

# Introduction à l'apprentissage profond

## Modèles actuariels en assurance non-vie

---

par Francis Duval

à l'Université du Québec à Montréal

le 14 mars 2023

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

## ARTIFICIAL INTELLIGENCE

Any technique that enables computers to mimic human behavior



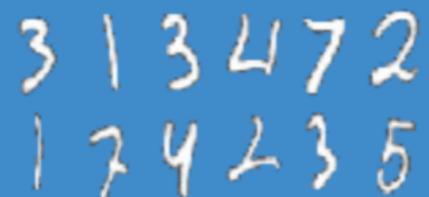
## MACHINE LEARNING

Ability to learn without explicitly being programmed



## DEEP LEARNING

Extract patterns from data using neural networks



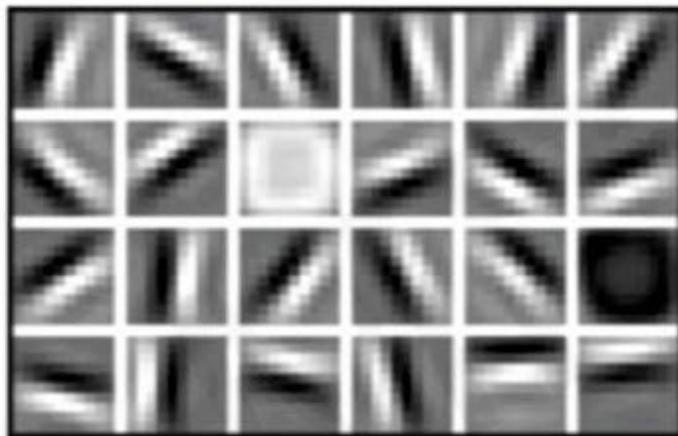
- L'apprentissage profond est la grande majorité du temps effectué avec des **réseaux de neurones profonds**, c'est-à-dire des réseaux de neurones avec plusieurs couches cachées.

# Pourquoi l'apprentissage profond?

Hand engineered features are time consuming, brittle, and not scalable in practice

Can we learn the **underlying features** directly from data?

Low Level Features



Lines & Edges

Mid Level Features



Eyes & Nose & Ears

High Level Features



Facial Structure

# Pourquoi l'apprentissage profond?

## Exemple: on veut tarifer les assurés avec leurs données télématiques

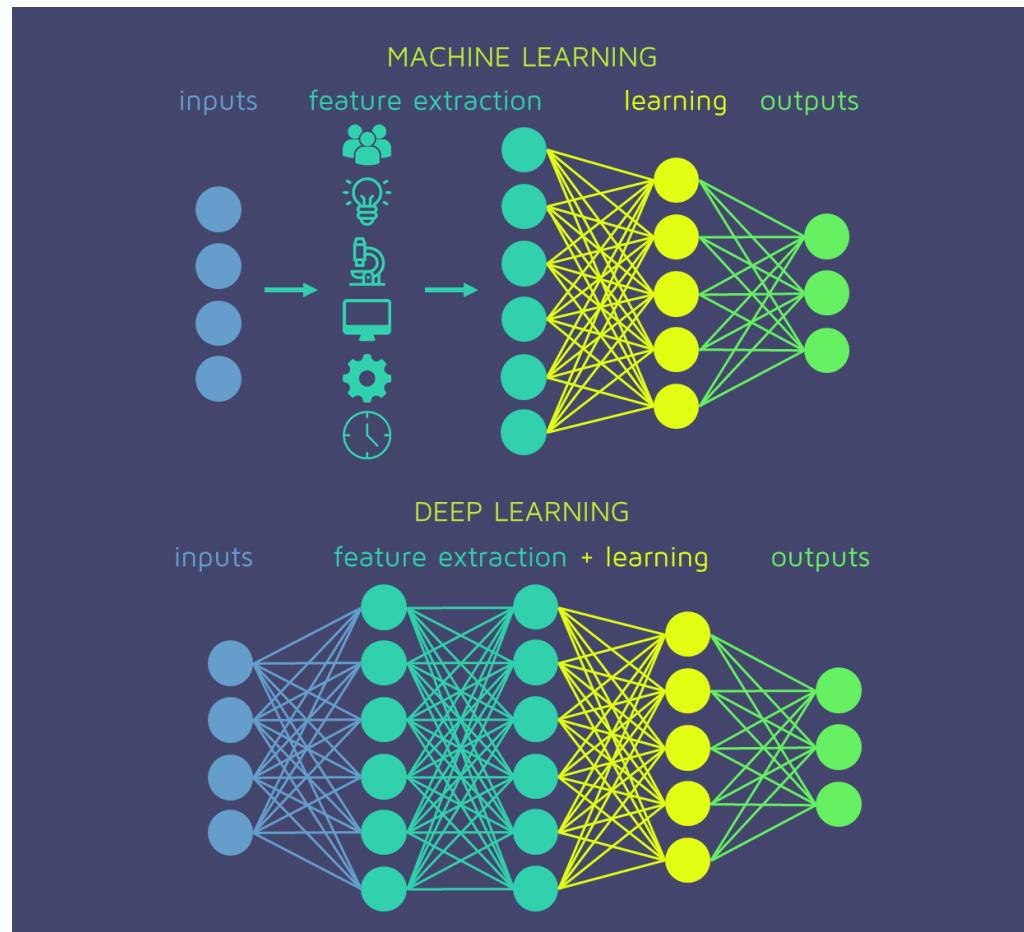
### Machine learning

- On va extraire des features « à la main » à partir des données brutes. Par exemple:
  - nb. freinages brusques
  - nb. accélérations brusques
  - % conduite le jour/soir/nuit
  - vitesse moyenne
  - etc.
- On utilise ensuite ces features extraites « à la main » dans un algorithme de machine learning, comme un GLM.

