

Bureau d'études Apprentissage automatique (machine learning)

Shahram.Hosseini@irap.omp.eu

Université Paul Sabatier
M1 EEA-SIA2 et IdS-IM
2024-2025

Introduction

Apprentissage automatique (machine learning) : donner aux machines la capacité d'**apprendre** à partir de données.

« **Apprendre** » : améliorer ses performances (mesurées par un critère), à partir d'expériences, pour effectuer une tâche.

Principales tâches en machine learning:

- 1) **Régression**: estimer la valeur d'une variable continue à partir des valeurs d'autres variables. L'algorithme d'apprentissage doit générer une fonction $f: \mathbb{R}^n \rightarrow \mathbb{R}$. Exemples: prédiction, interpolation, débruitage.
- 2) **Classification**: déterminer à quelle classe une entrée appartient. L'algorithme d'apprentissage doit générer une fonction $f: \mathbb{R}^n \rightarrow \{1, \dots, k\}$. La sortie est une variable discrète. Exemples: Reconnaissance de chiffres.
 - **supervisée**: on dispose d'exemples labélisés, pour lesquels la classe d'appartenance est connue.
 - **non supervisée (clustering)**: on n'a pas d'exemples labélisés.

Base de données d'apprentissage : les données utilisées pour entraîner un modèle capable de réaliser la tâche de régression ou classification souhaitée.

Base de données de test : les données non-utilisées pour l'apprentissage du modèle, qu'on peut utiliser à la fin d'apprentissage pour tester le bon fonctionnement du modèle.

Capacité de généralisation d'un modèle: la capacité du modèle à bien fonctionner sur les données qui n'ont pas été utilisées pour son apprentissage = les performances du modèle sur la base de test.

Sous-apprentissage: quand le modèle ne fonctionne pas bien sur la base d'apprentissage = un modèle qui n'a pas bien appris les données d'apprentissage.

Sur-apprentissage: quand le modèle fonctionne bien sur les données d'apprentissage mais qui ne fonctionne pas bien sur les données de test = faible capacité de généralisation = le modèle a appris les données d'apprentissage par cœur.

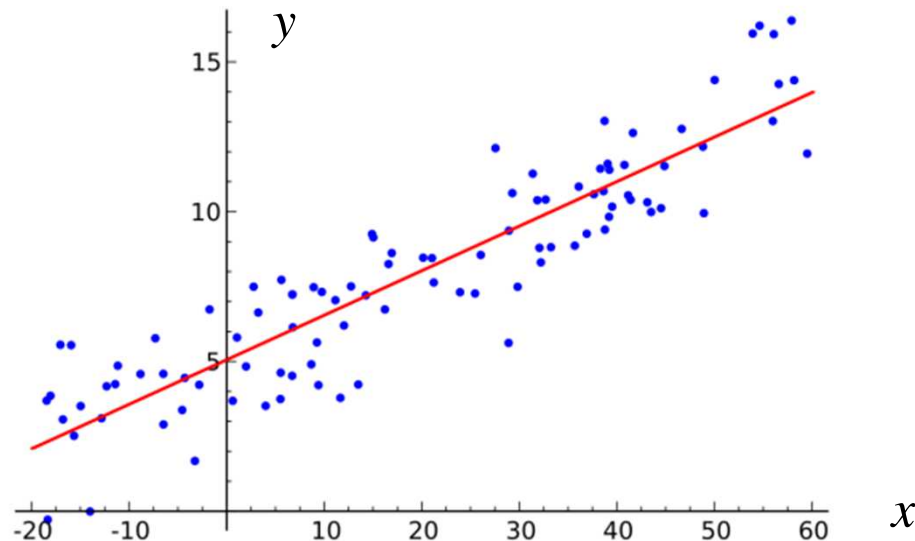
Chapitre 1: Rappel de quelques méthodes classiques d'apprentissage automatique

A. Régression linéaire

A.1. Régression linéaire simple

Modéliser la relation entre une variable explicative x et une variable expliquée y avec un modèle linéaire: $y = \beta_0 + \beta_1 x + \text{erreur du modèle}$:

A partir de plusieurs réalisations disponibles de ces 2 variables $(x(i), y(i))$ $i=1, \dots, n$ (on parle alors de n individus) qui forment un nuage de points, on veut estimer l'ordonnée à l'origine (β_0) et le coefficient directeur (β_1) de la droite de régression.

