avance** (ex : "chat" ou "chien", "spam" ou "non spam")
- Ex : random forest = un ensemble de nombreux arbres de décision, chacun entraîné sur un sous-échantillon légèrement différent des données ; la prédiction finale est obtenue en combinant leurs réponses (vote majoritaire en classification, moyenne en régression), ce qui rend le modèle plus robuste qu'un seul arbre isolé.

Régression *(supervisé + continu)*

- Le modèle doit prédire **une valeur numérique continue** (ex : un prix, une température, un âge).
- Même logique que la classification (on a des exemples étiquetés à l'entraînement), sauf que la sortie n'est plus une catégorie mais un nombre.
- Exemples d'algorithmes : régression logistique, Gradient Boosting Tree.

Bonus : to delve deeper

Random Forest: a collection of many decision trees, each trained slightly differently; the final prediction is decided by majority vote among all trees.

SVM (Support Vector Machine): finds the best boundary (line or curve) that separates classes by maximizing the margin, i.e. the distance between the boundary and the closest data points.

KNN (K-Nearest Neighbors): classifies a new point by looking at its K closest neighbors in the training data and assigning the most common class among them.

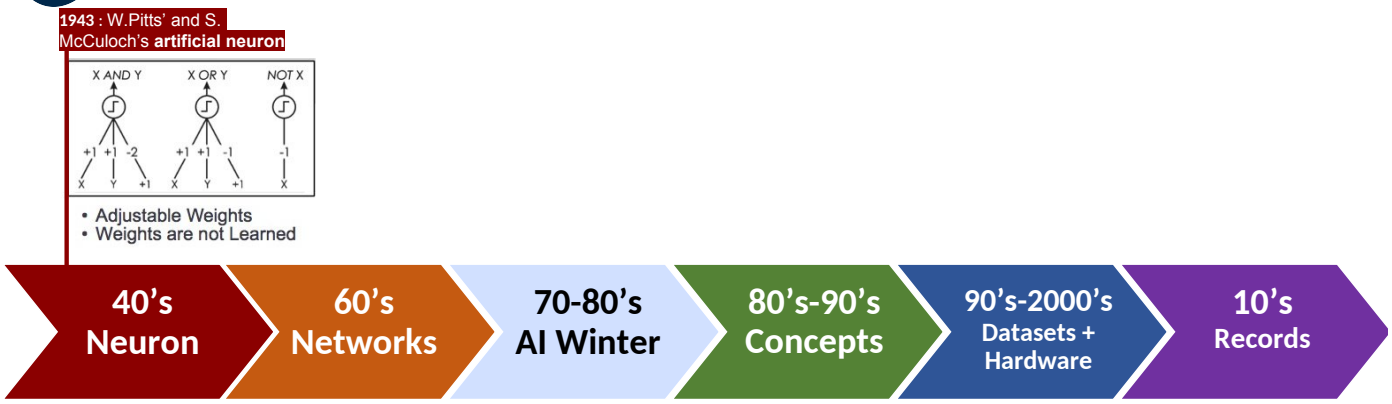
K-means: groups data into K clusters by repeatedly assigning points to the nearest cluster center, then updating each center to be the average of its assigned points.
DBSCAN: groups points that are densely packed together, and marks points in low-density regions as outliers/noise . Unlike K-means, it doesn't require choosing the number of clusters in advance.
Mixture models: assume the data comes from a mix of several underlying distributions (often Gaussian); the model estimates these distributions and assigns each point a probability of belonging to each one.

Logistic Regression: despite its name, mainly used for classification; it estimates the probability that an input belongs to a given class using a curve (the sigmoid function).

Gradient Boosting Tree: builds decision trees one after another, where each new tree focuses on correcting the errors made by the previous ones.

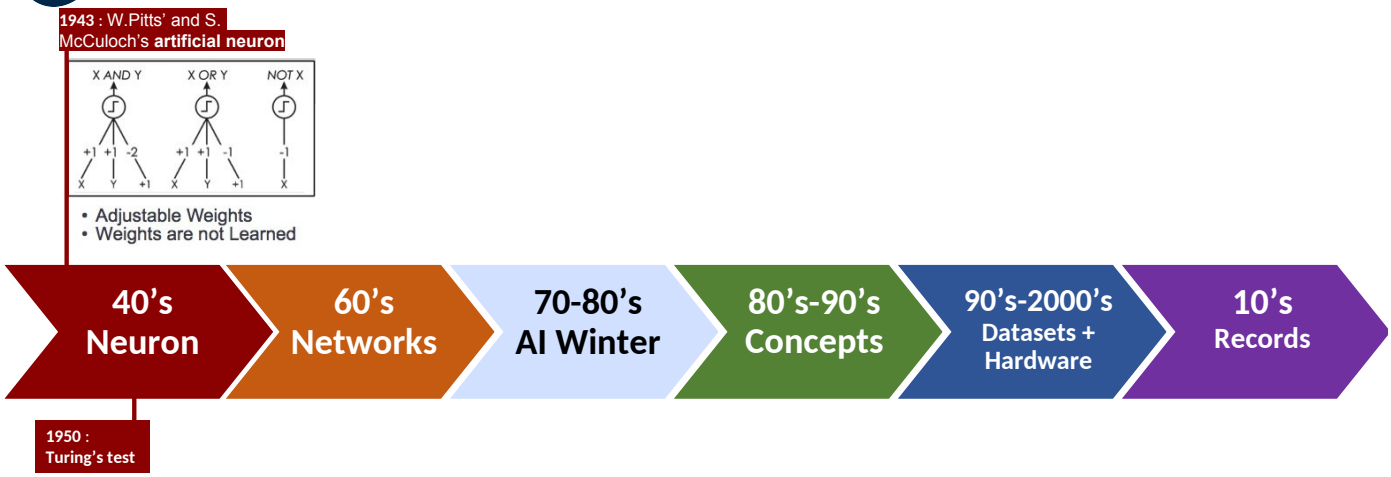
PCA (Principal Component Analysis): finds new axes (combinations of the original variables) that capture the most variation in the data, allowing it to be represented with fewer dimensions.
LDA (Linear Discriminant Analysis): similar to PCA, but finds axes that best separate known classes, rather than just capturing variation.
SVD (Singular Value Decomposition): a mathematical technique that breaks a data matrix into simpler components, often used as the underlying method behind PCA and other reduction techniques.


40's
Neuron
60's
Networks
70-80's
AI Winter
80's-90's
Concepts
90's-2000's
Datasets +
Hardware
10's
Records



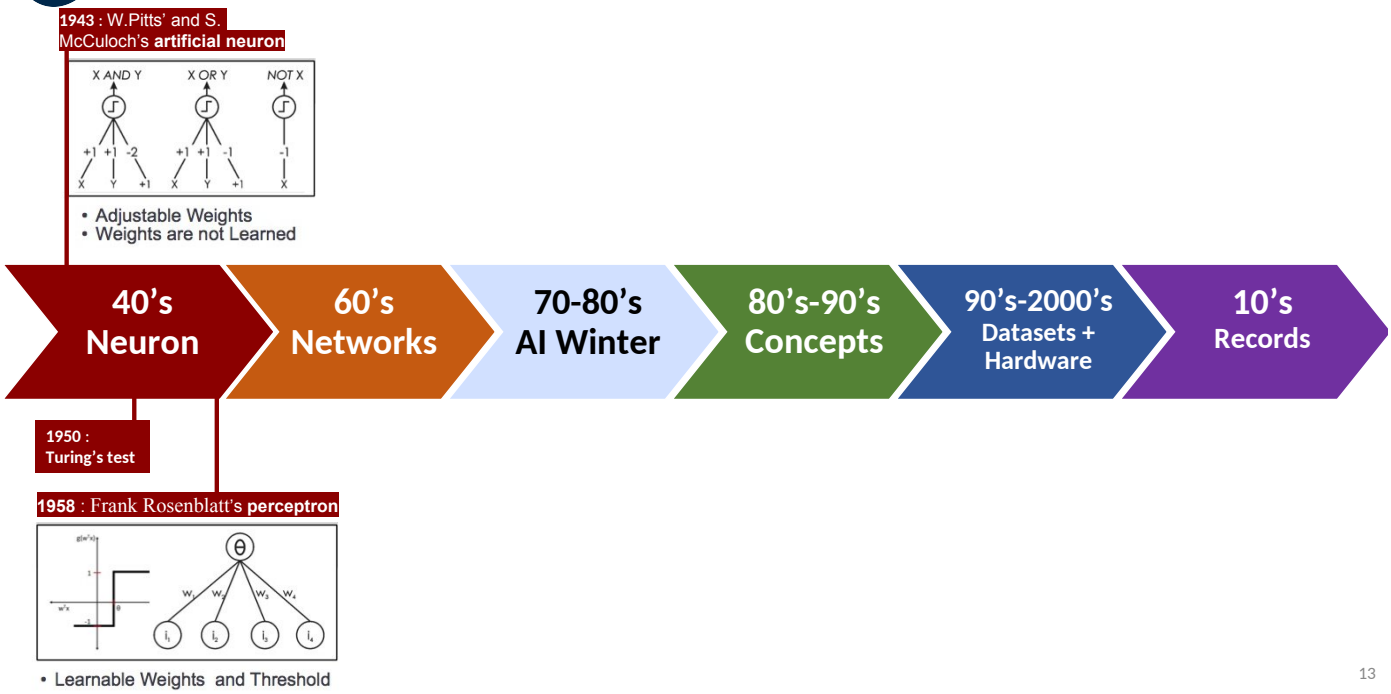
Les débuts : 1940

- **1943 Neurone artificiel : W.Pitts et S. McCuloch**
- —> Premier modèle mathématique simplifié d'un neurone biologique :
- il prend plusieurs entrées, leur applique un poids, additionne le tout, puis produit une sortie binaire (0 ou 1) si la somme dépasse un certain seuil.
- (schéma : les poids peuvent être mis à jour manuellement uniquement)



Les débuts : 1940

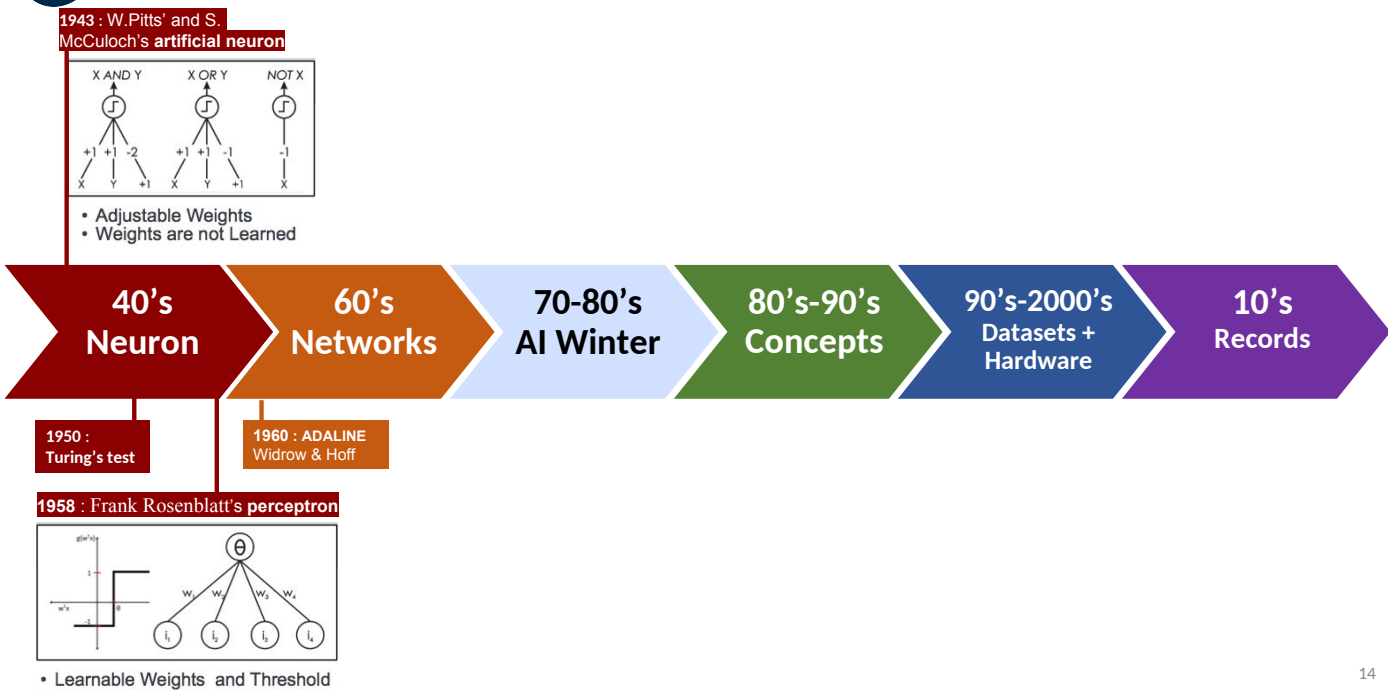
- **1950 Apprentissage machine : Le test de Turing**
- expérience de pensée suivante :
- création d'un test test consiste à mettre un humain en confrontation verbale à l'aveugle avec un ordinateur et un autre humain.
- Si la personne qui engage les conversations n'est pas capable de dire lequel de ses interlocuteurs est un ordinateur,
  - —> on peut considérer que le logiciel de l'ordinateur a passé avec succès le test. (définit l'IA)



