

# Apprentissage supervisé, troisième partie : les boîtes noires

# Les méthodes dites "boîtes noires"

Nous parlerons dans ce cours de deux types de méthodes : les **SVM** et les **réseaux de neurones**. Elles ont ceci en commun qu'on parle de ces méthodes comme des **boîtes noires** :

- elles reposent sur des calculs complexes
- les résultats sont difficilement interprétables
- elles sont populaires car performantes

Le modèle de **Random Forests** (cours 7) peut aussi être considéré comme une boîte noire.

De manière générale, les avis sur la question "tel modèle est-il une boîte noire?" ne trouve pas de consensus (comme toute question qui implique l'interprétation humaine).

1 Support Vector Machines (SVM)

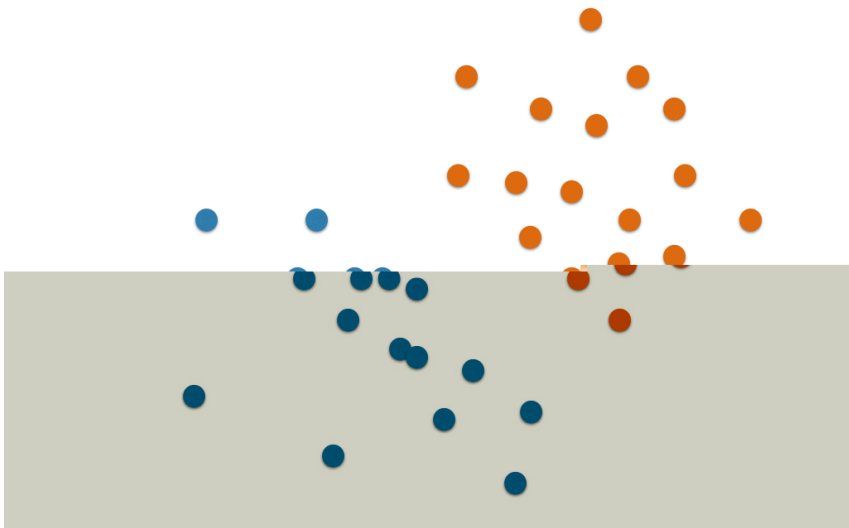
2 Réseaux de neurones

L'arrivée des **SVMs** en **1992** marque un tournant dans l'histoire du **machine learning**.

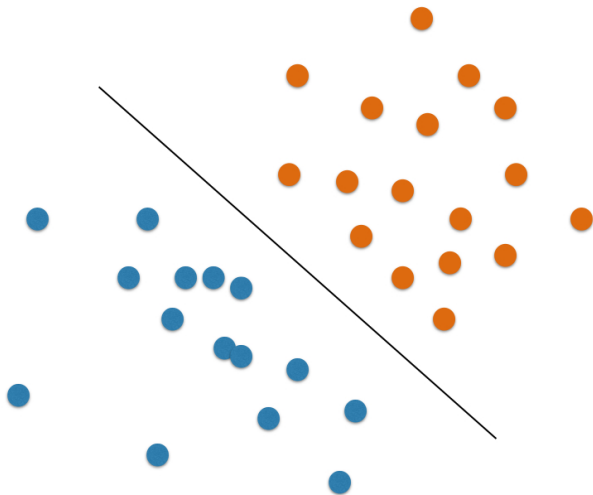
La théorie de laquelle ces méthodes sont issues est alors nouvelle, intuitive et permet de résoudre des problèmes complexes dans un **nouveau paradigme**.

Les notions mathématiques à la base des SVMs sont difficiles. Nous n'aborderons dans ce cours que les concepts.