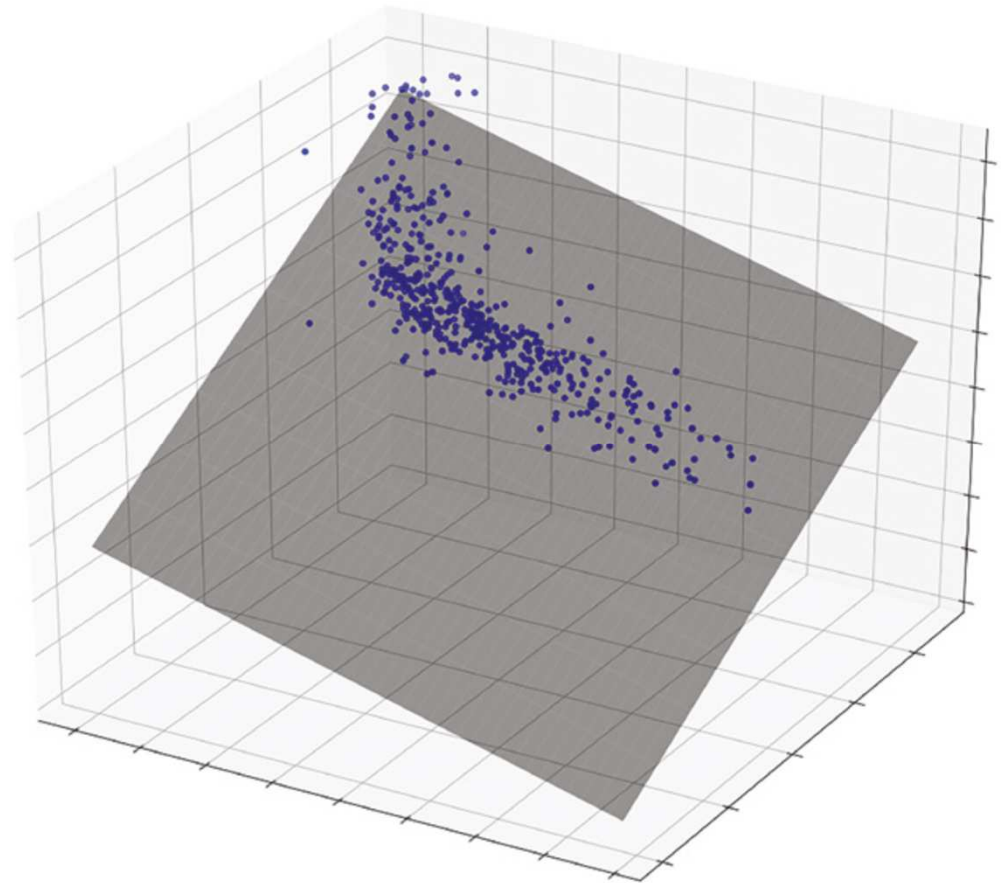


A.2. Régression linéaire multiple

Modéliser la relation entre plusieurs variables explicatives x_1, \dots, x_K et une variable expliquée y avec un modèle linéaire:

Pour l'individu i : $y(i) = \beta_0 + \beta_1 x_1(i) + \dots + \beta_K x_K(i) + \varepsilon(i)$ où $\varepsilon(i)$ représente l'erreur.

Exemple pour $K=2$



Modèle vectoriel

$$y(i) = \mathbf{x}(i)\boldsymbol{\beta} + \varepsilon(i) \quad i=1, \dots, n$$

$$\text{avec } \mathbf{x}(i) = [1, x_1(i), \dots, x_K(i)] \text{ et } \boldsymbol{\beta} = [\beta_0, \beta_1, \dots, \beta_K]^T$$

Modèle matriciel

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon} \quad \text{avec:}$$

$$\mathbf{y} = \begin{pmatrix} y(1) \\ y(2) \\ \vdots \\ y(n) \end{pmatrix}, \quad \mathbf{X} = \begin{pmatrix} 1 & x_1(1) & \cdots & x_K(1) \\ 1 & x_1(2) & \cdots & x_K(2) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1(n) & \cdots & x_K(n) \end{pmatrix}, \quad \boldsymbol{\varepsilon} = \begin{pmatrix} \varepsilon(1) \\ \varepsilon(2) \\ \vdots \\ \varepsilon(n) \end{pmatrix}$$