### Deep learning

- On donne typiquement les données sous un format plus brute à l'algorithme de deep learning. Par exemple, on pourrait lui donner:
  - les données de trajets seconde-par-seconde sous un certain format
  - les résumés de trajets sous un certain format
- Celui-ci va lui-même créer les features qu'il trouve pertinentes.
- Noter qu'il y a quand même un certain travail d'ingénierie de données à faire. Par exemple, il est peu probable qu'une date (stockée sous R en nombre de jours depuis 1970) soit pertinente pour la tarification.

# Pourquoi l'apprentissage profond?

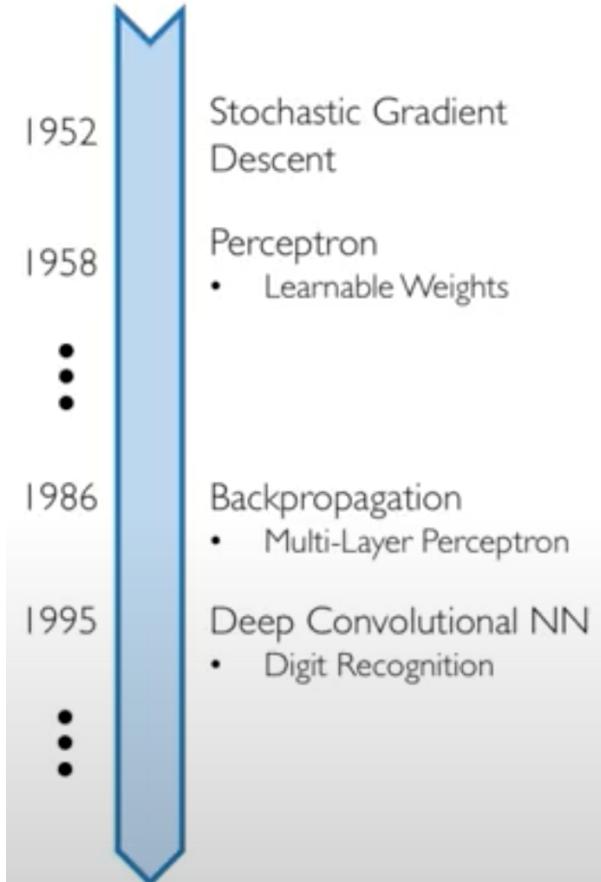


**Machine learning**

**vs**

**Deep learning**

# Petit historique



- Aussi, plusieurs avancées théoriques depuis quelques décennies: **backpropagation**, résolution du **vanishing (ou exploding) gradient problem**, etc.

## Pourquoi maintenant?

### 1. Données massives

- Plus facile de nos jours de collecter et stocker de grandes quantités de données.

### 2. Hardware

- Puissance de calcul a beaucoup augmenté (et a diminué en prix) depuis quelques décennies.
- GPUs (Graphics Processing Units)

### 3. Software

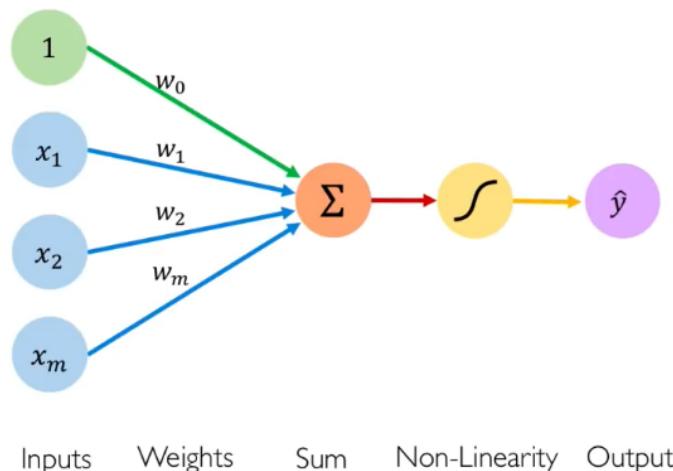
- Nouveaux logiciels qui facilite l'implémentation de modèles de deep learning sont apparus:
- Torch for R
- PyTorch
- TensorFlow
- etc.

# Qu'est-ce qu'un réseau de neurones (concrètement)?

## Perceptron

Le perceptron est l'instance la plus simple d'un réseau de neurones. Il est en quelque sorte la « brique » qui permet de construire des réseaux de neurones plus complexes.