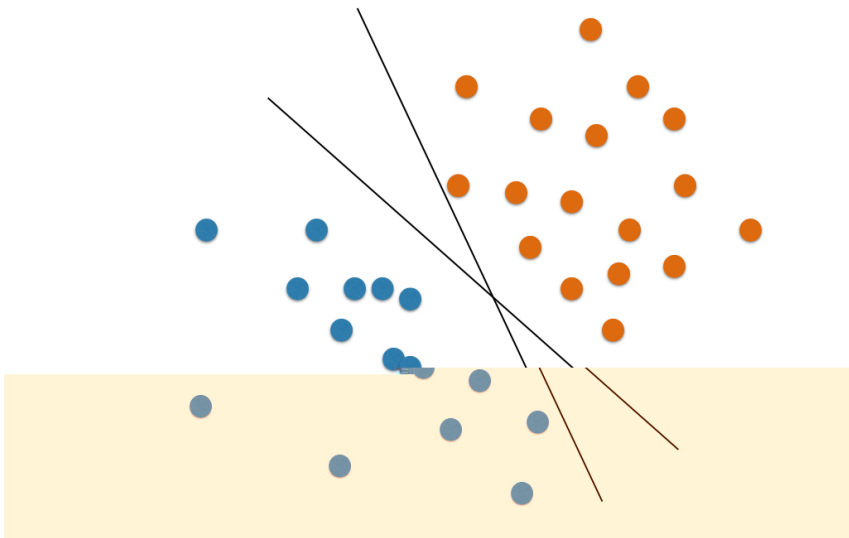
# Les classifieurs à vaste marge : cas de données linéairement séparables



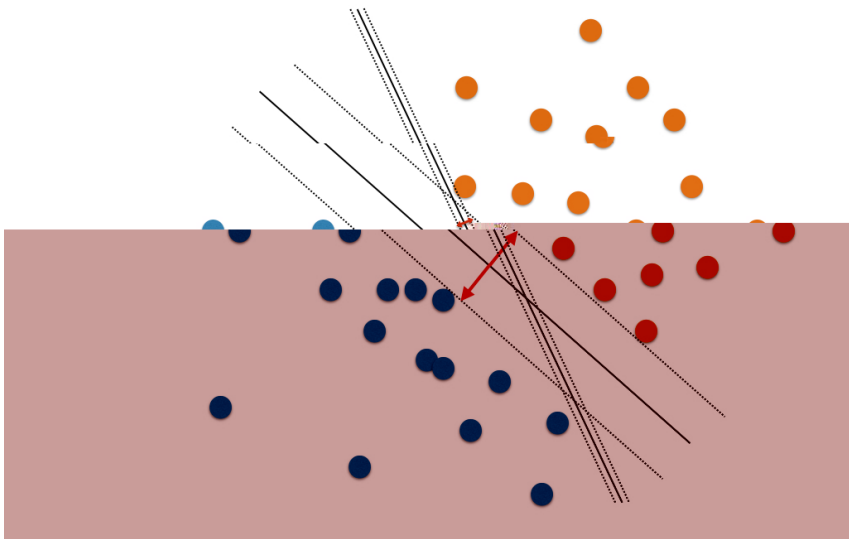
# Les classifieurs à vaste marge : cas de données linéairement séparables



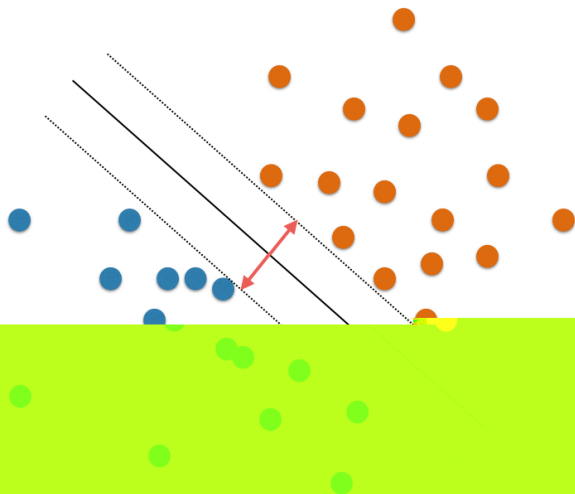
# Les classifieurs à vaste marge : cas de données linéairement séparables