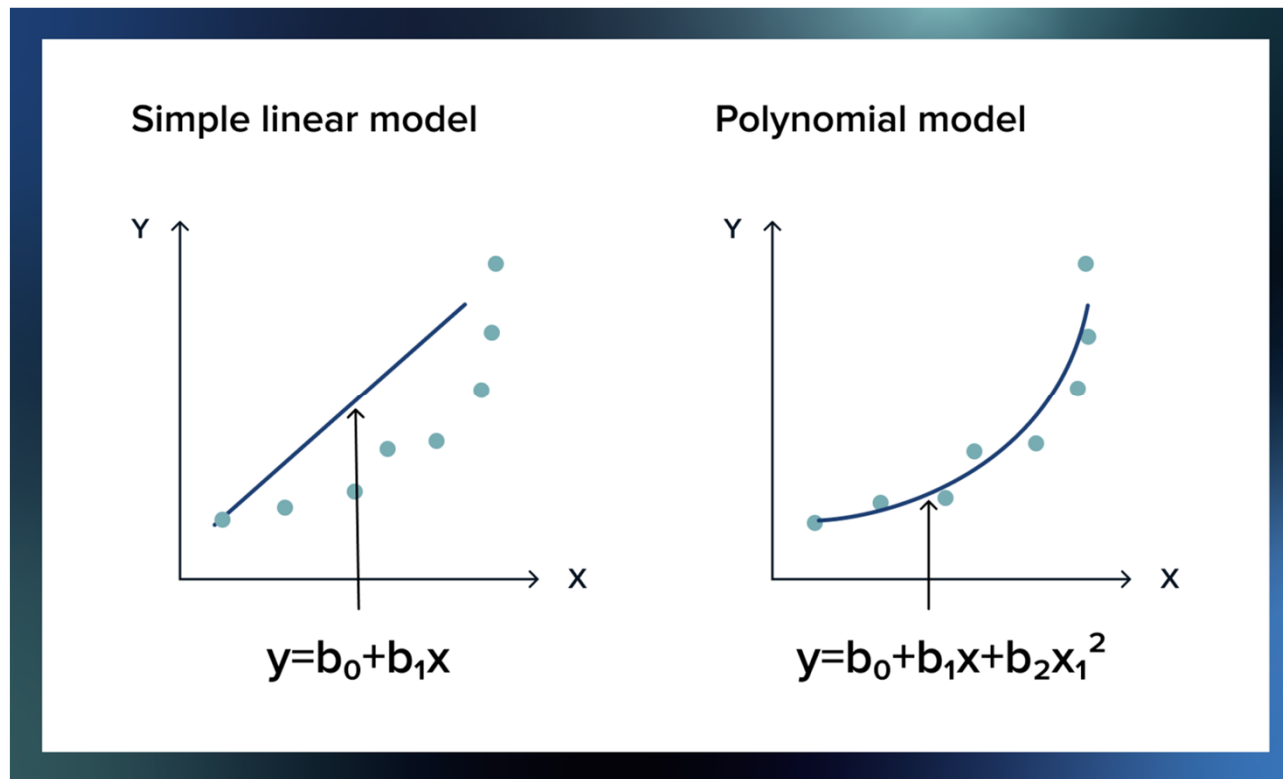
Estimateur des moindres carrés

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

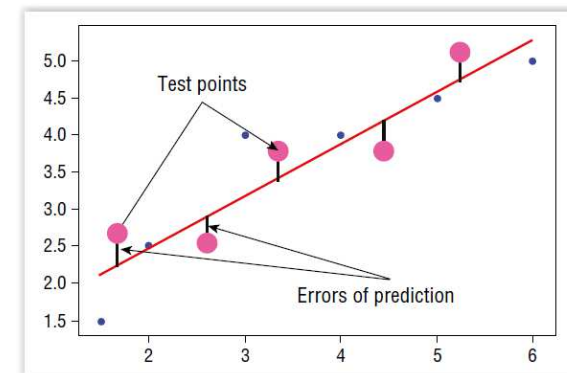
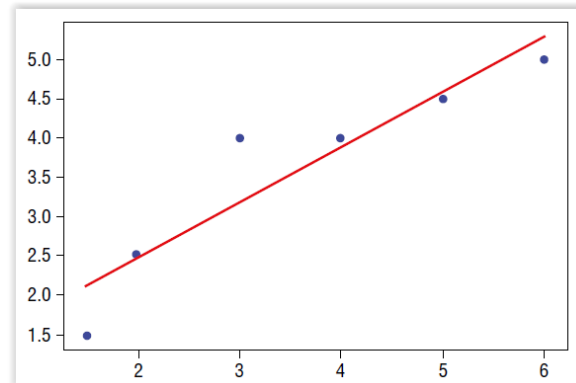
B. Régression polynomiale

B.1. Régression polynomiale simple

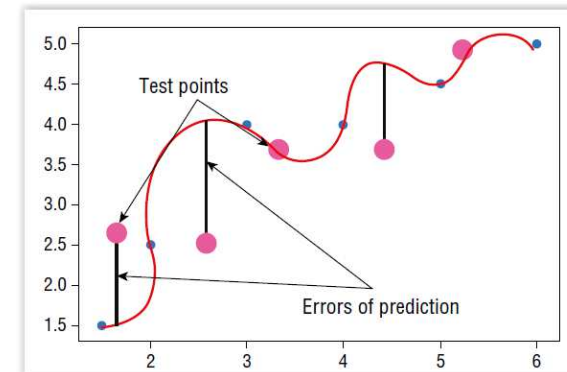
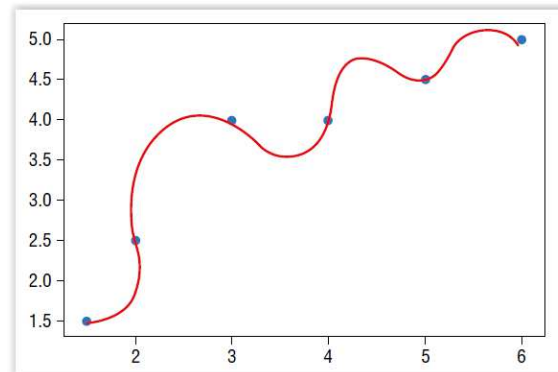
$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \dots + \beta_K x^K$$