### The Perceptron: Forward Propagation

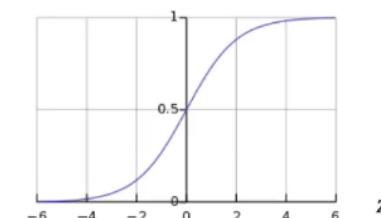


#### Activation Functions

$$\hat{y} = g(w_0 + \mathbf{X}^T \mathbf{W})$$

- Example: sigmoid function

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



# Qu'est-ce qu'un réseau de neurones (concrètement)?

## Exemple

- Un actuaire veut estimer la probabilité de réclamer pour des assuré.es en se basant sur 2 prédicteurs (inputs), c'est-à-dire l'**âge de l'assuré.e** et l'**âge de son véhicule**.
- L'actuaire considère un perceptron avec une fonction d'activation **sigmoïde**. Après avoir entraîné le perceptron sur une base de données, il obtient les poids suivants:
  - $w_0 = -2$ ,
  - $w_{\text{age\_ass}} = -0.02$ ,
  - $w_{\text{age\_veh}} = -0.01$ .

---

**Exercice:** Estimer la probabilité de réclamer pour un assuré de 25 ans avec un véhicule de 5 ans.

# Qu'est-ce qu'un réseau de neurones (concrètement)?

## Ajout de couches cachées

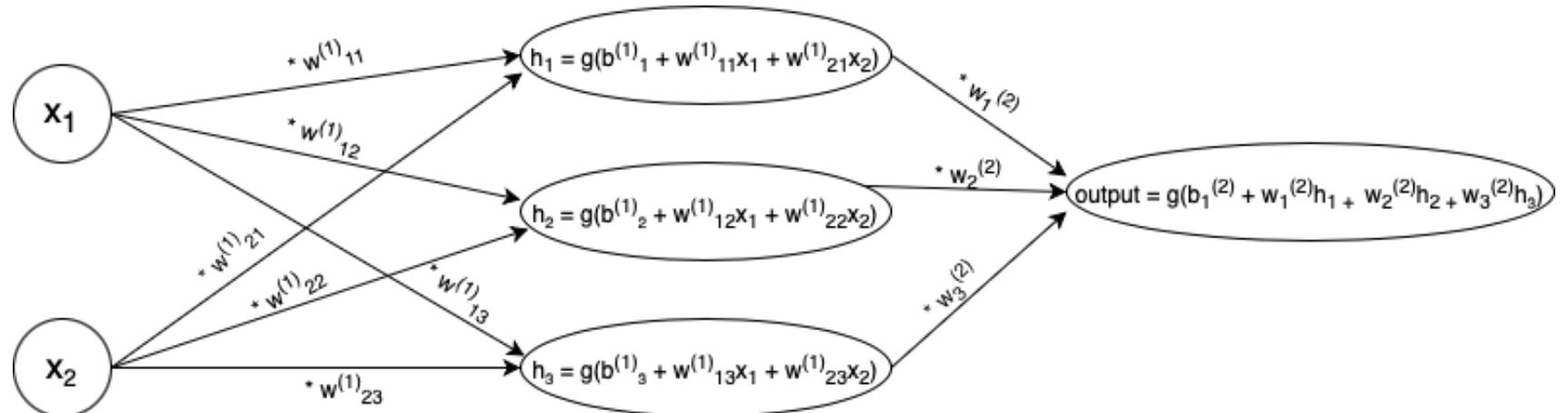
- Un perceptron est bien, mais ne permet qu'une fonction de prédiction linéaire (par rapport aux inputs).
- Ajouter des couches cachées permet d'obtenir un fonction de prédiction bien plus flexible. Lorsqu'il y a au moins 1 couche cachée, on parle de **perceptron multicouches**.

## Exercice

- Aller sur le site [playground.tensorflow.org](http://playground.tensorflow.org).
- Entrainer un perceptron avec fonction d'activation **sigmoïde** utilisant les 2 premiers inputs  $x_1$  et  $x_2$  sur **le premier jeu de données de classification**.
  - Remarquer que la fonction de prédiction obtenue est linéaire, alors que la tâche à effectuer ne l'est pas (c'est un cercle).
- Ajouter  $x_1^2$  et  $x_2^2$  comme inputs et entraîner le modèle.
  - Le modèle est maintenant capable de bien séparer les 2 classes, sauf qu'il a fallu penser à créer les 2 nouveaux inputs  $x_1^2$  et  $x_2^2$ .
- Utiliser seulement  $x_1$  et  $x_2$  comme inputs, mais ajouter une **couche cachée à 3 neurones**. Entrainer le modèle.
  - Le modèle a maintenant une fonction de prédiction plus raisonnable.

# Qu'est-ce qu'un réseau de neurones (concrètement)?

## Perceptron multicouches (exemple à 1 couche cachée de 3 neurones cachés)



La fonction de prédiction peut être écrite sous forme matricielle:  $\hat{y} = g \left[ \mathbf{w}^{(2)} \times g \left( \mathbf{w}^{(1)} \mathbf{x} + \mathbf{b}^{(1)} \right) + \mathbf{b}_1^{(2)} \right]$ , où «  $\times$  » est la multiplication matricielle standard et où:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \mathbf{w}^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{21}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} \\ w_{13}^{(1)} & w_{23}^{(1)} \end{bmatrix}, \quad \mathbf{w}^{(2)} = \begin{bmatrix} w_1^{(2)} & w_2^{(2)} & w_3^{(2)} \end{bmatrix}, \quad \mathbf{b}^{(1)} = \begin{bmatrix} b_1^{(1)} & b_2^{(1)} & b_3^{(1)} \end{bmatrix}, \quad \mathbf{b}_1^{(2)} = \begin{bmatrix} b_1^{(2)} \end{bmatrix}.$$