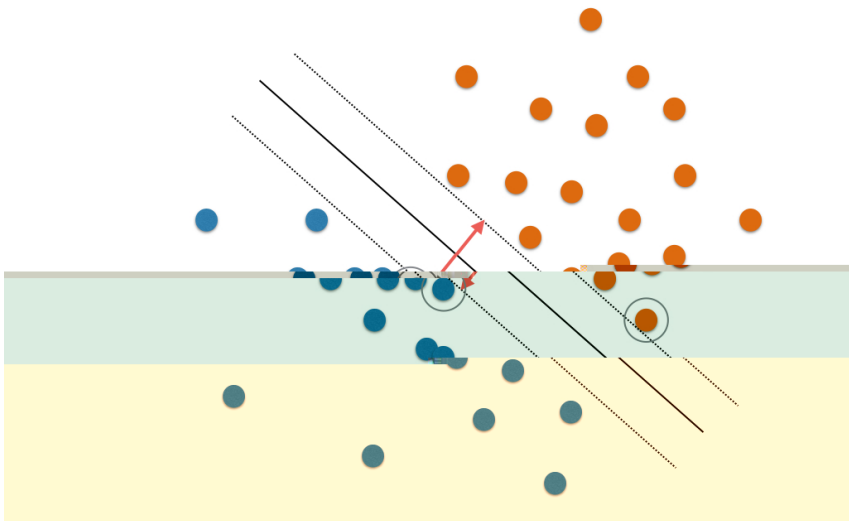
# Les classifieurs à vaste marge : cas de données linéairement séparables



# Les classifieurs à vaste marge : cas de données linéairement séparables

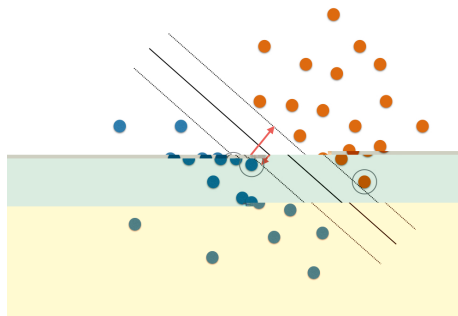


# Les classifieurs à vaste marge : cas de données linéairement séparables



# Les classifieurs à vaste marge : cas de données linéairement séparables

- Il existe une **infinité** de droites séparant les points.
- On cherche la **séparation linéaire** qui donne la plus grande **marge**.
- Les "bords" de cette marge s'appuient sur des **vecteurs supports**.
- On trouve cette séparation en résolvant un problème d'**optimisation convexe**.



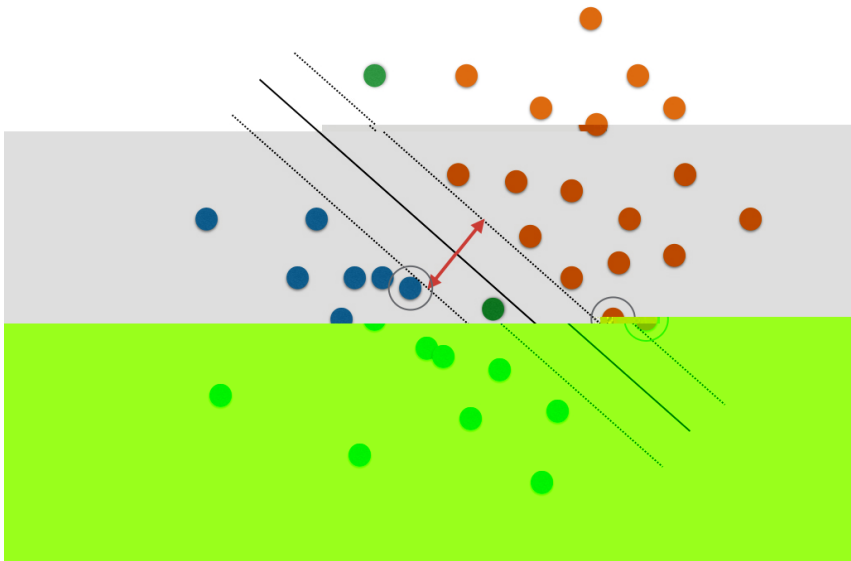
Les **vecteurs supports** sont les points les plus **ambigus** et donc les plus **difficiles à classifier**.

Ce sont ces points qui influencent le choix de la **meilleure droite** : si ces points changent, la droite change.