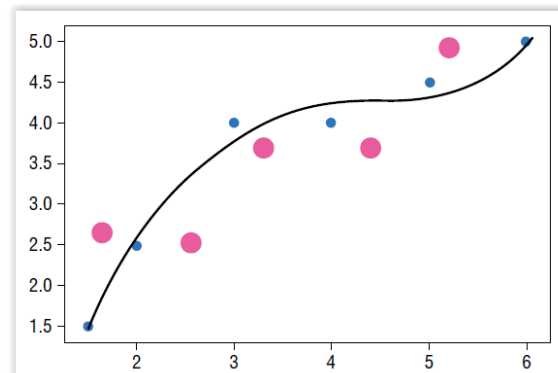
K trop petit →
Sous-apprentissage
(underfitting):
Erreur élevée sur la
base d'apprentissage



K trop grand →
Sur-apprentissage
(overfitting):
Erreur faible sur la
base d'apprentissage,
mais élevée sur la base
de test



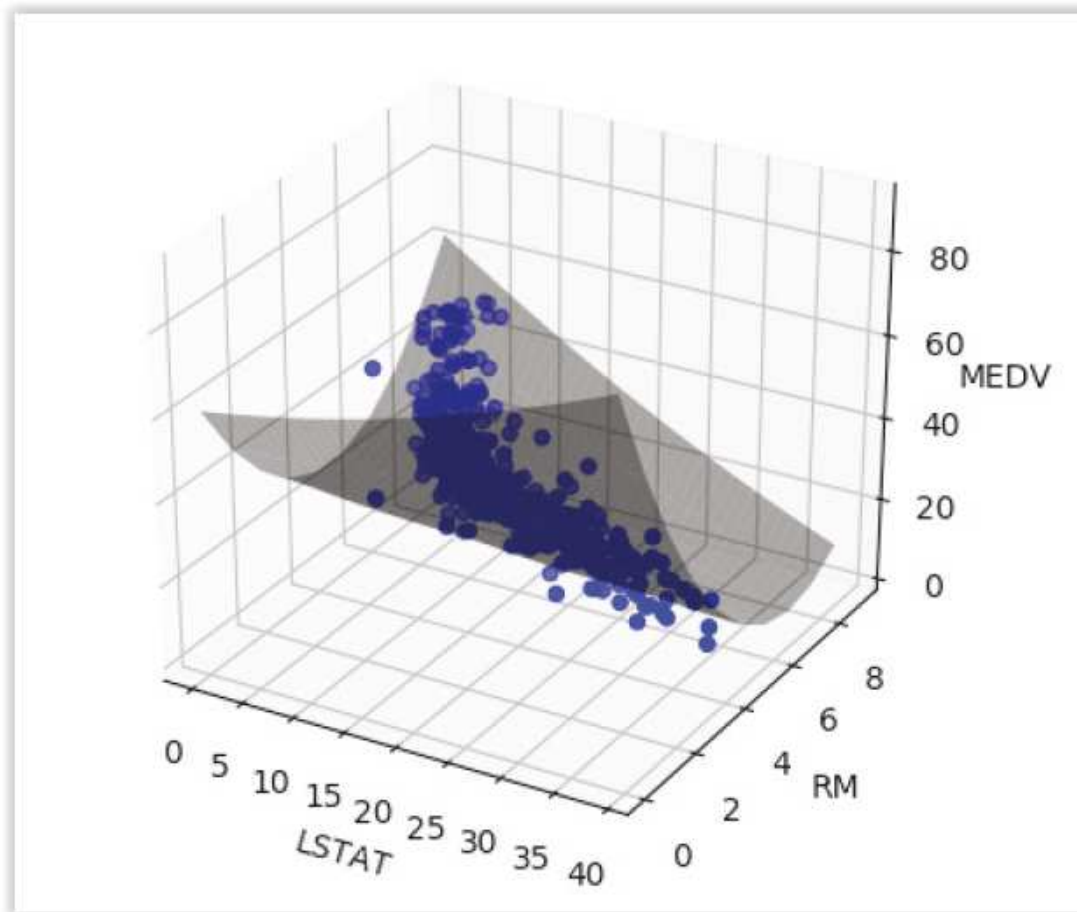
Bon choix de K



B.2. Régression polynomiale multiple

Exemple avec 2 variables et un modèle d'ordre K=2:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1^2 + \beta_4 x_1 x_2 + \beta_5 x_2^2$$