# Entraîner un réseau de neurones

## Fonction de perte

- Entrainer le réseau = trouver de « bonnes » valeurs pour les paramètres  $\mathbf{w}$  et  $\mathbf{b}$ .
- Bonnes valeurs de paramètres  $\rightarrow$  valeurs qui donnent des prédictions  $\hat{y}$  « proches » des réponses  $y$ .
- Pour définir « proche », on doit se choisir une **fonction de perte**.
- Exemples de fonctions de perte:
  - Erreur quadratique (pour  $y \in \mathbb{R}$ ):

$$\ell(y, \hat{y}) = (y - \hat{y})^2$$

- Entropie croisée binaire (pour  $y \in \{0, 1\}$ ):

$$\ell(y, \hat{y}) = [y \ln(\hat{y}) + (1 - y) \ln(1 - \hat{y})]$$

- Moins la log-vraisemblance Poisson pour données de comptage:

$$\ell(y, \hat{y}) = [-\hat{y} + y \ln(\hat{y})]$$

- Le but est alors de minimiser la fonction de perte moyenne sur l'ensemble d'entraînement, qu'on appelle parfois **risque empirique**. On voudra donc trouver les paramètres qui minimisent  $\frac{1}{n} \sum_{i=1}^n \ell(y_i, \hat{y}_i)$ .

# Entraîner un réseau de neurones

## Minimisation du risque empirique

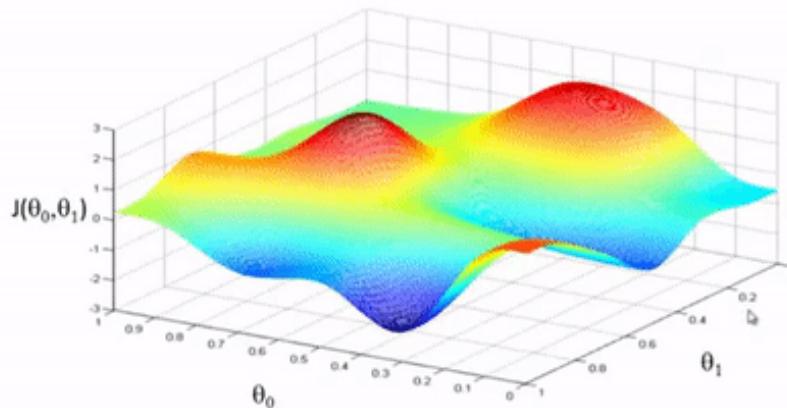
- Le but est donc de trouver les paramètres  $\mathbf{w}^*$  qui minimisent le risque empirique:

$$\begin{aligned}\mathbf{w}^* &= \operatorname{argmin}_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^n \ell(y, \hat{y}) \\ &= \operatorname{argmin}_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^n \ell(y, f(\mathbf{x}_i; \mathbf{w})) \\ &= \operatorname{argmin}_{\mathbf{w}} J(\mathbf{w})\end{aligned}$$

- Avec les GLMs, on dérivait par rapport aux paramètres et on égalait à zéro pour trouver le minimum.
- Cette méthode ne fonctionne pas ici, car rien ne garantit que la fonction de risque empirique  $J$  est convexe!
- On utilise plutôt la **descente de gradient** (et ses dérivées).

# Entraîner un réseau de neurones

## Descente de gradient



Andrew Ng

Fonction de risque empirique en fonction des paramètres. Note: ici, les paramètres sont dénotés par la lettre grecque theta.

### Pseudo-algorithme:

1. Initialiser les paramètres  $(w_0, w_1)$  au hasard
2. Calculer le gradient de la fonction de risque au point  $(w_0, w_1)$
3. Bouger dans la direction opposée du gradient (le gradient pointe vers le haut de la fonction alors qu'on veut aller vers le bas)
4. Répéter jusqu'à convergence

# Entraîner un réseau de neurones

## Descente de gradient

### Algorithme

1. Initialiser les paramètres  $\mathbf{w}$  au hasard  $\sim \mathcal{N}(0, \sigma^2)$
2. Répéter jusqu'à convergence:
  - Calculer le gradient  $\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$
  - Mettre à jour les paramètres:  $\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}$
3. Output: paramètres  $\mathbf{w}$

# Entraîner un réseau de neurones

## Rétropropagation

- Algorithme de **différentiation automatique** pour calculer le gradient de la fonction de risque empirique.
- L'algorithme de rétropropagation utilise la **règle de dérivation en chaîne** pour calculer le gradient, en allant de la fin du réseau jusqu'au début (d'où le « rétro »).

## Exemple

- Si vous avez 500 paramètres (ce qui n'est pas énorme pour un réseau de neurones), le gradient sera un vecteur en 500 dimensions.
- La rétropropagation permet de calculer ce gradient compliqué de manière efficace.