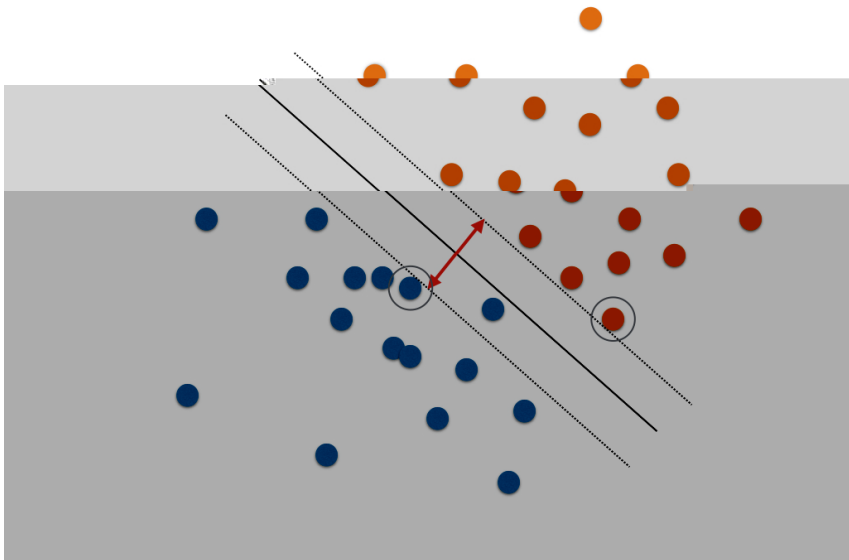
En quelque sorte, on se met dans un "worst-case" scenario : puisque l'on sait classifier les points les plus ambigus, le système devrait être **robuste**.

**Plus la marge est grande, plus l'on est sûrs de nous.**

# Phase de test



# Phase de test



Les données sont rarement aussi simples !