Les débuts : 1940

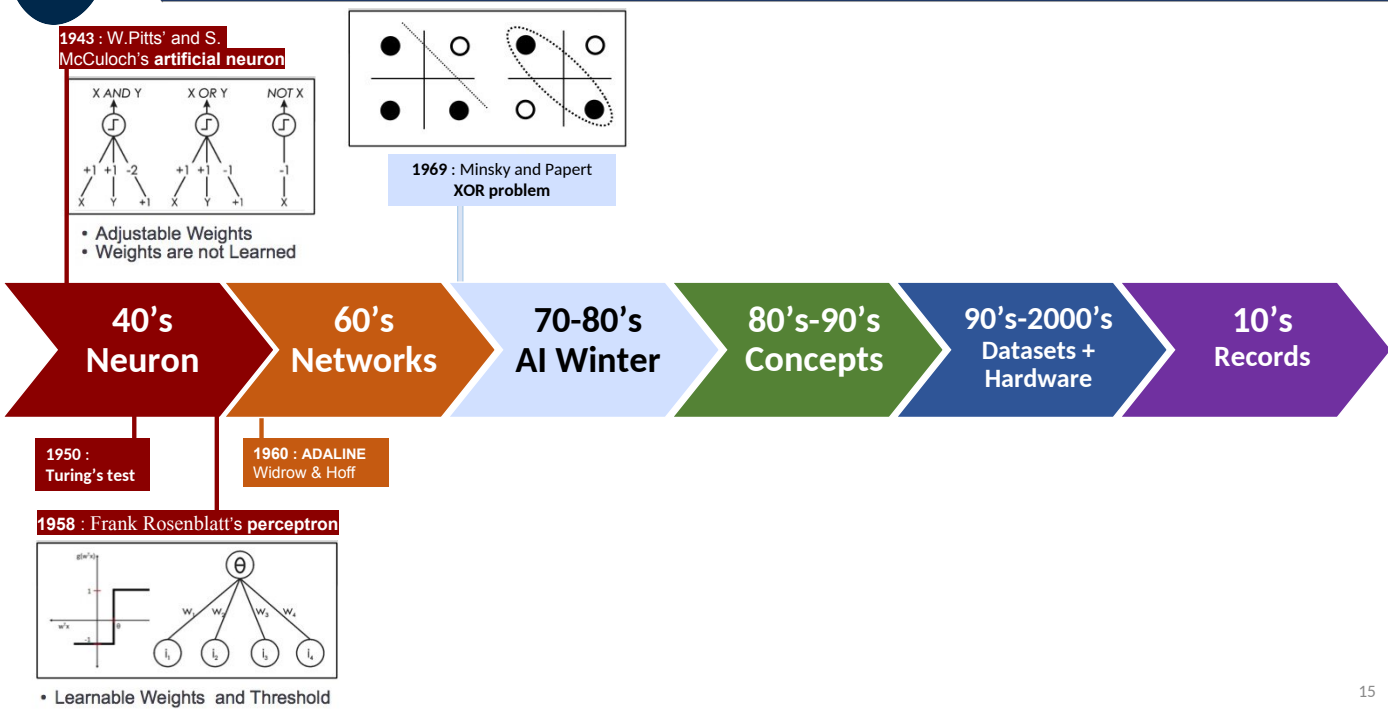
1958 Le Perceptron : Frank Rosenblatt

- avancée majeure : contrairement au neurone de McCulloch & Pitts, ici les poids **sont appris automatiquement**, à partir d'exemples, via un processus itératif (on ajuste petit à petit les poids en fonction des erreurs commises)
- Pourquoi c'est une avancée majeure : C'est la première fois qu'une machine peut **apprendre toute seule** à partir d'exemples, sans qu'un humain ait à calculer ou fixer les poids à la main



- **1960 ADALINE (ADAptive LINear Element)** Widrow & Hoff :
- **Réseau de neurones à couche unique sommée**
- —> calcule l'erreur sur la somme pondérée **avant** d'appliquer le seuil (qui lui n'intervient qu'à la fin pour donner la décision finale)
- ce qui donne une erreur plus précise, continue et nuancée (et pas juste "juste/faux") pour ajuster les poids.
  - —> ouvre la voie aux méthodes de descente de gradient utilisées aujourd'hui dans tous les réseaux de neurones.

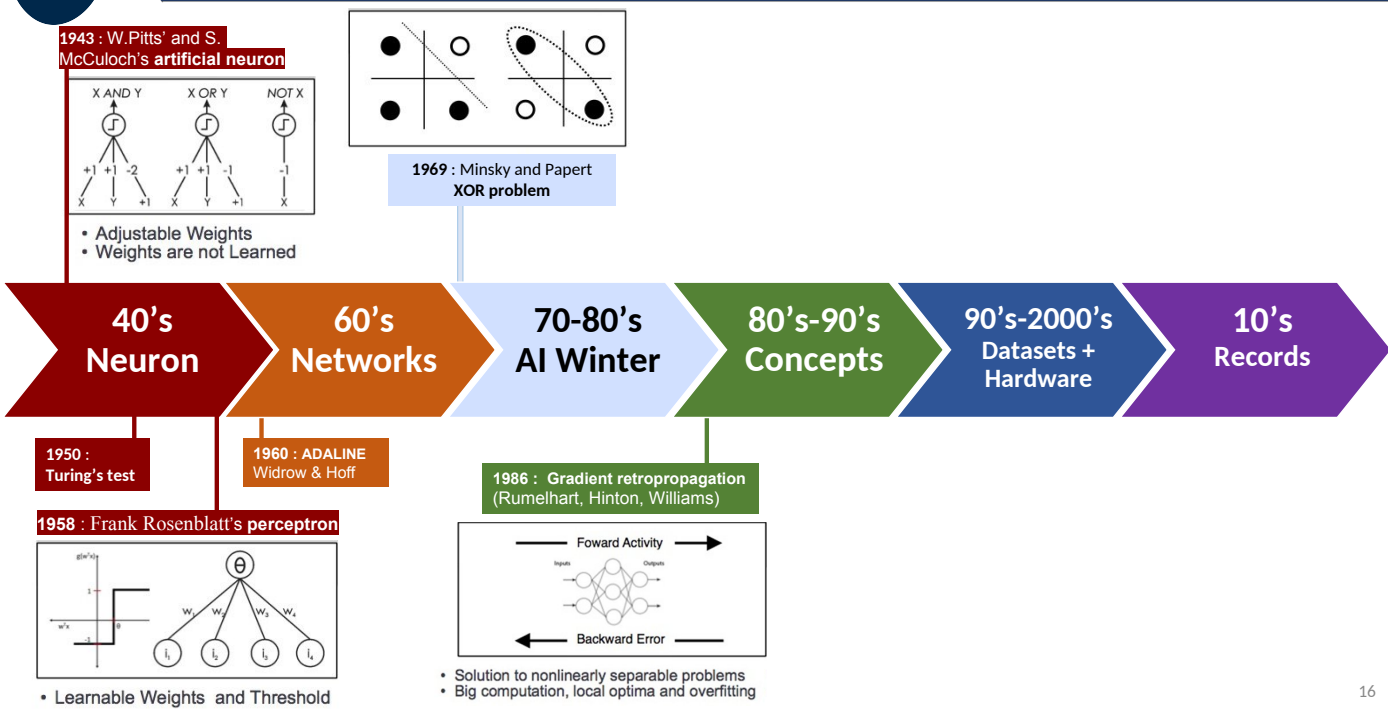
History of Deep Learning



L'hiver de l'IA : 1974 – 1980

- **1969 — Le problème XOR**
- Minsky et Papert ont démontré ce problème en 1969
- Perceptron/ADALINE : ne séparent que des données **linéairement séparables**
- XOR : impossible à séparer avec une seule droite
- → ce qui a jeté un doute sérieux sur les capacités réelles des réseaux à une seule couche.
- **L'hiver de l'IA (1974-1980)**
- Limitation théorique (XOR) + combinée à des promesses excessives faites dans les années précédentes non tenues (on annonçait des machines "intelligentes" imminentes)
- → perte de confiance des financeurs
- → chute des investissements en recherche IA

History of Deep Learning

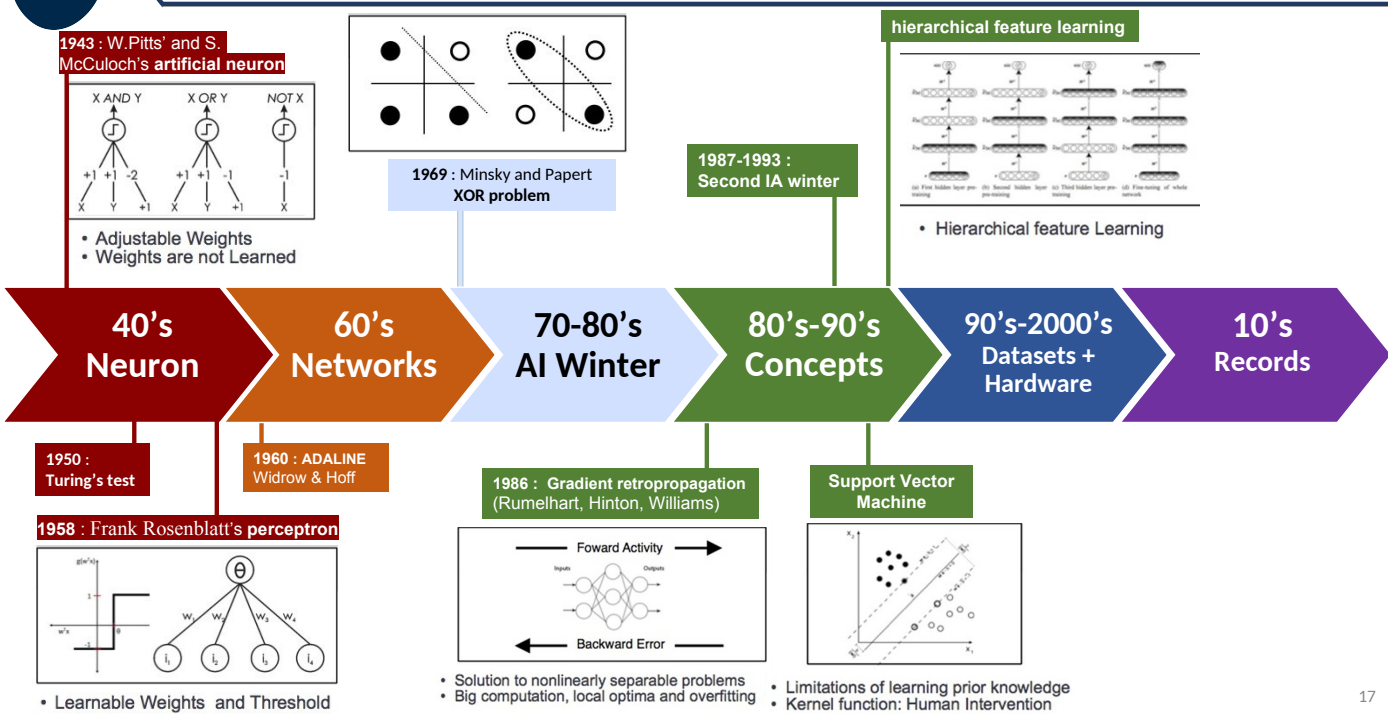


80's-90's : Transition vers les multicouches, nouvelles fonctions d'activation, optimiseurs, … chaque partie est améliorée de manière incrémentale.

Les modèles augmentent en complexité et l'entraînement devient difficile.