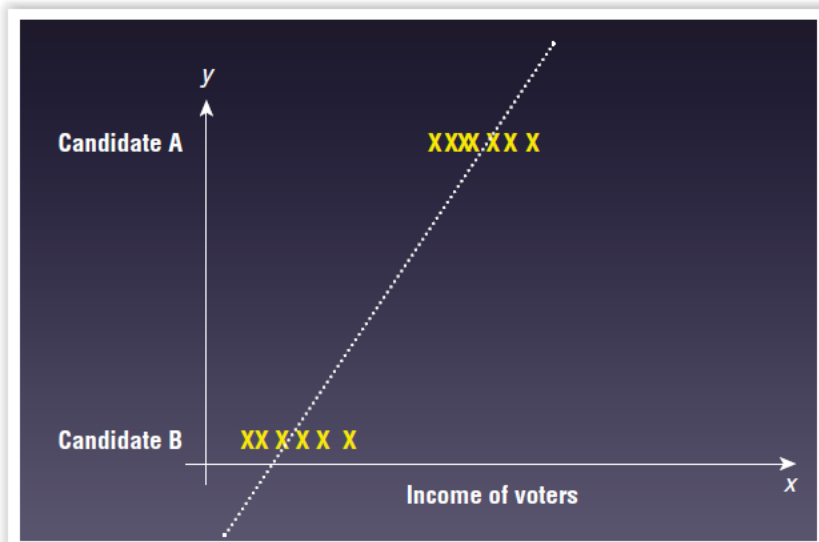


C. Régression logistique

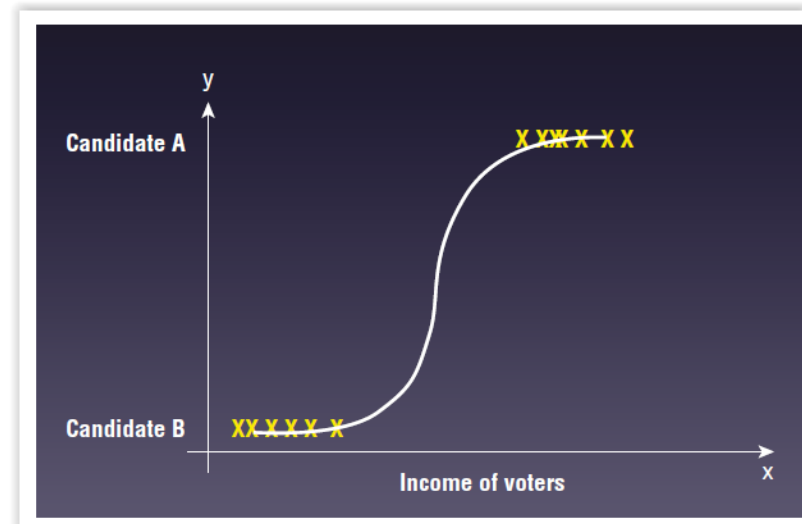
Contrairement à son nom, cette méthode est souvent utilisée pour la classification supervisée.

Un exemple: les intentions de votes pour 2 candidats en fonction des salaires des votants