## Cas de données non linéairement séparables

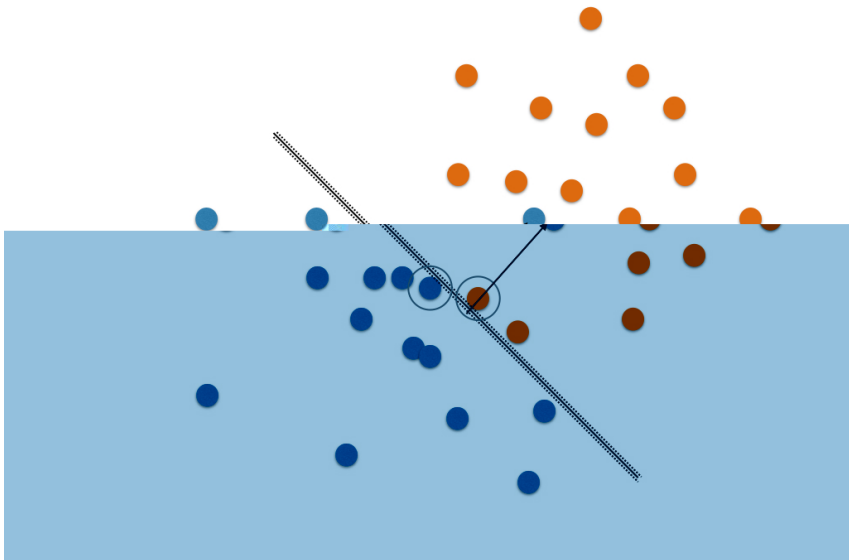


## Cas de données non linéairement séparables

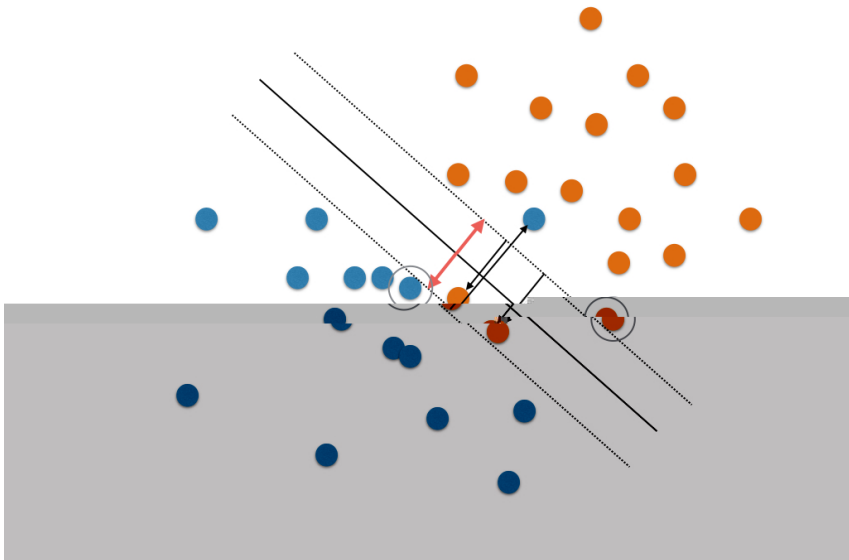
On **autorise** des erreurs de classification sur l'**ensemble d'entraînement**. Il sera moins sensible aux **outliers** et donc plus robuste.

Mais jusqu'à quel point autoriser ces erreurs ?

# Cas de données non linéairement séparables

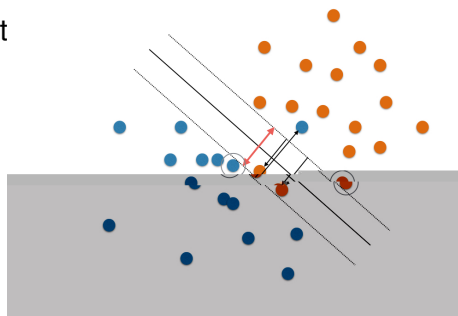


# Cas de données non linéairement séparables



# Cas de données non linéairement séparables

- C'est un **paramètre de régularisation** qui décide à quel point on peut faire des erreurs. Plus il est grand, plus les erreurs sont autorisées et plus la marge est large (ou l'inverse selon les définitions).
- On préfère des erreurs sur l'ensemble d'entraînement plutôt que du **sur-apprentissage**.
- Il faut trouver le bon **compromis** entre peu d'erreurs et une marge importante.
- Le paramètre de régularisation est optimisé en **validation croisée**.