1986 Rétropropagation du gradient (Rumelhart, Hinton, Williams) :

- méthode permet enfin d'entraîner efficacement des réseaux à plusieurs couches en propageant l'erreur de la sortie vers l'entrée, couche par couche, pour ajuster tous les poids.
- C'est ce qui permet de résoudre le problème XOR



1987-1993 : second hiver de l'IA

Systèmes experts et machines Lisp en échec coomercial (très cher)

Réseaux multicouches coûteux et instables à entraîner <— mis de cotés

1990's : émergence des SVM

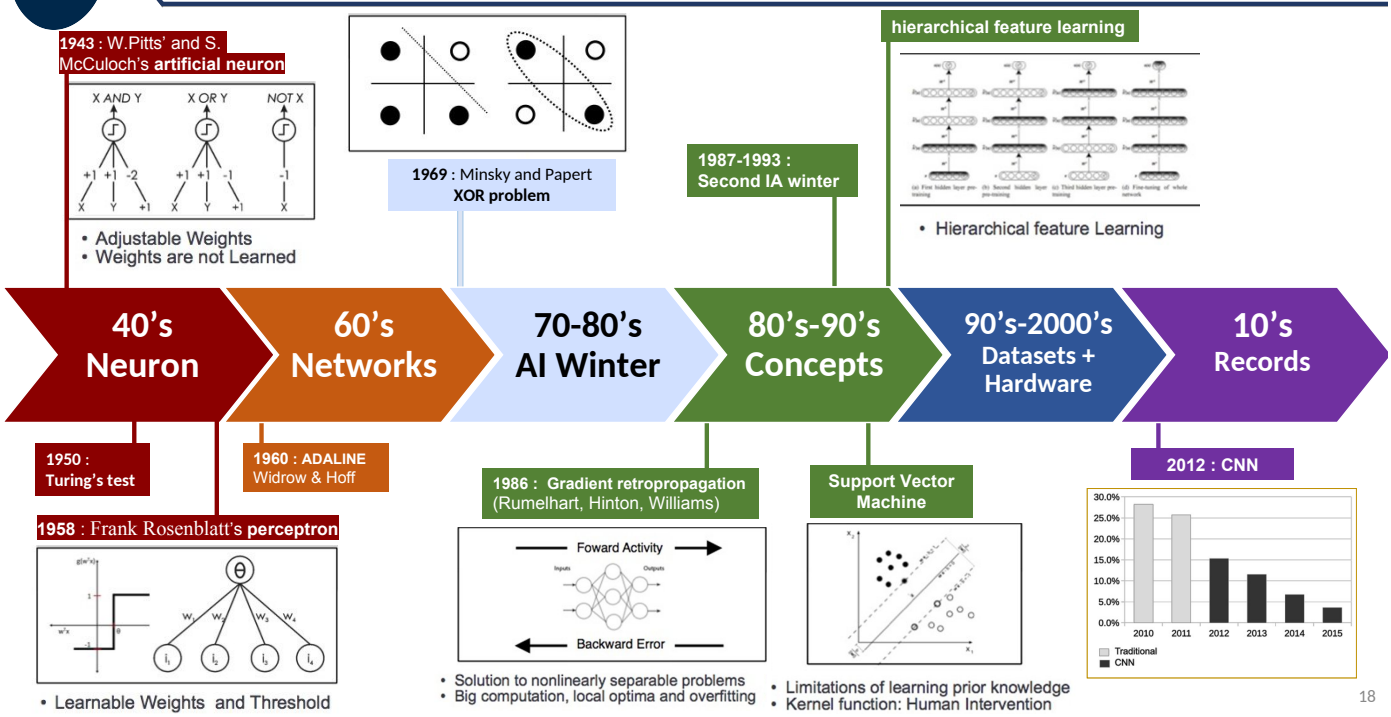
Alternative solide, les réseaux de neurones passent au second plan

- Bases mathématiques rigoureuses, moins coûteuses en calcul

Limite : choix de la fonction noyau fait **manuellement** (pas appris automatiquement)

2000's : hierarchical feature learning

- L'idée émerge que dans un réseau à plusieurs couches, chaque couche peut apprendre des caractéristiques de plus en plus abstraites à partir de la précédente. Par exemple sur une image : les premières couches détectent des contours simples, les suivantes des formes, puis des objets entiers.
- Retour des réseaux profonds
- Chaque couche apprend des caractéristiques plus abstraites



2000s-2010s — Datasets + Hardware

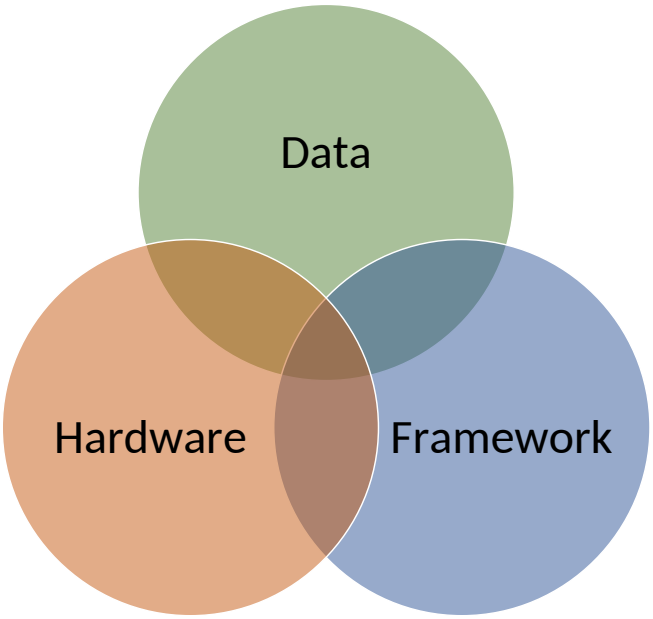
Deux obstacles pratiques des décennies précédentes sont enfin levés :

- l'apparition de **bases de données massives** (comme ImageNet en 2009 : fournit la matière première nécessaire à l'entraînement)
- **les GPU** : offrent la puissance de calcul requise pour entraîner de gros modèles en un temps raisonnable.
- —> Ces deux avancées ouvrent la voie aux percées majeures du deep learning des années 2010.

2012 — Victoire d'AlexNet :

- Victoire d'AlexNet à la compétition ImageNet (compétition annuelle de classification d'images)
- marqué le vrai début de l'ère du deep learning moderne

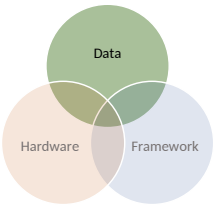
3 decisive factors

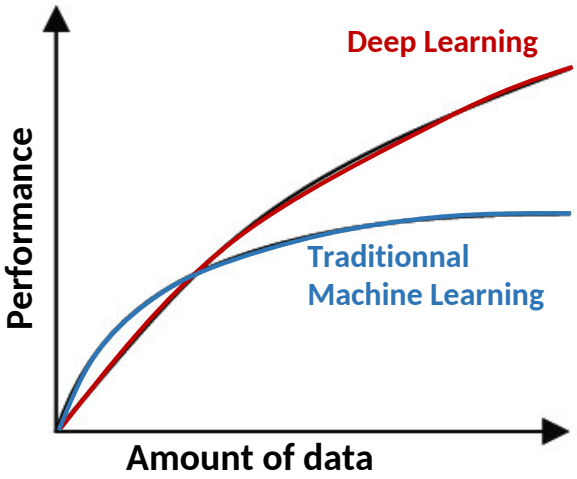


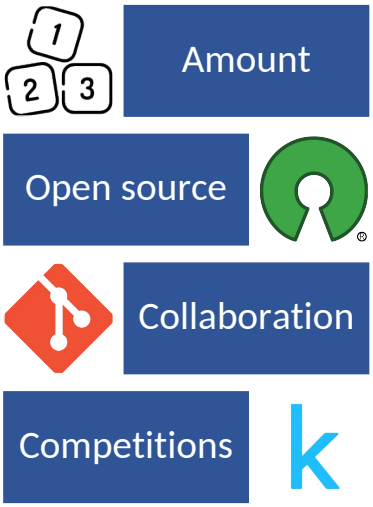
Ce qui a changé, c'est la réunion de **trois facteurs qui se combinent** :

- **Data** : explosion du volume de données disponibles + nouvelles techniques pour les étiqueter à grande échelle
- **Hardware** : les GPU apportent la puissance de calcul + après l'échec des machines Lisp
- **Framework** : architectures de plus en plus complexes grâce à la recherche, facilitées par les bibliothèques

Les trois se combinent : aucun ne suffit seul, d'où le diagramme







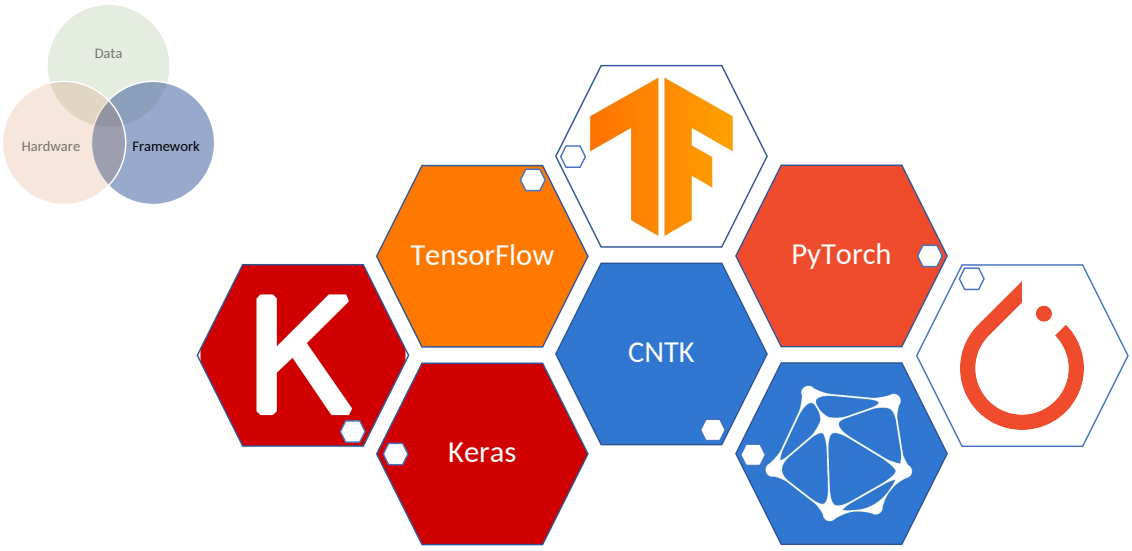
Cette abondance de données vient de plusieurs choses :

- simplement plus de volume disponible,
- mais aussi l'essor d'internet qui a donné l'essor de l'open source,
- des outils de collaboration comme Git,
- et des compétitions comme Kaggle qui stimulent la création et le partage de données.

- Les méthodes traditionnelles de machine learning progressent vite au début, mais **plafonnent rapidement** : au-delà d'une certaine quantité de données, leurs performances stagnent.
- Le deep learning, au contraire, démarre plus lentement mais **continue de s'améliorer** à mesure qu'on lui donne plus de données, sans vraiment plafonner sur ce graphique.

.

Frameworks and libraries

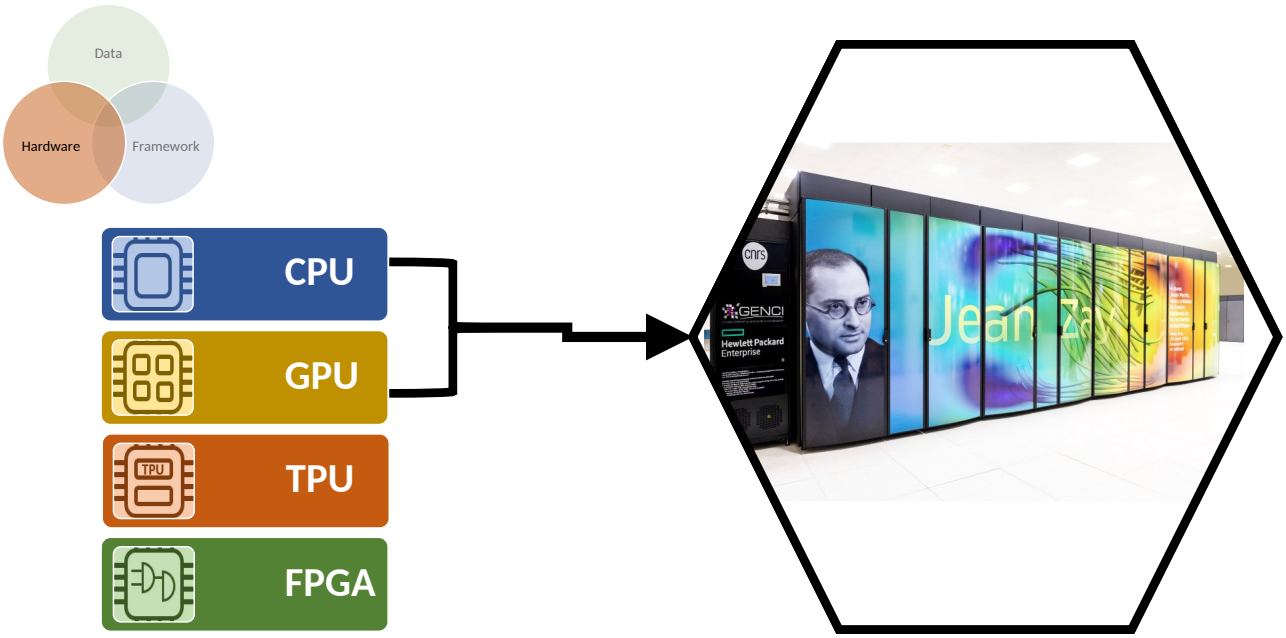


Acteurs divers :