Ce qu'on obtient avec régression linéaire



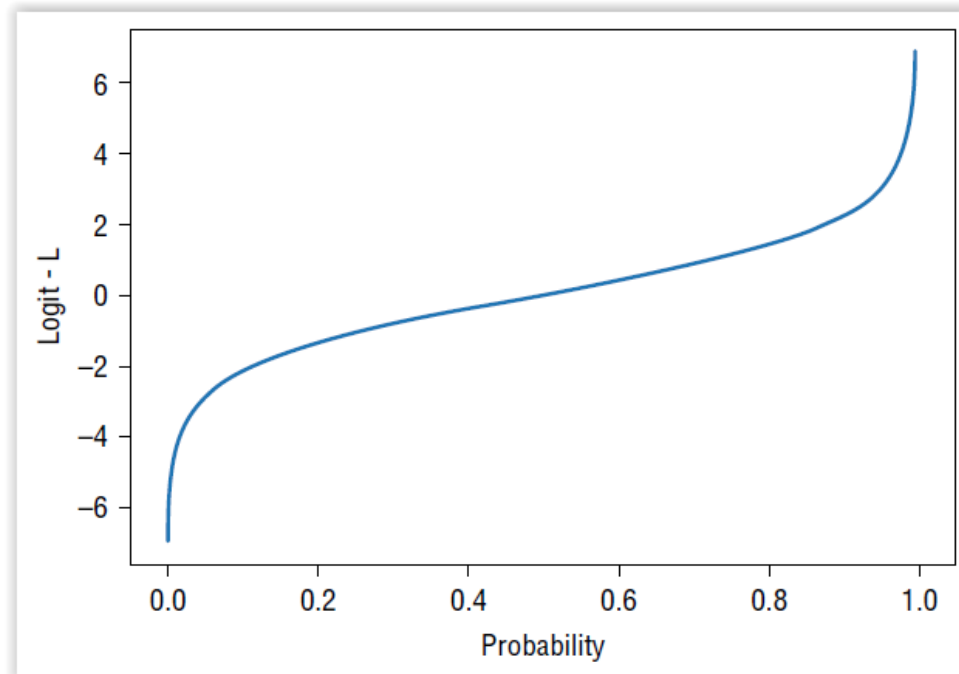
Ce qu'on souhaite obtenir



Ce qu'on souhaite obtenir: une valeur entre 0 et 1 en sortie, représentant une probabilité. Ensuite, si la valeur est < 0.5 , on peut supposer un vote pour le candidat B, et si elle est > 0.5 un vote pour le candidat A.

Fonction logit: $L = \ln(P/(1-P))$ où P représente la probabilité de succès, et $1-P$ est la probabilité d'échec.

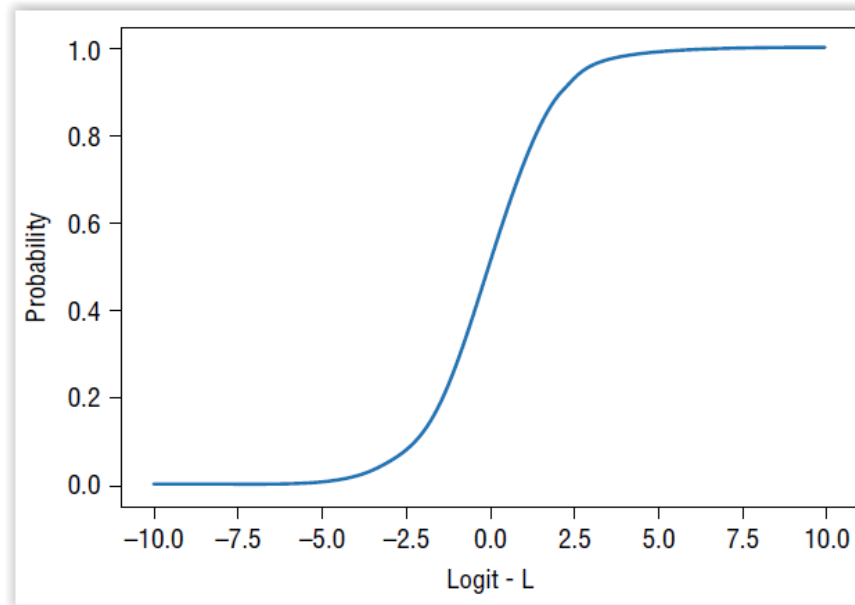
Cette fonction transforme une variable définie sur $[0, 1]$ en une variable définie sur $[-\infty, \infty]$



Ce qu'on souhaite obtenir avec la régression logistique: une valeur entre 0 et 1 en sortie, représentant une probabilité.

On cherche une fonction qui projette les valeurs réelles sur $[0, 1]$: c'est l'inverse de la fonction logit, appelé *sigmoïde*.

Fonction sigmoïde: $P=1/(1+\exp(-L))$



En remplaçant L par une combinaison linéaire des variables explicatives, on obtient la formule d'une régression logistique

$$P = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 x_1 + \dots + \beta_K x_K))}$$

Les coefficients (β_i) sont estimés avec la méthode du maximum de vraisemblance et un algorithme itératif.

La régression logistique peut être utilisée comme un classifieur binaire.

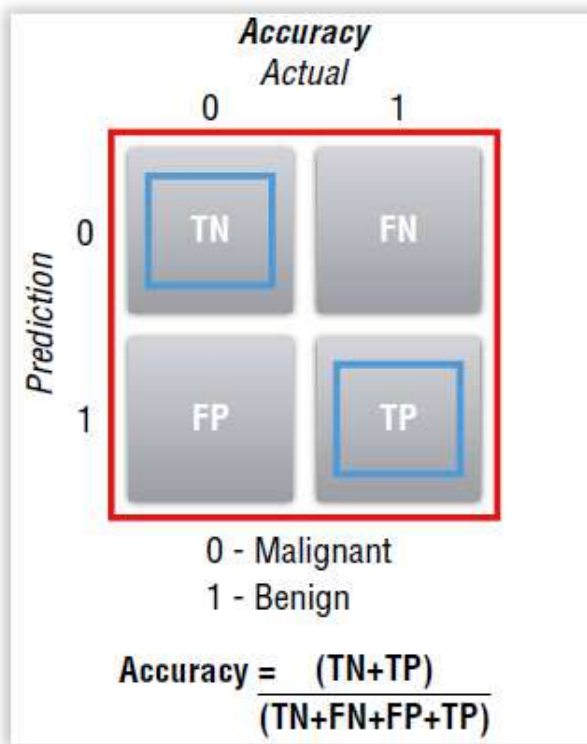
Evaluation des performances d'un classifieur binaire