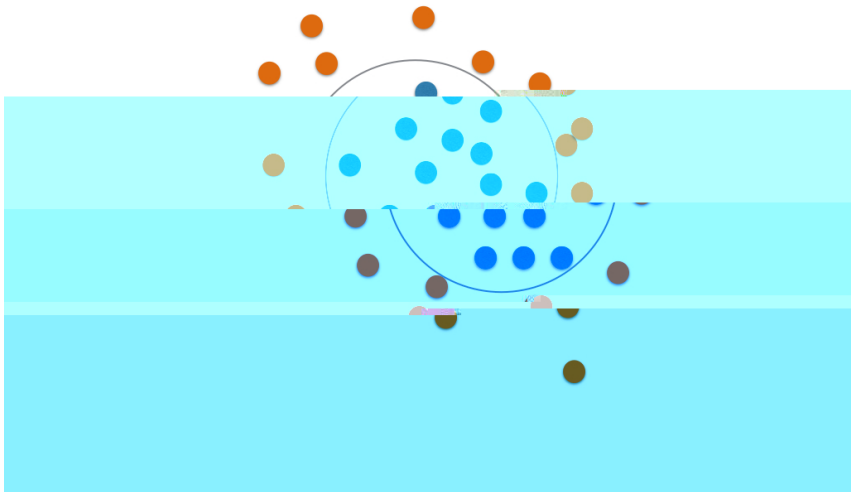


Démo

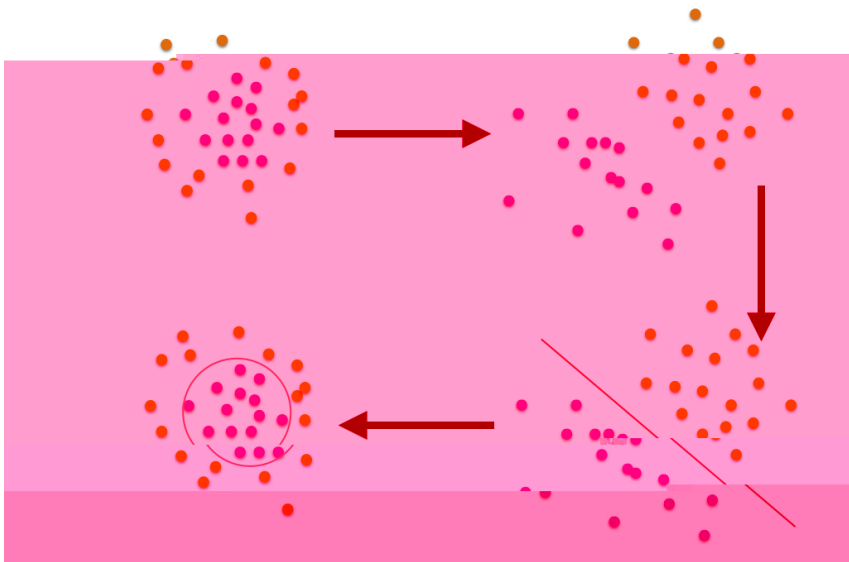
Et si le cas linéaire n'est pas adapté?



Et si le cas linéaire n'est pas adapté?



Et si le cas linéaire n'est pas adapté?



- L'idée est donc d'appliquer une **transformation** aux données pour se trouver dans un cas où le problème redevient linéaire.
- Cette transformation passe par ce qu'on appelle un **noyau**.
- Un noyau est une mesure de **similarité** souvent notée  $K$  (kernel) qui mesure à quel point deux vecteurs sont proches.
- En réalité, on n'applique pas la transformation à chaque vecteur directement mais à leur similarité deux à deux.
- Il existe différents types de noyaux. Il faut choisir celui qui est le plus adapté aux données.

# Quelques noyaux

- Le noyau **linéaire**

$$K(x, y) = x_1 y_1 + x_2 y_2 + \dots + x_n y_n$$

- Le noyau **RBF**

$$K(x, y) = \exp(-\gamma((x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2))$$

- Le noyau **polynomial**

$$K(x, y) = (x_1 y_1 + x_2 y_2 + \dots + x_n y_n + c)^q$$

- Et surtout : le noyau customisé ! Sans même savoir décrire les objets, il suffit de savoir décrire leur similarité par un noyau pour faire fonctionner un SVM.

Les noyaux **déforment** donc les distances entre les points et changent par conséquent la forme du problème. Nous n'irons pas plus loin dans l'explication car l'intuition s'arrête là!

Démo

# Conclusion sur les SVMs

## Avantages :

- Très performants.
- Ne nécessitent que la similarité des objets en entrée et non les objets en eux-mêmes. On peut donc traiter images, séquences biologiques, vidéos... Il n'y a pas de limite.
- Très efficaces en grande dimension.

## Inconvénients :

- Il faut pouvoir fournir un noyau intelligent, sans quoi le modèle échoue.
- Difficiles à interpréter
- Dépendent souvent de paramètres à optimiser.
- Parfois long !

## En Python :

```
from sklearn.svm import SVC
model = SVC(C=1, kernel = 'linear')
model.fit(Xtrain, Ytrain)
predictions = model.predict(Xtest)
```

- 1 Support Vector Machines (SVM)
- 2 Réseaux de neurones

- Premières idées dans les années 50 : basé sur le fait de "copier" le fonctionnement des neurones du cerveau.
- Modèle courant aujourd'hui : **perceptron multi-couches** introduit en 1986.
- **Théorème (Cybenko, 1989)** : Toute fonction continue bornée est estimable, avec une précision arbitraire, par un réseau à deux couches.
- Aujourd'hui : extension au **deep learning**.
- **Black box** par excellence : il est impossible de les interpréter.