- Facebook
- Google
- Amazon
- Microsoft
- Secteur académique (universités, chercheurs, …)
- NVIDIA
- Communautés en ligne
- Start-ups
- …

Couches logicielles à différents niveaux :

- Intégration GPU (Cuda, OpenCL)
- API optimisées : Torch, Keras
- Bibliothèques : Pytorch, Tensorflow
- Wrappers : Pytorch Lightning
- Outils de visualisation et de profilage



CPU (central processing unit) :

- polyvalent, peu de cœurs puissants
- Démarre le programme
- organise les calculs
- Gère la mémoire
- Envoie du travail aux GPU

GPU (Graphics processing Unit) :

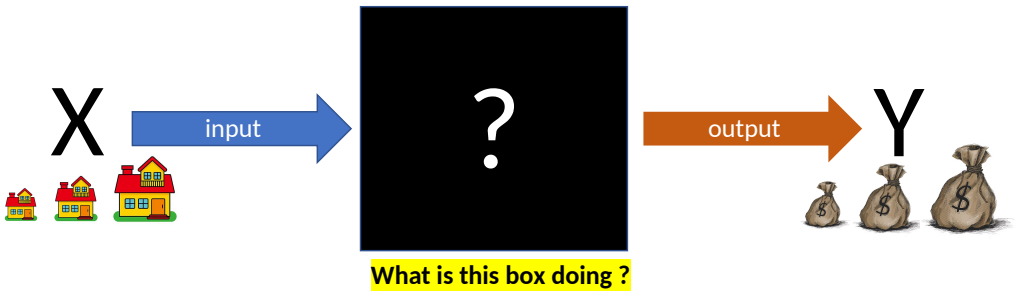
- des milliers de cœurs simples en parallèle.
- Conçu pour l'image, idéal pour les calculs matriciels des réseaux de neurones.
- Idéal pour traiter de gros volumes de calculs identiques en //

Tensor Processing Unit (TPU) :

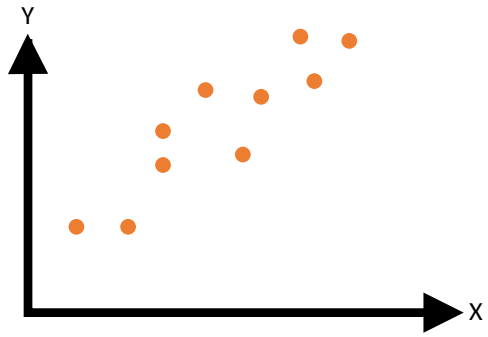
un circuit intégré spécifique (ASIC) conçu par Google pour accélérer les charges de travail liées à l'intelligence artificielle et au machine learning (apprentissage profond).

FPGA (Field-Programmable Gate Array) :

circuit logique programmable ou PLD (Programmable Logical Device) —> est un circuit intégré logique qui peut être programmé après sa fabrication.



What is this box doing ?



↳ We need to find a function that relates X to Y as much as possible. The simplest case is a linear relationship : a straight line

- données d'entrée : X

ex : la taille de plusieurs maisons

- sortie : Y

ex : leur prix

Un modèle prédictif essaie de répondre à cette question : que fait cette boîte qui prend X en entrée et donne Y en sortie ?

On voit sur le graphique que la relation entre X et Y ressemble à une tendance linéaire.

=> On va donc chercher une fonction qui relie X à Y, et le cas le plus simple, c'est une droite.

Linear regression

The « real » Y :

$$Y = X \cdot \Theta + N$$

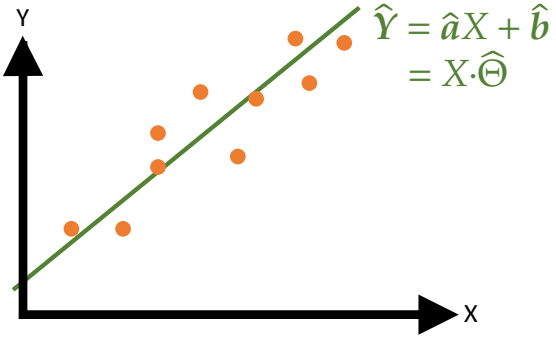
with :

$$\Theta = \begin{pmatrix} a \\ b \end{pmatrix}$$

N , noise

The estimated Y :

$$\hat{Y} = X \cdot \hat{\Theta}$$



We need to find a function that relates X to Y as much as possible

↳ It all comes down to finding $\hat{\Theta}$

- Le vrai Y suit un modèle Y = X·Θ + N
  - Θ contient les deux paramètres qu'on cherche
    - a : la pente / **le poids** (notation : poids souvent noté w)
    - b : l'ordonnée à l'origine / **le biais**
  - N représente le bruit, ce qu'on ne peut pas expliquer parfaitement.

On ne connaît jamais Θ ni N dans la réalité —> on doit les estimer —> (donne la droite verte sur le graphique)

=> Trouver le bon modèle revient donc entièrement à trouver .

Linear regression

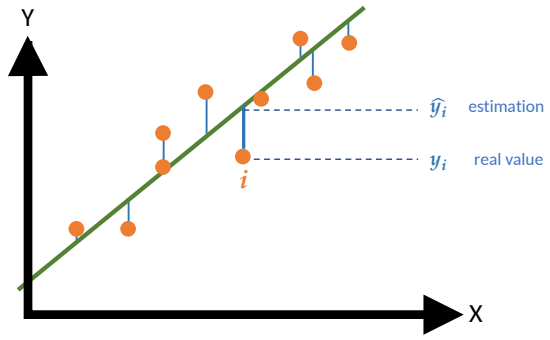
The « real » Y :

$$Y = X \cdot \Theta + N$$

with :

$$\Theta = \begin{pmatrix} a \\ b \end{pmatrix}$$

N , noise



The estimated Y :

$$\hat{Y} = X \cdot \hat{\Theta}$$

Cost function

$$\min_{\theta} J(\theta) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

- The smaller $J(\theta)$ is, the closer to the truth is the estimation
- We are looking for the parameters a and b such as the mean squared error is minimized

Comment savoir si une droite est la bonne ? —> fonction de coût

- Pour chaque point i, on compare la prédiction ■■ à la vraie valeur y■ ('écart en bleu sur le graphique)
- La fonction de coût fait la moyenne de tous ces écarts **au carré : (■■ – y■)²** , 2 raisons à ça :
  - Éviter que les erreurs positives et négatives ne s'annulent entre elles quand on les additionne
  - Pénaliser davantage les grosses erreurs que les petites (une erreur de 10 —> 100, une erreur de 2 —> 4)

Plus J(θ) est petit, plus l'estimation est proche de la réalité.

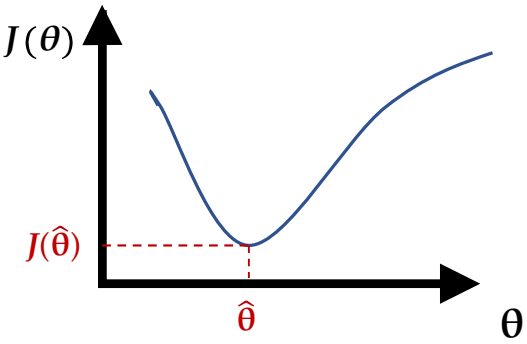
=> Le but est donc de trouver les paramètres a et b qui minimisent cette erreur moyenne.

Cost function

$$\min_{\theta} J(\theta) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$



This is a convex problem



But for a non- linear situation, it is not that simple...

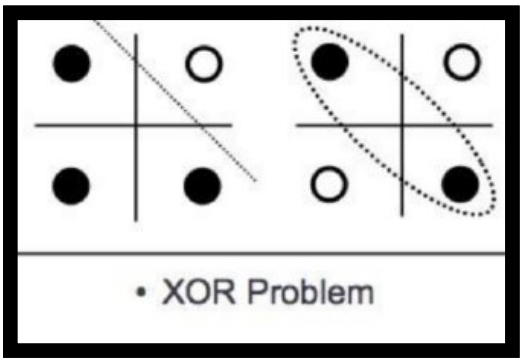
Dans un cas linéaire il n'existe qu'un min de

—> Problème convexe : il existe une solution mathématique directe pour trouver le θ■ optimal, sans avoir à « tâtonner ».

Mais attention, ça n'est plus aussi simple dès qu'on sort du cas linéaire (suite)

What about a non-linear problem?

What if we tried to combine two linear classifiers to resolve a non-linear problem ?



$$f = W_1 x$$

with W : weights matrix

$$\hat{f} = W_2 f$$
$$= W_2 (W_1 x)$$
$$= \underbrace{(W_2 \cdot W_1)}_{\widehat{W}} x$$

$$\hat{f} = \widehat{W} x$$

the combinaison of linear classifiers is a new linear classifier

This doesn't work . We need to break the linearity with an **activation function**

problème XOR —> pas linéairement séparable —> problème non-linéaire

Et si on empilait deux classifieurs linéaires l'un après l'autre, pour le rendre plus puissant ?

- La première couche : f = W■x.
- On applique ensuite une seconde couche sur ce résultat.
- Or, le produit de deux matrices comme W■·W■ donne une seule matrice :
- Peu importe le nombre de couches qu'on empile, tant qu'elles restent purement linéaires,

=> **la combinaison de plusieurs classifieurs linéaires reste un classifieur linéaire.**

Pour que ça fonctionne, il faut casser cette linéarité —> avec la **fonction d'activation**

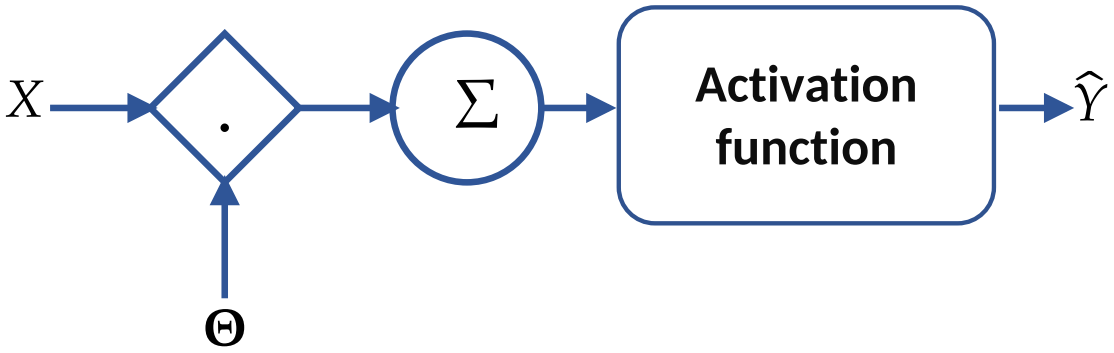
This is
a neuron

A neuron

A small computing unit that takes several inputs, combines them using weighted parameters, and passes the result through an activation function to produce an output.

Activation function

A non-linear function applied to the weighted sum of a neuron's inputs, allowing the network to model non-linear relationships



Cette fonction d'activation a un rôle essentiel :

elle applique une transformation non-linéaire à la somme pondérée des entrées du neurone, ce qui permet de casser la linéarité qu'on obtiendrait sinon.

Sans cette fonction, empiler des neurones ne servirait à rien.

Logistic regression

Loss function

Measures the error **for a single observation i** : the gap between the predicted value $\hat{y}_i$ and the true value y_i, for that specific data point.