Après la classification d'une donnée, il peut y avoir 4 cas possibles:

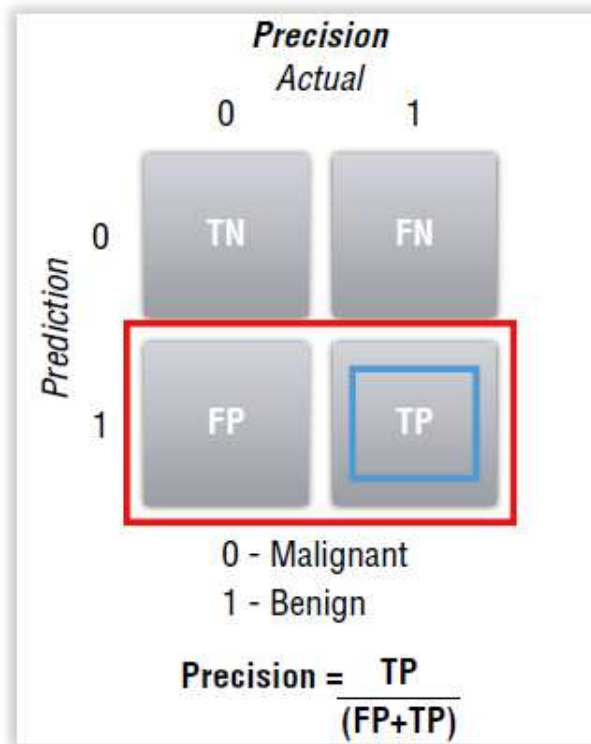
- True Positive (TP): Donnée correctement classée comme positive
- True Negative (TN): Donnée correctement classée comme négative
- False Positive (FP): Donnée faussement classée comme positive
- False Negative (FN): Donnée faussement classée comme négative

Matrice de confusion:

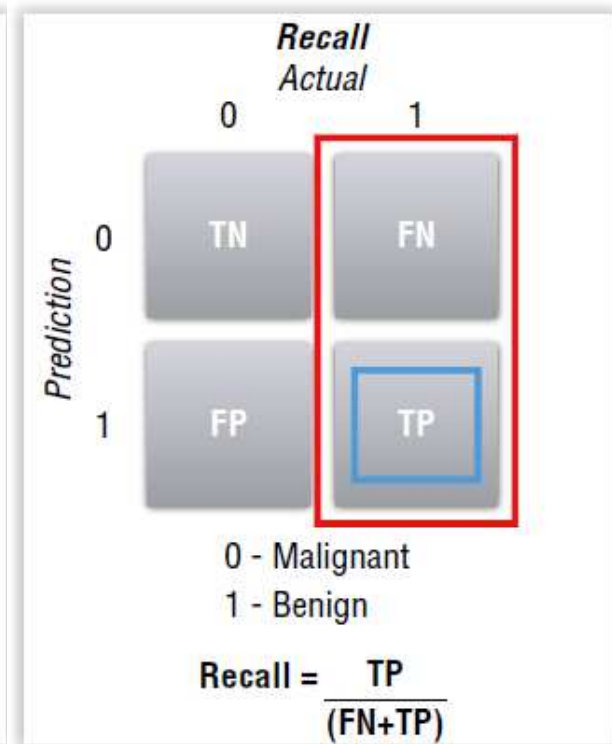
Confusion matrix		Reality	
		Negative : 0	Positive : 1
Prediction	Negative : 0	True Negative : TN	False Negative : FN
	Positive : 1	False Positive : FP	True Positive : TP



Accuracy (*Exactitude*):
quel pourcentage des
« données » est correctement
classé par le classifieur?

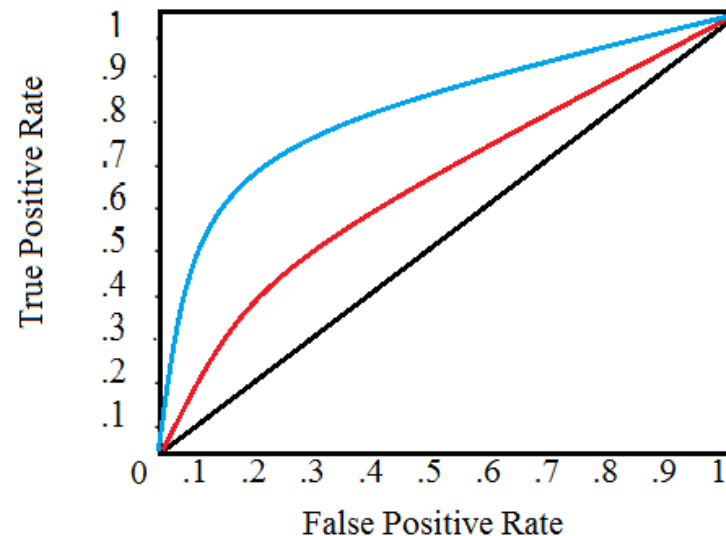


Precision (*Précision*):
quel pourcentage des
« données classées positives
par le classifieur » est
vraiment positif?