# Principe

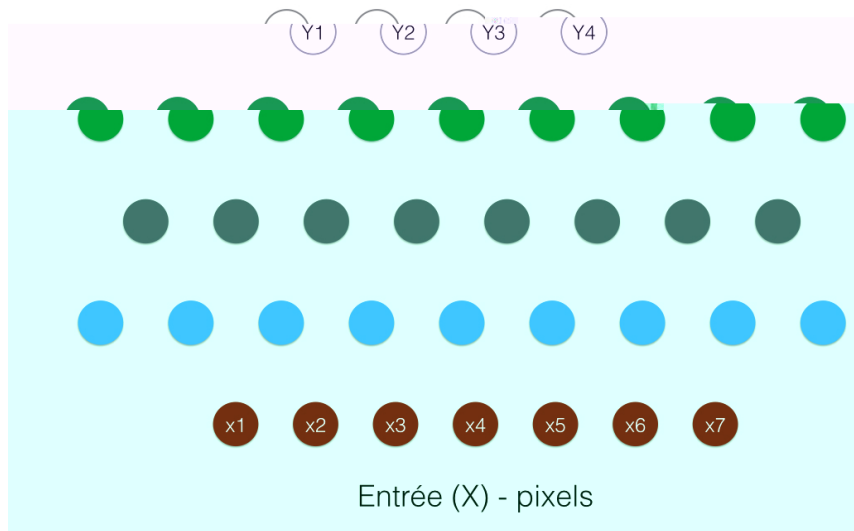
Sortie (Y) - type d'image



Entrée (X) - pixels

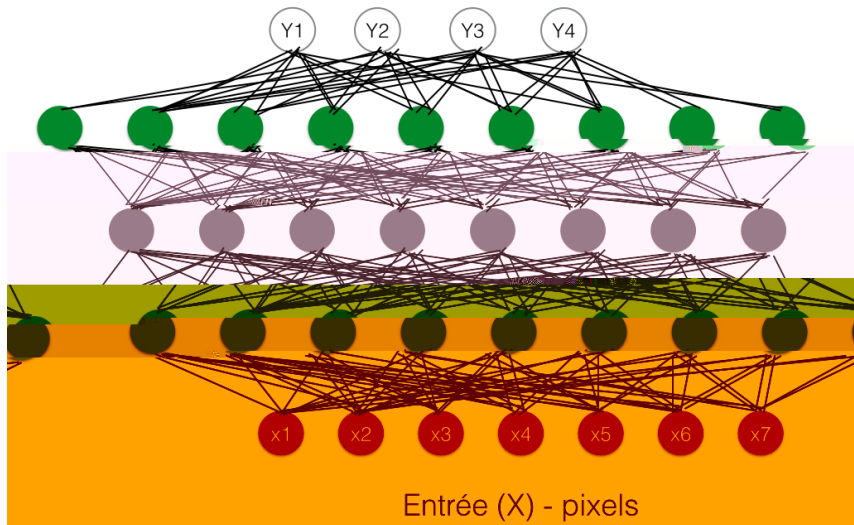
# Principe

Sortie (Y) - type d'image



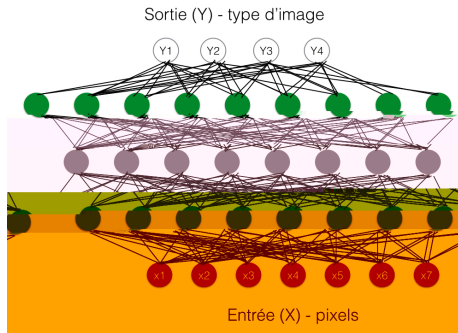
# Principe

Sortie (Y) - type d'image



# Principe

- Entre la **couche d'entrée** et la **couche de sortie** on trouve des **couches cachées**.
- A partir de la première couche cachée, chaque neurone est une **fonction des neurones précédents**.
- On cumule ainsi les transformations jusqu'à la couche de sortie. La réponse est donc une **aggrégation de fonctions** de fonctions (de fonctions de fonctions de fonctions) de l'entrée.
- Le vocabulaire du cerveau est utilisé ici : neurones, connexions, activation.