Cost function

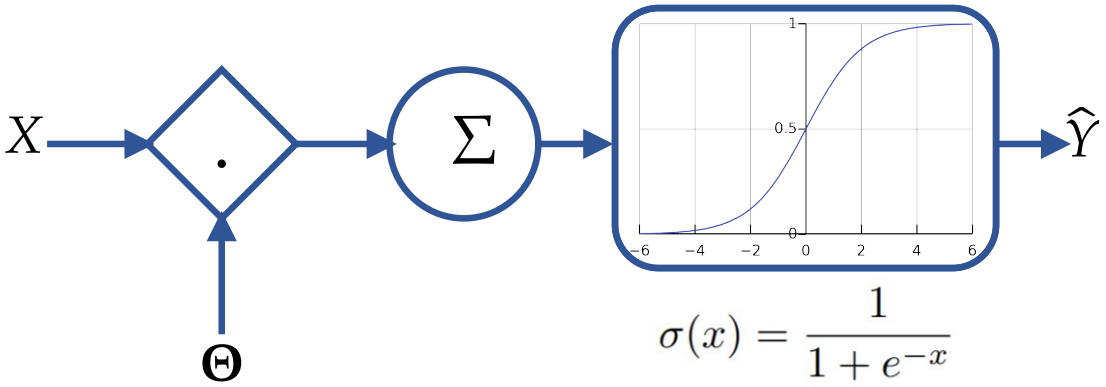
the average of the loss **over the entire dataset** (but computed from the loss rather than directly)

Binary Cross-entropy loss

$$L(\hat{y}_i, y_i) = -(y_i \log \hat{y}_i + (1-y_i) \log (1-\hat{y}_i))$$

mostly used in classification

$y_i = 0$ *and* $\hat{y}_i$ *close to* 0 ➡ $L(\hat{y}_i, y_i)$ close to 0
$y_i = 1$ *and* $\hat{y}_i$ *close to* 1 ➡ $L(\hat{y}_i, y_i)$ close to 0
$y_i = 0$ *and* $\hat{y}_i$ *close to* 1 ➡ $L(\hat{y}_i, y_i)$ tends to ∞
$y_i = 1$ *and* $\hat{y}_i$ *close to* 0 ➡ $L(\hat{y}_i, y_i)$ tends to ∞



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

On a maintenant un neurone qui produit une prédiction.

Reste à savoir si elle est bonne.

Pour cela, on la compare à la vraie valeur avec une fonction de perte.

Fonction de perte (loss) : mesure l'erreur pour **une seule observation** i , l'écart entre la prédiction ■■ et la vraie valeur y■, pour ce point précis.

Fonction de coût (cost) : c'est la **moyenne** des pertes sur tout le dataset , exactement comme J(θ) = (1/n)Σ(...) qu'on a vu avant, mais calculée à partir de la loss plutôt que directement.

Ici, on utilise l'**entropie croisée binaire** comme fonction de perte, principalement utilisée en classification.

Elle a une propriété très utile qu'on peut résumer en quatre cas (cf slide) :

- si le modèle prédit une valeur proche de la vraie catégorie (que ce soit 0 ou 1) la perte est proche de 0, tout va bien.
- Mais si le modèle est confiant et se trompe complètement, par ex il prédit une probabilité proche de 1 alors que la vraie réponse est 0 —> la perte explose vers l'infini.
- Cette formule pénalise donc très sévèrement les erreurs commises avec beaucoup de confiance.

Pour obtenir cette probabilité ■ en sortie, on utilise ici une **fonction d'activation** particulière : la sigmoïde (dont la formule est σ(x) = 1/(1+e■■)).

Elle a une forme en S caractéristique, visible sur ce graphique, qui compresse n'importe quelle valeur d'entrée pour la ramener toujours entre 0 et 1, ce qui en fait un choix naturel quand on veut générer une probabilité en sortie du modèle

- **Regression loss**
  - Average absolute deviation : $L(y,\hat{y},\theta) = \frac{1}{n}\sum_i^n |y_i - \widehat{y_i}|$
  - Least squares method : $L(y,\hat{y},\theta) = \frac{1}{n}\sum_i^n (y_i - \widehat{y_i})^2$

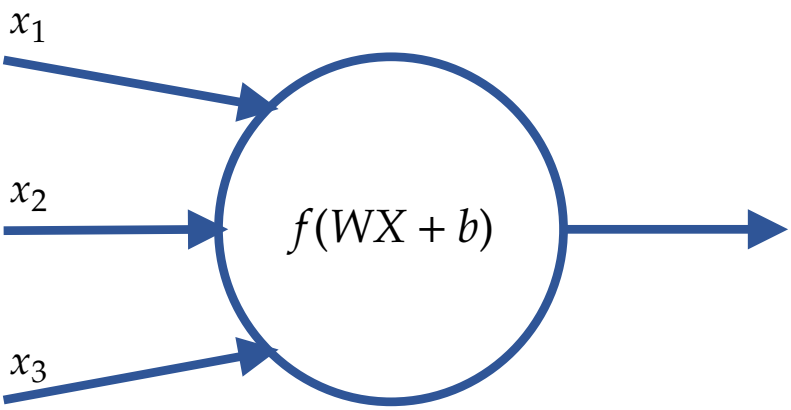
- **Classification loss**
  - Cross-Entropy : $E(y,\hat{y},\theta) = -\frac{1}{n}\sum_i^n \sum_j^m y_{ij} \log \widehat{y_{ij}}$

Il n'existe pas de fonction de perte parfaite pour toutes les situations.

Critères :

- Statistiques liées à la base de données
- Quantité de valeurs hors norme (valeurs aberrantes)
- Résultats recherchés :
  - Régression
  - Classification
  - Nombre de sorties
  - …
- Plusieurs stratégies sont possibles et combinables

1 neuron

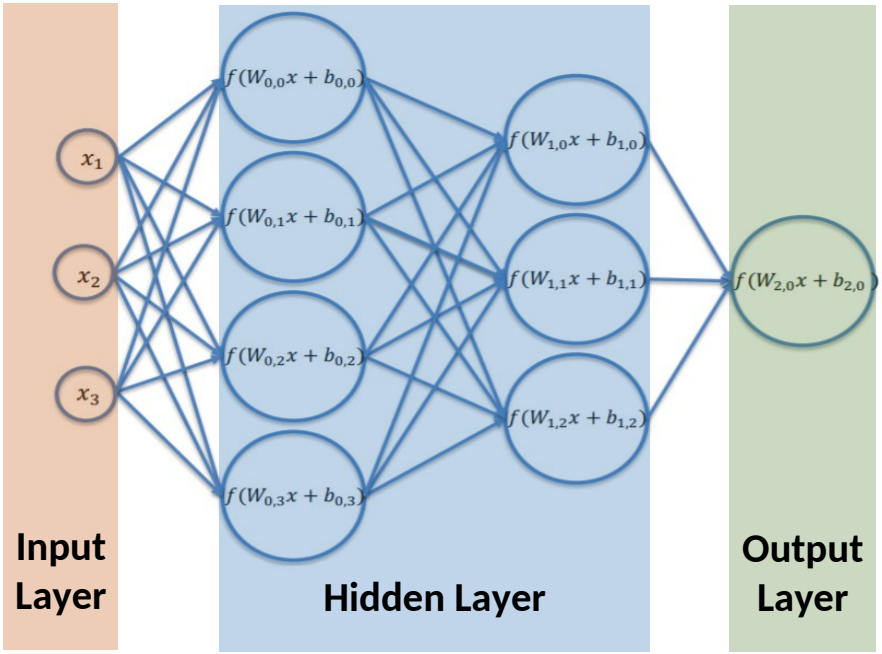


Le neurone de base :

- x : Entrée
- f : Fonction d'activation
- W : Poids
- b : Biais

Ce qui est complexe :

- L'architecture du neurone
- Le choix des hyperparamètres
- L'optimiseur
- La sélection d'une fonction de coût appropriée
- L'entraînement du modèle
- L'obtention des données nécessaires à l'entraînement



Un neurone seul est simple, et il reste limité —>La puissance vient de l'assemblage.

Le premier choix difficile de la liste, c'est l'architecture, c'est-à-dire comment on assemble les neurones.

—> on assemble les neurones en couches successives, et l'ensemble forme un **réseau de neurones**

Couche d'entrée (Input Layer)

- C'est ici qu'entrent les données.
- taille n'est pas un choix elle est directement imposée par les données elles-mêmes

Couche cachée (Hidden Layer)

- partie du réseau où le modèle apprend à combiner et transformer l'information.
- taille pas imposée : c'est un choix qu'on doit faire, en fonction de la complexité du problème.
- profondeur = le nombre de couches cachées empilées
- largeur = nombre de neurones dans chaque couche

Couche de sortie (Output Layer)

- taille pas un choix libre —> déterminée par la tâche à accomplir.
- Par exemple, un seul neurone en sortie pour une classification binaire (chat/pas chat) ou une régression (un prix), mais autant de neurones que de classes possibles pour une classification multi-classes (chat/chien/oiseau → 3 neurones).

Les questions à se poser en pratique :

Comment dimensionner le réseau ? —> c'est en grande partie empirique. Plusieurs approches aident à démarrer :

L'ordinateur / la puissance de calcul disponible

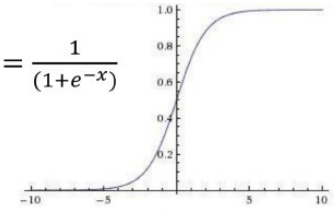
- Un réseau plus grand demande plus de mémoire et de temps de calcul
- la puissance de calcul disponible limite donc concrètement les dimensions qu'on peut se permettre.

Étude préliminaire des données :

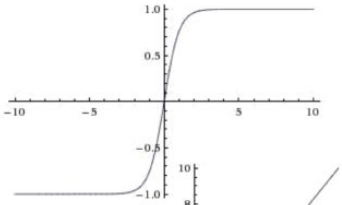
- Regarder la complexité et le volume des données avant de choisir
- peu de données et une tâche simple —> ne justifient pas un réseau immense (risque d'**overfitting**)
- données riches et complexes —> peuvent nécessiter davantage de capacité

Activation functions

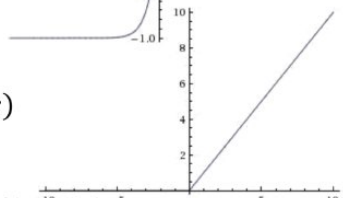
Sigmoid: $\sigma(x) = \frac{1}{(1+e^{-x})}$



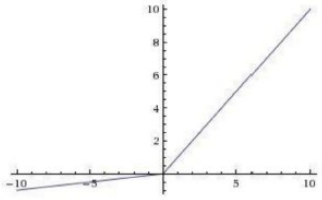
tanh: $\tanh(x)$



ReLU: $\max(0, x)$



Leaky ReLU: $\max(0.1x, x)$



Parametric ReLU: $\max(\alpha x, x)$

Maxout $\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU $f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$

Diverses fonctions d'activation :

- Linéaire : Utilisée pour une régression simple
- ReLU : Populaire, efficace, rapide. Pour des CNN très performants. Spécialise les neurones.
- Softmax : Populaire pour la classification multi-classes.

How to choose the right activation function ?



Here are the criteria to take in account :

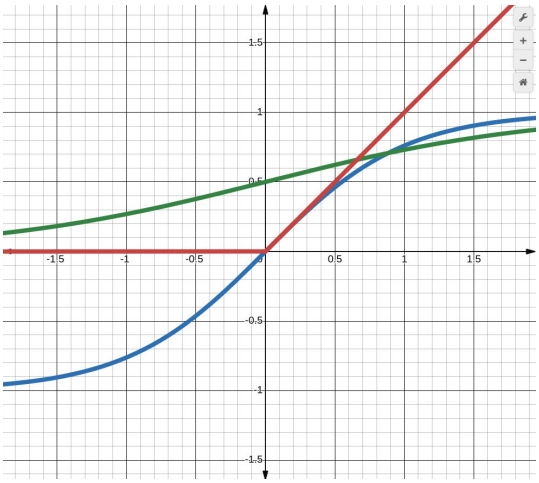
- Range
- Smoothness
- Monotonicity
- Zero-centered
- Monotonic derivative
- Computational cost

How to choose the right activation function ?



Here are the criteria to take in account :

Range
Smoothness
Monotonicity
Zero-centered
Monotonic derivative
Computational cost



— ReLU : [0,+∞[
— tanh(x) :]-1,1[
— sigmoïd :]0,1[

Range (plage de sortie)

C'est l'intervalle de valeurs que peut prendre la sortie du neurone.

Par exemple,

- ReLU donne toujours une valeur entre 0 et l'infini
- tanh entre -1 et 1,
- la sigmoïde entre 0 et 1

Pourquoi c'est important :

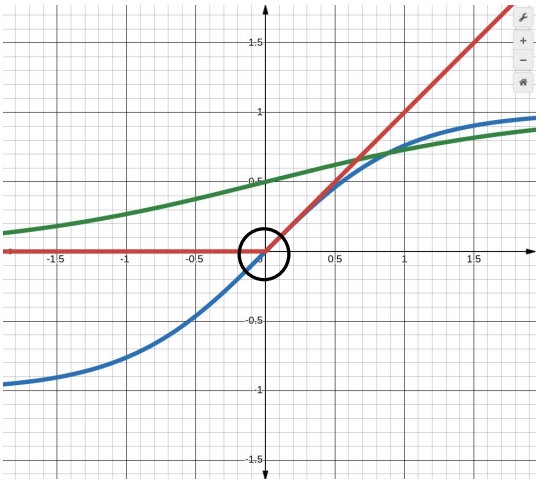
- si vous voulez une probabilité en sortie, il vous faut une fonction bornée entre 0 et 1 —> la sigmoïde s'impose naturellement.
- Si vous êtes au milieu du réseau, cette contrainte compte moins.