## 2 bonnes vidéos sur la rétropropagation:

- [3Blue1Brown \(sans formules\)](#)
- [3Blue1Brown \(avec formules\)](#)

# Entraîner un réseau de neurones

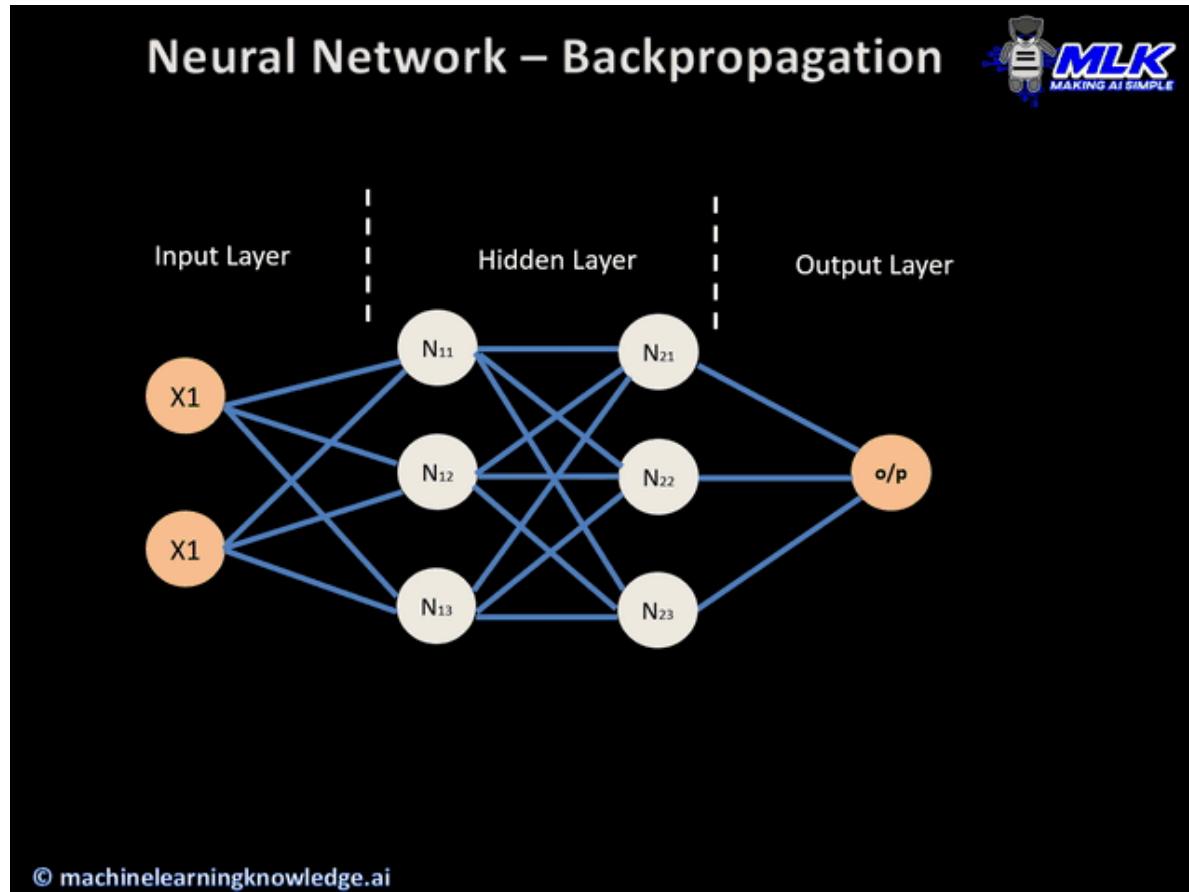
## Récapitulation

Pour entraîner un réseau de neurones, on va:

- **Initialiser** les paramètres au hasard
- Faire une première « **passe forward** » pour obtenir le risque empirique initial
- Appliquer la **rétropropagation** pour calculer le gradient
- Effectuer la **descente de gradient** pour mettre à jour les paramètres
- Faire une deuxième « **passe forward** » pour obtenir la nouvelle valeur du risque empirique
- Appliquer la **rétropropagation** pour calculer le nouveau gradient
- Effectuer la **descente de gradient** pour mettre à jour les paramètres
- etc.

# Entraîner un réseau de neurones

## Récapitulation



# Autres types d'architectures

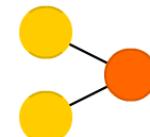
- Dans cette présentation, nous nous sommes concentrés sur le perceptron multicouches dans le cadre de l'apprentissage supervisé.
- Cependant, il existe une panoplie d'autres architectures.

## A mostly complete chart of Neural Networks

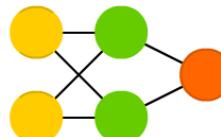
©2019 Fjodor van Veen & Stefan Leijnen asimovinstitute.org

- Input Cell
- Backfed Input Cell
- △ Noisy Input Cell
- Hidden Cell
- Probabilistic Hidden Cell
- △ Spiking Hidden Cell
- Capsule Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell

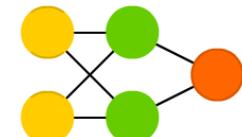
Perceptron (P)



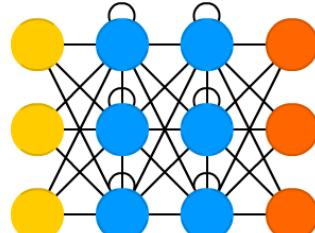
Feed Forward (FF)