## Plus précisément... Comment ça marche?

- Chaque fonction interne est **paramétrée**, c'est-à-dire qu'elle dépend, par exemple, d'un paramètre par neurone.
- Il faut **optimiser** ces paramètres, i.e. trouver ceux qui expliquent le mieux le passage de l'entrée à la sortie.
- On utilise des techniques d'optimisation type descente de gradient, gradient stochastique, etc. pour faire ce qu'on appelle la **rétro-propagation** (backpropagation) : pour chaque neurone du réseau et pour chaque exemple d'entrée, on utilise la valeur de la sortie et on apporte cette information, étage par étage, à toutes les fonctions.
- Autrement dit, on utilise la réponse (Y) pour **corriger** les paramètres. Plus on a d'observations, plus le réseau prédit bien la sortie.

# Exemple

- Je reçois une **nouvelle** image, je **prédis** grâce à mon réseau qu'il s'agit d'un chat.
- Une fois au niveau de la **sortie**, je m'aperçois que c'est en fait un camion.
- Je **rétro-propage** cette information pour **updater** la valeur de mes (millions de) paramètres.
- Plus le réseau voit d'exemples labélisés, plus il emmagasine de connaissances et devient "bon".

# Lien avec l'apprentissage par renforcement

L'**apprentissage par renforcement** est un ensemble de techniques qui permettent d'apprendre **au cours du temps** grâce à un principe de **récompenses**.

Au départ, l'**agent** ne sait rien. Il doit **explorer**, les résultats sont mauvais. A la fin de chaque tour d'entraînement il reçoit une récompense s'il a réussi. Exemple : DeepMind et les jeux vidéos.

Pour faire évoluer l'agent, on utilise le principe de **rétro-propagation**. Beaucoup de ces méthodes utilisent d'ailleurs des réseaux de neurones lors de l'apprentissage.

(Démonstration)

# Les paramètres globaux

Il faut choisir un certain nombre de choses pour son réseau:

- La complexité:
  - Le nombre de **couches** : plus elles sont nombreuses, plus on pourra prédire des choses compliquées, plus le temps de calcul est long. Si elles sont trop nombreuses, on tombe dans le **sur-apprentissage**.
  - Le nombre de **neurone** : idem !
- Les **fonctions** entre les neurones : on en propose un certain nombre. Elles dépendent en partie du type de sortie (par exemple : classification binaire vs multi-classes)
- Le **learning rate** : le taux d'apprentissage. S'il est trop élevé, chaque exemple va influencer beaucoup sur le réseau et on peut tomber dans un optimum **local**. S'il est faible, le réseau va être très long à optimiser. On choisit souvent un paramètre dynamique : il baisse au fur et à mesure de l'apprentissage.

L'avantage des réseaux de neurones est que l'on n'a pas besoin de **pré-traiter** les entrées : c'est lui qui va créer les patterns/features intéressantes.

Dans le cas de la reconnaissance d'images, on peut ne lui donner en entrée que des pixels et non des résumés de l'image obtenus manuellement (handcrafted).

Démo

# L'apprentissage par batch et l'apprentissage online

Bien que l'on ait accès à de puissants serveurs de calculs, on peut vouloir limiter l'utilisation de la mémoire pour entraîner ces algorithmes très lourds.

On peut alors envisager de ne fournir les données au réseau que **par batches**. Ainsi, il ne traite qu'une seule partie des données à la fois, sans avoir besoin de stocker les autres. Il update les paramètres à la fin de chaque batch.

Afin que le réseau continue d'apprendre et s'améliore **au fur et à mesure**, on peut utiliser le **online learning** : le réseau reste en phase d'apprentissage "toute sa vie" et les paramètres sont updatés à chaque nouvel exemple.

# Conclusion sur les réseaux de neurones

## Avantages :

- Potentiellement extrêmement performants, en particulier dans certains domaines comme la reconnaissance d'images ou du langage.
- Flexibles sur la complexité.

## Inconvénients :

- Peu/pas de résultats théoriques pour expliquer la performance.
- Impossibles à interpréter.
- Calculs extrêmement lourds.
- Complexité difficile à calibrer : gros risques de **sur-apprentissage**.

## En Python :

- PyBrain
- neurolab