How to choose the right activation function ?



Here are the criteria to take in account :

- Range
- **Smoothness**
- Monotonicity
- Zero-centered
- Monotonic derivative
- Computational cost



— **ReLU : angular**
— **tanh(x) : smooth**
— **sigmoïd : smooth**

Smoothness (régularité)

- Est-ce que la fonction est lisse, sans cassure brutale ?
- La sigmoïde et tanh sont parfaitement lisses ; ReLU a un point anguleux en 0.

Pourquoi c'est important :

une fonction lisse a une dérivée qui varie en douceur, ce qui rend l'optimisation plus stable et prévisible.

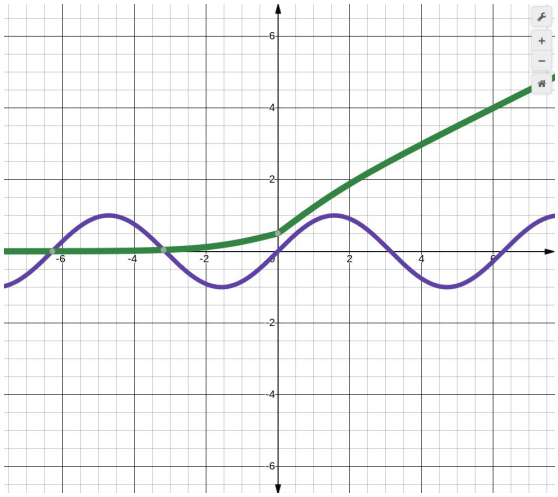
ReLU, malgré sa cassure, reste très utilisée en pratique car ce défaut est mineur comparé à ses avantages.

How to choose the right activation function ?



Here are the criteria to take in account :

- Range
- Smoothness
- **Monotonicity**
- Zero-centered
- Monotonic derivative
- Computational cost



— Monotonous
— Not monotonous

Monotonicity (monotonie)

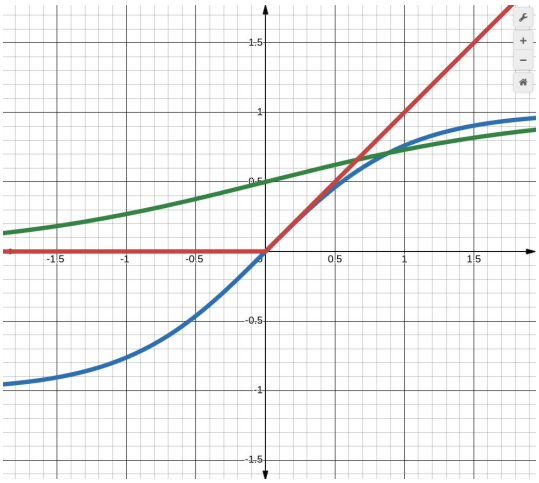
- Est-ce que la fonction est toujours croissante, sans jamais changer de sens ?
- Toutes les fonctions qu'on a vues le sont.
- Pourquoi c'est important :
- ça garantit qu'une entrée plus grande donne toujours une sortie plus grande (ou égale)
- —> ça simplifie l'apprentissage, car le sens de la relation entre entrée et sortie reste cohérent, sans comportement erratique.

How to choose the right activation function ?



Here are the criteria to take in account :

- Range
- Smoothness
- Monotonicity
- **Zero-centered**
- Monotonic derivative
- Computational cost



— tanh(x) : zero-centered
— ReLU : not zero-centered
— sigmoïd : not zero-centered

Zero-centered (centrée en zéro)

- Est-ce que la fonction est centrée en zero ?
- C'est le cas de tanh

<u>Pourquoi c'est important :</u>

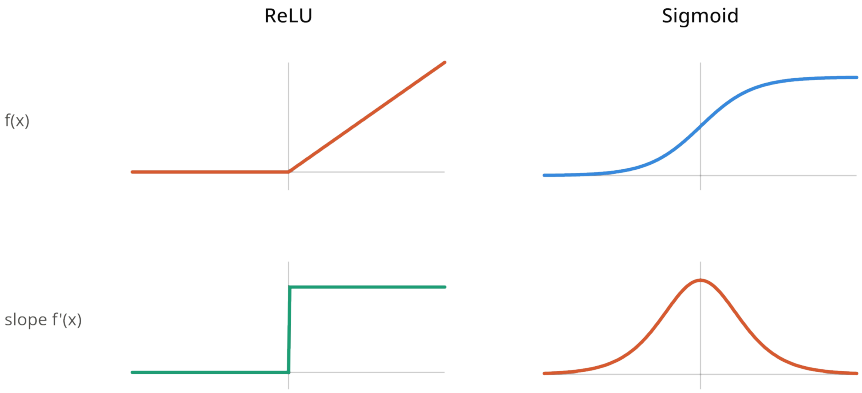
- quand les sorties d'une couche sont toujours positives —> ça peut ralentir l'apprentissage de la couche suivante, car les mises à jour de poids sont contraintes dans une seule direction.
- Une fonction centrée en zéro évite ce biais et accélère souvent la convergence.

How to choose the right activation function ?



Here are the criteria to take in account :

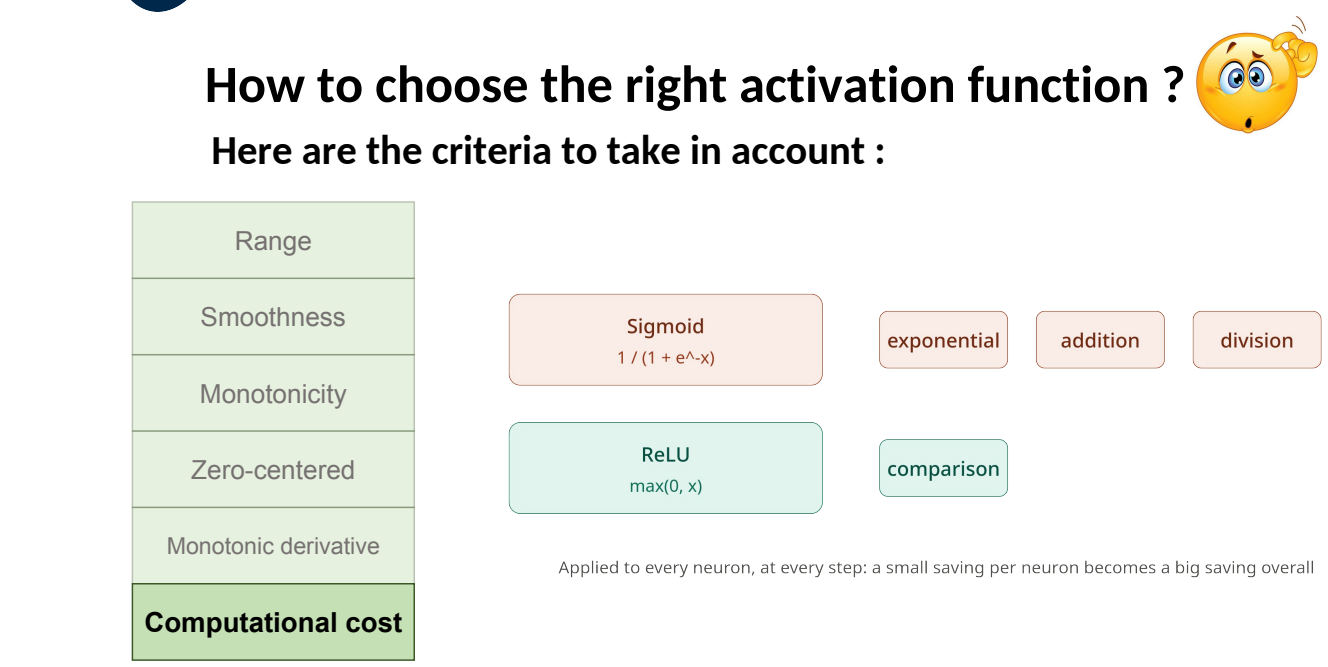
- Range
- Smoothness
- Monotonicity
- Zero-centered
- **Monotonic derivative**
- Computational cost



Monotonic derivative (dérivée monotone)
est-ce que la dérivée elle-même varie de façon régulière, toujours croissante ou toujours décroissante ?

Pourquoi c'est important :

ça influence la stabilité de l'apprentissage —> une dérivée qui se comporte de façon prévisible facilite l'optimisation alors qu'une dérivée irrégulière peut rendre l'entraînement plus chaotique.



Computational cost (coût de calcul)
Certaines fonctions, comme la sigmoïde ou tanh, nécessitent de calculer une exponentielle (opération relativement coûteuse).

- ReLU, elle, n'est qu'un simple maximum entre 0 et x (quasi instantané).

<u>Pourquoi c'est important :</u>

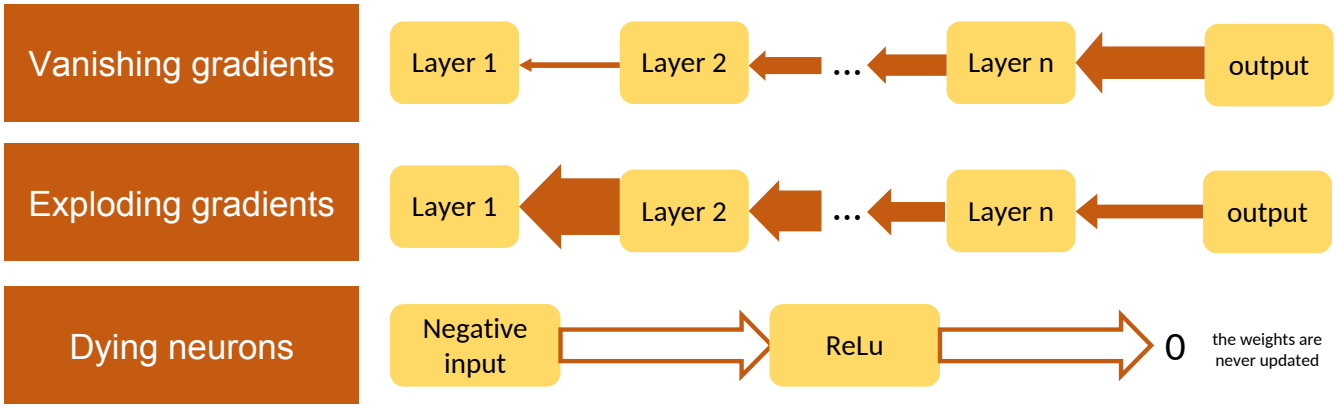
- un réseau contient souvent des millions de neurones, donc chaque fonction d'activation est appliquée un nombre gigantesque de fois.
- Un petit gain de rapidité par neurone se traduit par un gain de temps énorme à l'échelle du réseau entier
- —> c'est une des raisons du succès de ReLU dans les grands modèles.

Le bon choix dépend du problème : la sigmoïde pour une probabilité en sortie, ReLU pour la rapidité au sein du réseau, et ainsi de suite.

Problems to avoid

The Gradient

The slope of the cost function with respect to each weight; it tells the model in which direction, and by how much, to adjust its weights to reduce the error. During training, it is computed layer by layer, from the output back to the input (backpropagation).



- **Le gradient : c'est la pente de la fonction de coût par rapport à chaque poids du modèle, autrement dit, sa dérivée.**
- Il nous dit deux choses :
  - dans quelle direction ajuster le poids pour réduire l'erreur,
  - et de quelle ampleur (indique comment ajuster les poids pour réduire l'erreur.).
  - Pour un réseau à plusieurs couches, ce gradient se calcule couche par couche, en partant de la sortie et en remontant vers l'entrée —> c'est ce qu'on appelle la **rétropropagation**. À chaque couche, le gradient est multiplié par la dérivée de la fonction d'activation de cette couche.
- c'est justement lors de ce trajet que les problèmes suivants peuvent survenir

Vanishing gradients

- À chaque couche traversée —> il est multiplié par une petite valeur,
- (Alégorie du tuyau d'eau)
- —> Le résultat : en arrivant à la couche 1, le gradient est devenu quasi nul.
- Cette 1ère couche reçoit donc une information d'ajustement minuscule, et n'apprend presque plus, peu importe le nombre d'itérations.

Exploding gradients

problème inverse : dû à des instabilités et au fait qu'à chaque couche, le gradient est cette fois multiplié par une grande valeur

- Le gradient devient alors énorme, ce qui provoque des mises à jour de poids disproportionnées
- l'entraînement devient instable, et peut même diverger au lieu de converger vers une solution.

Dying neurons

problème spécifique à ReLU.