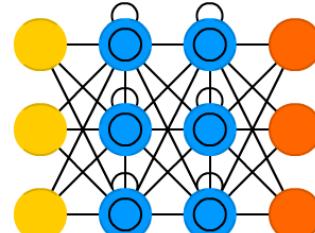
Radial Basis Network (RBF)



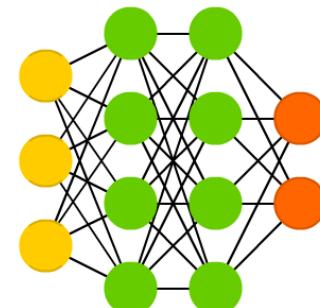
Recurrent Neural Network (RNN)



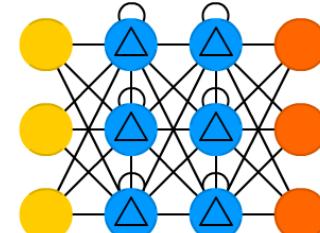
Long / Short Term Memory (LSTM)



Deep Feed Forward (DFF)

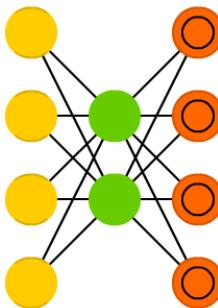


Gated Recurrent Unit (GRU)

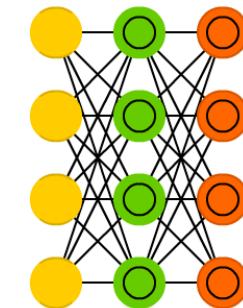


- Recurrent Cell
- Memory Cell
- △ Gated Memory Cell
- Kernel
- Convolution or Pool

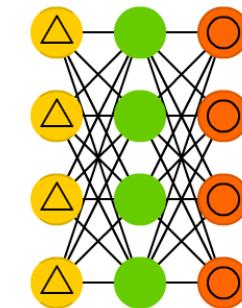
Auto Encoder (AE)



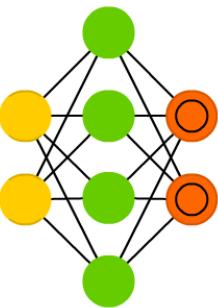
Variational AE (VAE)



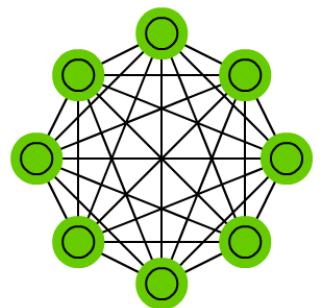
Denoising AE (DAE)



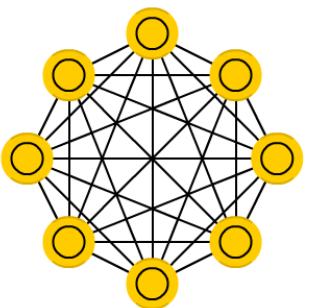
Sparse AE (SAE)



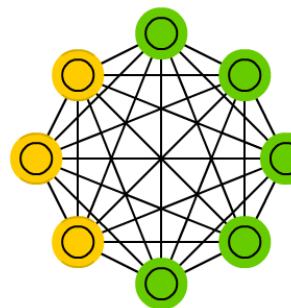
Markov Chain (MC)



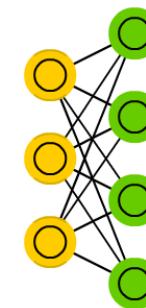
Hopfield Network (HN)



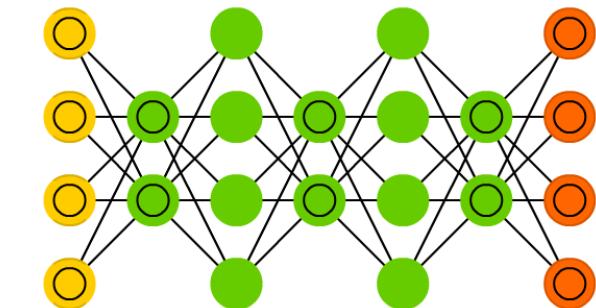
Boltzmann Machine (BM)



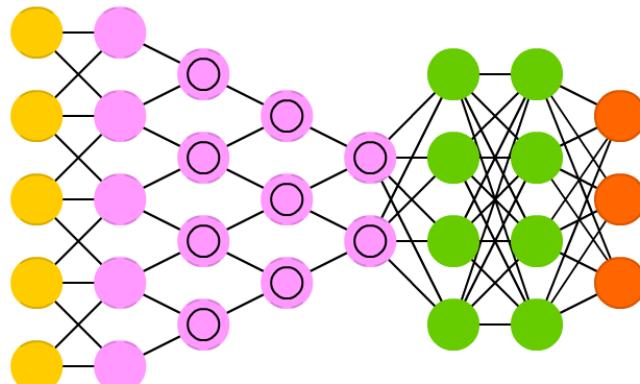
Restricted BM (RBM)



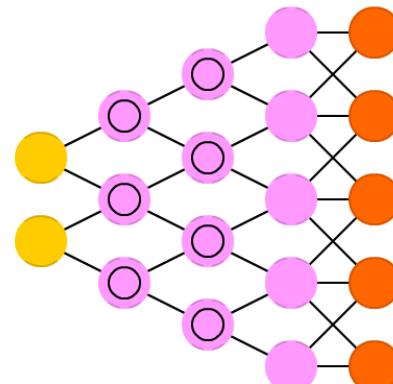
Deep Belief Network (DBN)



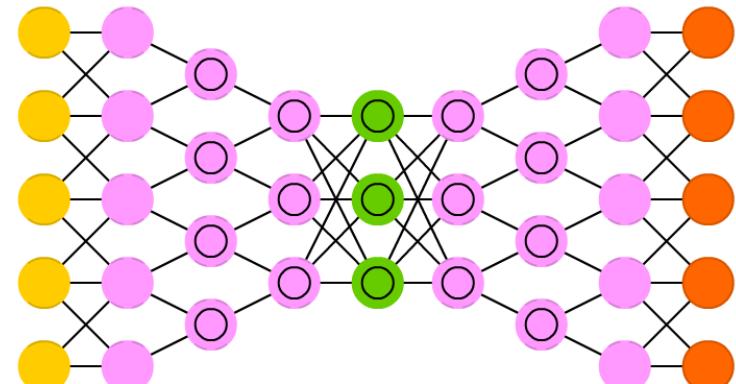
Deep Convolutional Network (DCN)



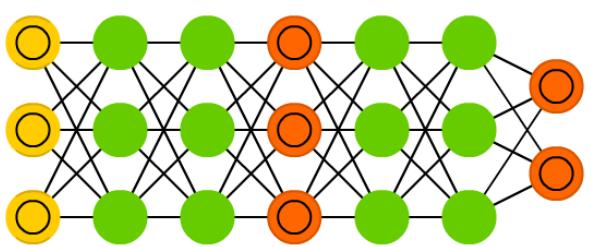
Deconvolutional Network (DN)



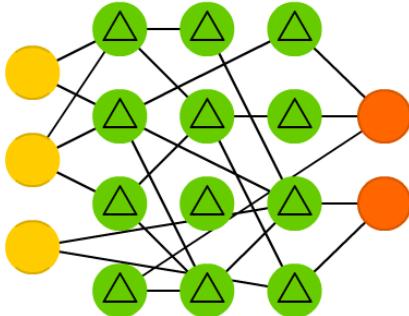
Deep Convolutional Inverse Graphics Network (DCIGN)



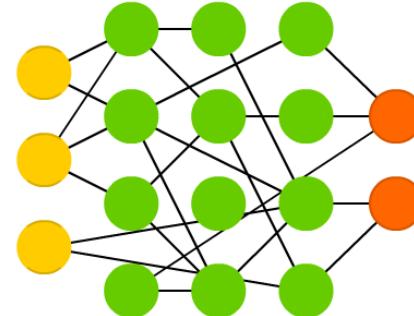
Generative Adversarial Network (GAN)



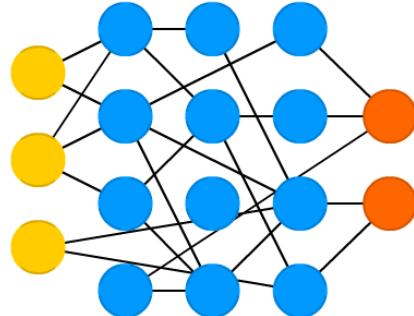
Liquid State Machine (LSM)



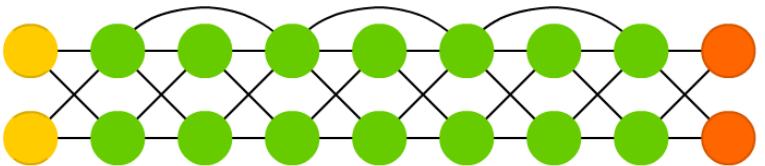
Extreme Learning Machine (ELM)



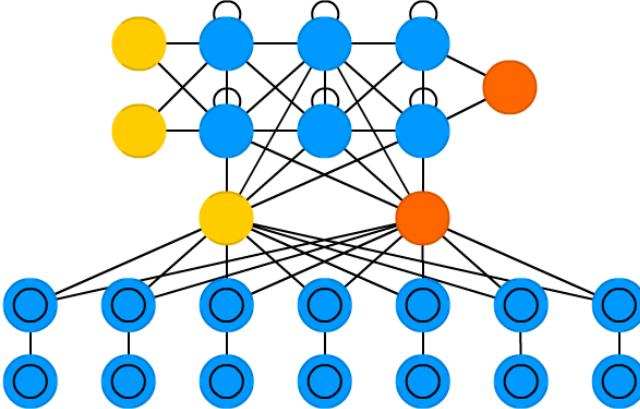
Echo State Network (ESN)



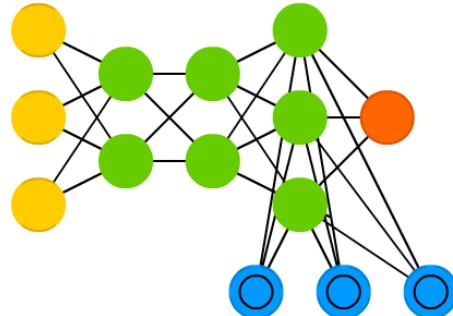
Deep Residual Network (DRN)