- Si un neurone reçoit toujours une entrée négative, ReLU renvoie systématiquement 0.
- Or, la dérivée de ReLU pour une entrée négative est aussi 0 , donc le gradient associé à ce neurone est nul, et ses poids ne sont plus jamais mis à jour.
- Contrairement au vanishing gradient qui ralentit juste l'apprentissage, ici le neurone reste bloqué en permanence, même sur de nouvelles données : il est littéralement 'mort'.

- **Linear systems**
  - LU, QR, Cholesky, Jacobi, Gauss-Seidel, CG, PCG, ...
- **Non-linear systems**
  - First order : Gradient Descent, SGD
  - Second order : Newton, Gauss-Newton, LM, (L)BFGS
- **Others**
  - Genetic algorithms, Metropolis-Hastings, ...
  - Complex and constrained solver : ADMM, Primal-Dual, ...

Il existe plusieurs autres méthodes d'optimisation.

Problème linéaire (notre régression linéaire, convexe) → solution directe, via des méthodes d'algèbre linéaire (LU, Cholesky...).

Problème non-linéaire (ex : réseau de neurones) → pas de solution directe, il faut itérer :

- 1er ordre (Gradient Descent, SGD) : utilise la pente
- 2nd ordre (Newton, BFGS...) : utilise aussi la courbure, plus précis mais plus coûteux

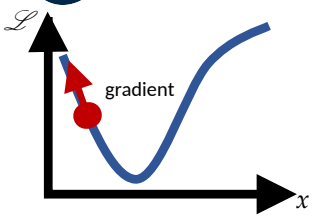
Autres approches : algorithmes génétiques, méthodes probabilistes (sans gradient), solveurs pour problèmes contraints.

Le choix dépend du problème, du matériel disponible, et souvent de l'expérimentation.

On ne choisit pas systématiquement la descente de gradient lorsque le problème est simple.

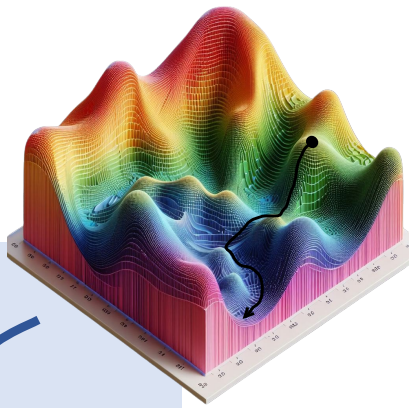
Le choix du solveur est déterminé par :

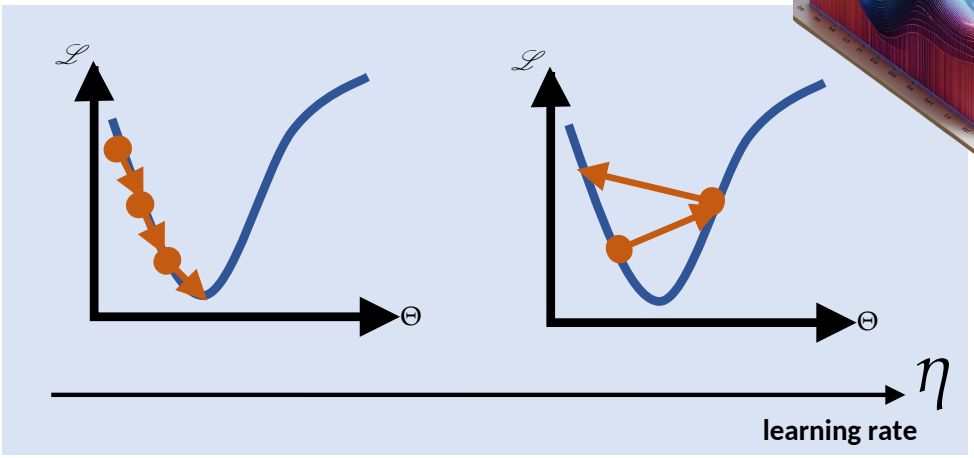
- Le type de problème
- Le matériel informatique disponible
- La curiosité et l'expérimentation scientifique



Learning rate η

A tuning hyperparameter in machine learning that controls the step size taken during each iteration of an optimization algorithm (such as gradient descent) as it moves toward the minimum of a loss function.





Gradient (schéma en haut à gauche)

Pointe vers où la fonction monte le plus vite

On veut minimiser, donc on va dans le sens opposé : d'où "descente" de gradient

Pourquoi l'utiliser ?

Certains problèmes n'ont pas de solution directe

Ou elle est trop difficile à calculer

Question clé : quelle taille de pas à chaque itération ?

C'est le rôle du taux d'apprentissage η (learning rate)

η faible (schéma du milieu)

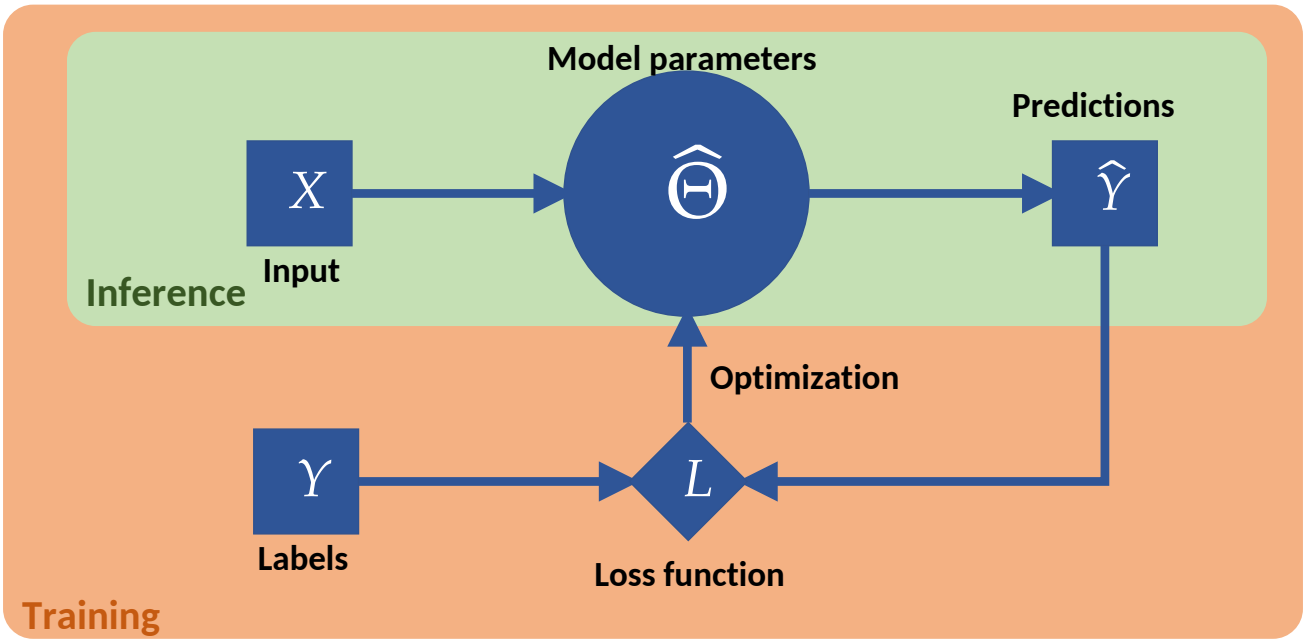
Petits pas : très lent

Risque de ne jamais converger, ou de rester bloqué dans un minimum local

η élevé (schéma de droite)

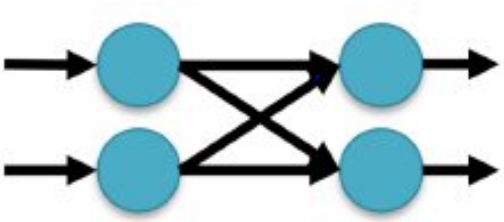
Grands pas qui sautent par-dessus le minimum

Oscillations, pas de convergence



Ce n'est pas à chaque fois qu'on utilise le modèle, mais seulement pendant une phase précise.
En effet le modèle fonctionne en deux modes :

- **Inférence**
  - Mode de fonctionnement normal lors de l'utilisation du modèle
- **Entraînement**
  - Mode de mise à jour des paramètres du modèle à chaque itération
  - Plus lent
  - Consomme plus de mémoire

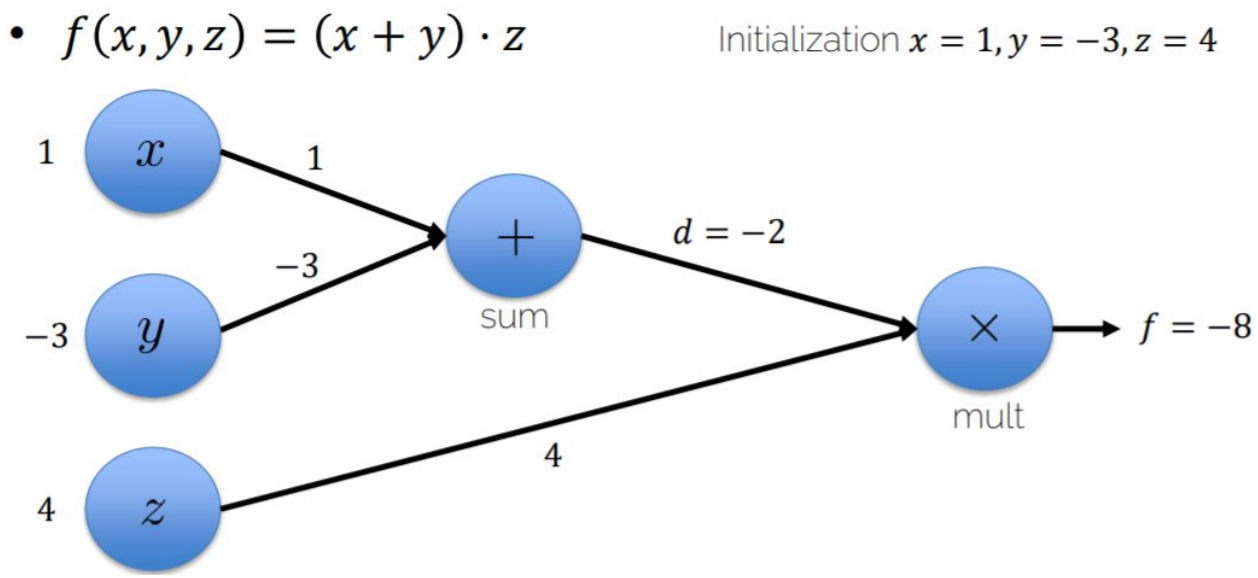


$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial d} \cdot \frac{\partial d}{\partial x}$$

$$\frac{\partial L}{\partial w_{i,k}} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial w_{i,k}}$$

Comment mettre à jour individuellement chaque paramètre ?

—> on utilise ce qu'on appelle la règle de la chaine



Prenons un exemple très simple pour illustrer le graphe de calcul

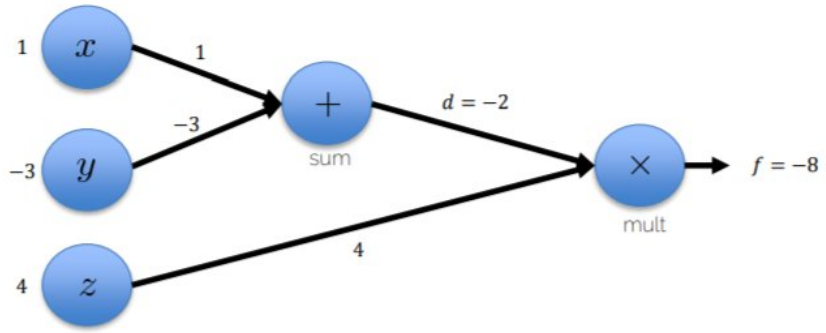
- On suit les flèches dans leur sens normal —> c'est le forward pass.
- D'abord 'sum' additionne x et y : 1 + (-3) = -2, qu'on appelle d.
- Puis 'mult' multiplie d par z : -2 × 4 = -8. C'est notre résultat final.

C'est exactement ce que fait un réseau pour produire une prédiction : les données traversent le réseau dans l'ordre normal, jusqu'à la sortie.

Backward pass

$$f(x,y,z) = (x+y)\cdot z$$

with $x = 1, y = -3, z = 4$



$$d = x + y \qquad \frac{\partial d}{\partial x} = 1, \frac{\partial d}{\partial y} = 1$$

$$f = d \cdot z \qquad \frac{\partial f}{\partial d} = z, \frac{\partial f}{\partial z} = d$$

What is $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$?

On garde le même exemple,
avec un nœud “sum” qui calcule d = x+y,
et un nœud “mult” qui calcule f = d·z.

On connaît déjà la valeur de chaque variable après le forward pass.

- pour ‘sum’, dérivée = 1 par rapport à x et à y.
- Pour ‘mult’, dérivée = z par rapport à d, et d par rapport à z.