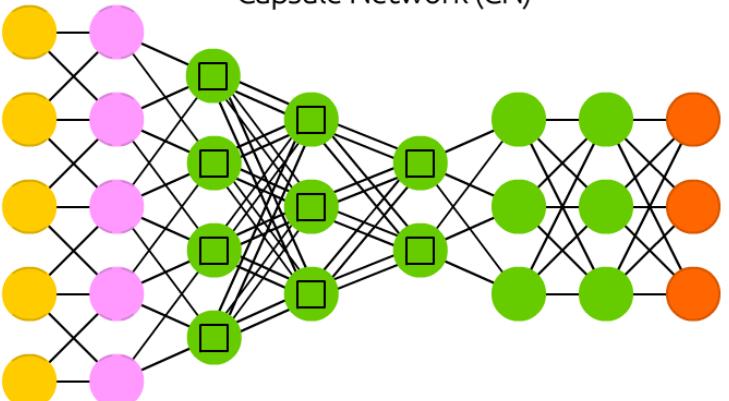
Differentiable Neural Computer (DNC)



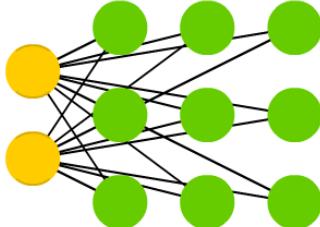
Neural Turing Machine (NTM)



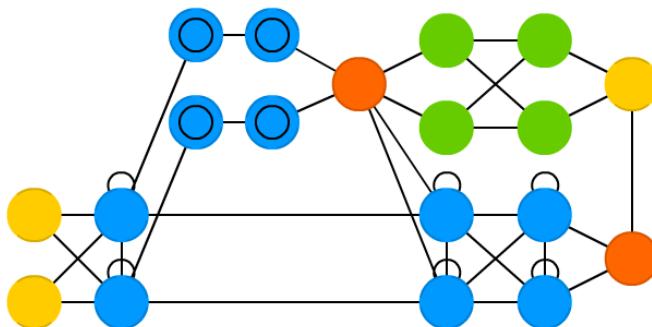
Capsule Network (CN)



Kohonen Network (KN)



Attention Network (AN)



# Application - tarification avec données télématiques

## Données

Véhicule	Numéro de trajet	Date-heure de départ	Date-heure d'arrivée	Distance	Vitesse maximale
A	1	2017-05-02 19:04:15	2017-05-02 19:24:24	25	104
A	2	2017-05-02 21:31:29	2017-05-02 21:31:29	6.4	66
:	:	:	:	:	:
A	2320	2018-04-30 21:17:22	2018-04-30 21:18:44	0.2	27
B	1	2017-03-26 11:46:07	2017-03-26 11:53:29	1.5	76
B	2	2017-03-26 15:18:23	2017-03-26 15:51:46	35.1	119
:	:	:	:	:	:
B	1485	2018-03-23 20:07:08	2018-03-23 20:20:30	10.1	92
:	:	:	:	:	:

# Application - tarification avec données télématiques

## Problème

**Supposons que l'on veuille faire de la tarification par véhicule. On pourrait alors créer « à la main » plusieurs variables explicatives, ou features, avec les données télématiques. Par exemple:**

- Distance moyenne par jour
- Nombre de trajets moyen par jour
- Vitesse moyenne médiane
- Vitesse maximale médiane
- Proportion de longs trajets (disons > 100km)
- Fraction de la conduite:
  - Le matin
  - L'après-midi
  - Le soir
  - La nuit
  - À l'heure de pointe
  - etc.
- etc.

**Cependant, ces features sont créées de manière arbitraire et ne sont probablement pas optimale pour la tâche de tarification.**

# Application - tarification avec données télématiques

## Solution proposée

Une solution est de donner en entrée à un réseau de neurones les données télématiques sous un format plus brute afin que celui-ci crée lui-même des features pertinentes.

### Ingénierie des données

- On se définit les vecteurs suivants:

$\mathbf{h} = (h_1, h_2, \dots, h_{24})$ , où  $h_i$  est la fraction de la conduite dans la  $i^{\text{e}}$  heure de la journée

$\mathbf{p} = (p_1, p_2, \dots, p_7)$ , où  $p_i$  est la fraction de la conduite dans la  $i^{\text{e}}$  journée de la semaine

$\mathbf{vmo} = (vmo_1, vmo_2, \dots, vmo_{14})$ , où  $vmo_i$  est la fraction de trajets dans le  $i^{\text{e}}$  intervalle de vitesse moyenne

$\mathbf{vma} = (vma_1, vma_2, \dots, vma_{16})$ , où  $vma_i$  est la fraction de trajets dans le  $i^{\text{e}}$  intervalle de vitesse maximale

- Ensuite, on concatène ces 4 vecteurs en un gros vecteur d'inputs de dimension  $24 + 7 + 14 + 16 = 61$  qu'on va donner en entrée à un perceptron multicouches:

$\mathbf{input} = (\mathbf{h}, \mathbf{p}, \mathbf{vmo}, \mathbf{vma})$

# Application - tarification avec données télématiques