Backward pass

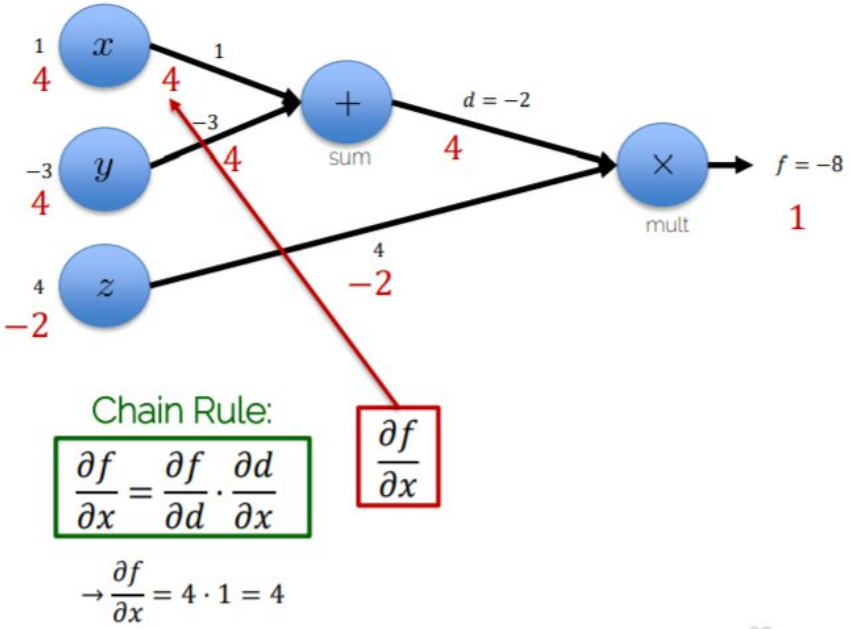
$f(x,y,z) = (x+y)\cdot z$

with $x = 1, y = -3, z = 4$

$d = x + y$ $\quad \frac{\partial d}{\partial x} = 1 \quad \frac{\partial d}{\partial y} = 1$

$f = d \cdot z$ $\quad \frac{\partial f}{\partial d} = z, \frac{\partial f}{\partial z} = d$

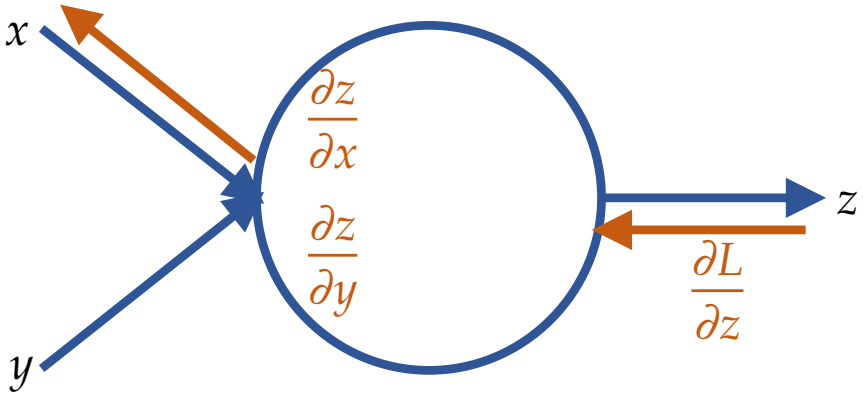
What is $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$?



- Pour trouver ∂f/∂x, ∂f/∂y, ∂f/∂z, on remonte le graphe avec la chain rule.
  - Pour z, c'est direct : ∂f/∂z = d = -2.
  - Pour x et y, il faut passer par d : ∂f/∂x = ∂f/∂d × ∂d/∂x = 4 × 1 = 4, et pareil pour y.

C'est ça, le backward pass : remonter le graphe nœud par nœud, en multipliant les dérivées locales.

$$\frac{\partial L}{\partial x} = \frac{\partial z}{\partial x}\frac{\partial L}{\partial z}$$



Cette règle peut être appliquée dans un réseau de neurones.

Limitations :

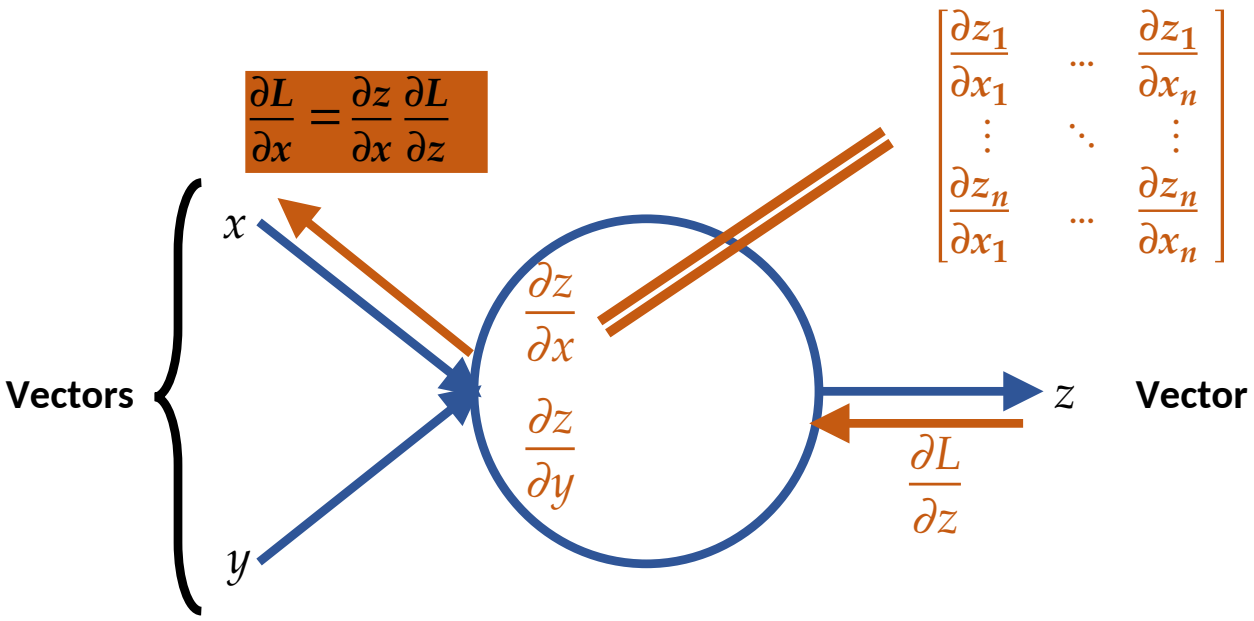
- Nécessite d'avoir les activations de chaque neurone en mémoire

Il existe deux types de fonctions dans nos implémentations :

- **Forward** pour le calcul de la sortie pour une donnée
- **Backward** pour la rétropropagation de l'erreur pour les poids

Activations en mémoire :

- = les valeurs de sortie de chaque couche (le résultat du calcul après la fonction d'activation), que l'on garde en mémoire pendant le forward pass.
- —> Lors du backward pass, on a besoin de ces valeurs pour calculer les dérivées locales.
- Conséquence : plus le réseau est profond et le lot grand, plus il y a d'activations à stocker. C'est un des principaux postes de consommation de mémoire pendant l'entraînement (à côté des poids et des jacobiennes).
- Ce qu'on garde en mémoire, ce sont donc les activations de chaque couche, pas les jacobiennes. C'est ce qui rend la rétropropagation faisable en pratique.

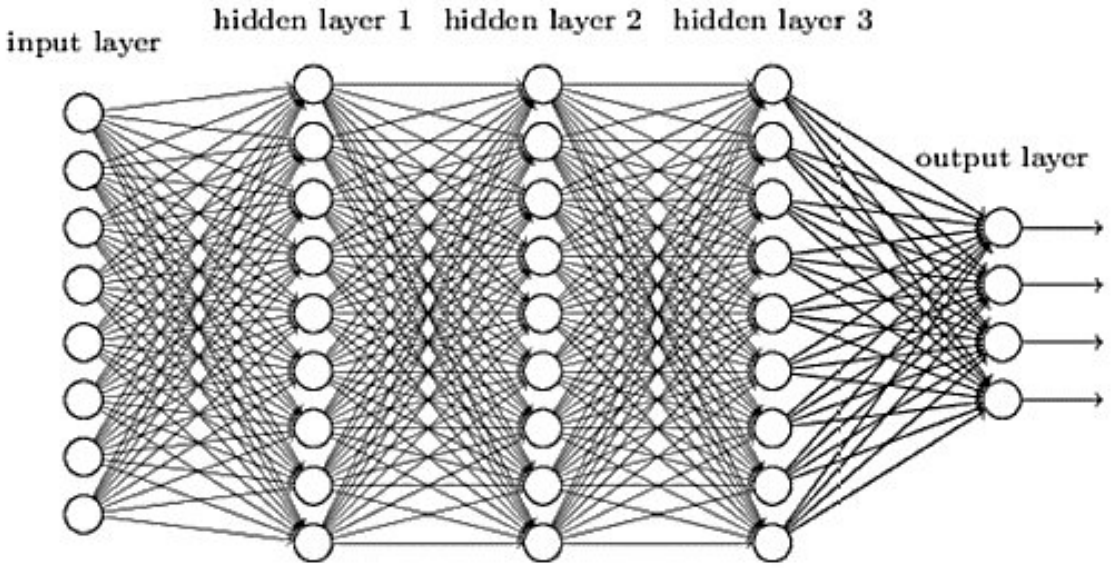


Mais dans un vrai réseau, chaque couche manipule des vecteurs entiers —> par exemple 4096 valeurs à la fois.

La dérivée de z par rapport à x devient alors toute une matrice, qu'on appelle la **matrice jacobienne** : elle contient la dérivée de chaque sortie par rapport à chaque entrée.

- Pb —> la taille de cette matrice.
- Avec des vecteurs de 4096 valeurs, la jacobienne contient déjà 16 millions de nombres —> environ 64 Mo de mémoire pour relier seulement deux couches.
- Et comme on entraîne généralement par lots de plusieurs exemples à la fois (ici 16) cette taille explose encore, jusqu'à environ 16 Go, juste pour une seule étape de calcul.

C'est pour cette raison qu'en pratique, on ne stocke jamais cette matrice jacobienne complète en mémoire —> les bibliothèques comme PyTorch calculent directement le résultat final du produit, sans jamais la matérialiser entièrement.



- plusieurs couches cachées empilées, bien plus complexe que le simple neurone par lequel on a commencé.
- [schéma] Réseau fully connected : chaque neurone d'une couche est relié à tous les neurones de la couche suivante
- On parle aussi de réseau dense
- c'est l'option par défaut —> on relie tout, car on ne sait pas à l'avance quelles connexions seront utiles. C'est coûteux, mais simple

Plus les données et les problèmes sont complexes, plus il faut de profondeur et de largeur pour les modéliser correctement.

Mais cette complexité a un prix :

- plus de couches —> c'est plus de paramètres à stocker et calculer, on vient de voir que la mémoire explose vite.
- C'est aussi plus de risques de gradients qui explosent ou disparaissent, puisque le gradient doit traverser davantage de couches lors de la rétropropagation.

Pour conclure

Construire un bon réseau, c'est donc toujours un compromis entre la capacité du modèle à bien représenter les données, et les contraintes pratiques —> mémoire disponible, stabilité de l'entraînement

References

- Chollet, Francois. *Deep learning with Python*. Simon and Schuster, 2021.
- *CS230 Deep Learning*. cs230.stanford.edu. Accessed 14 Mar. 2022.
- *I2DL*. niessner.github.io/I2DL. Accessed 14 Mar. 2022.
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

Definitions

Artificial Intelligence

Systems capable of reproducing human actions or decisions

Machine Learning

Algorithms mainly based on statistical methods, often with an iterative aspect from which comes the term «Learning »

Deep Learning

A group of models based on logical units called neurons and distributed in layers

Discrete Data

data that can only take specific, separate values, usually whole numbers or categories you can count

Continuous Data

data that can take any value within a range, including fractions and decimals, usually something you measure

A neuron

A small computing unit that takes several inputs, combines them using weighted parameters, and passes the result through an activation function to produce an output.

Activation function

A non-linear function applied to the weighted sum of a neuron's inputs, allowing the network to model non-linear relationships

Loss function

Measures the error **for a single observation i** : the gap between the predicted value $\hat{y}_i$ and the true value y_i, for that specific data point.

Cost function

the average of the loss **over the entire dataset** (but computed from the loss rather than directly)

The Gradient

The slope of the cost function with respect to each weight; it tells the model in which direction, and by how much, to adjust its weights to reduce the error. During training, it is computed layer by layer, from the output back to the input (backpropagation).

Learning rate η

A tuning hyperparameter in machine learning that controls the step size taken during each iteration of an optimization algorithm (such as gradient descent) as it moves toward the minimum of a loss function.