**Recall [Sensitivity or
True Positive Rate :
TPR] (*Rappel*):**
quel pourcentage des
« données réellement
positives » est correctement
classé par le classifieur ?

- **F1 score:** $(2 * \text{precision} * \text{recall}) / (\text{precision} + \text{recall})$
moyenne harmonique de precision et recall
- **False Positive Rate (FPR):** $\text{FP} / (\text{FP} + \text{TN})$
- **Specificity [true negative rate (TNR)]:** $\text{TN} / (\text{FP} + \text{TN}) = 1 - \text{FPR}$
- **Receiver Operating Characteristic (ROC):** courbe de TPR vs FPR pour différents seuils de classification

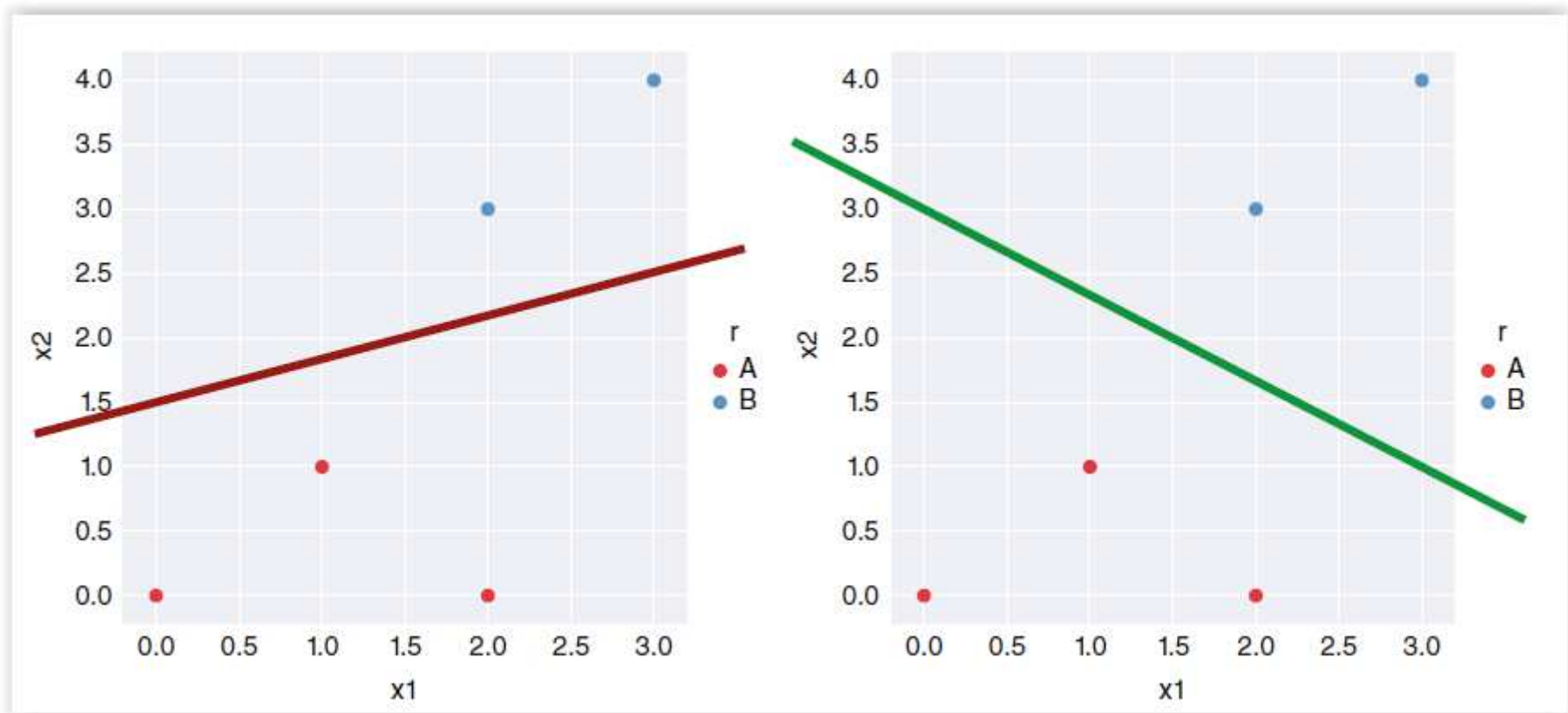


- **Area Under the Curve (AUC):** la surface sous la courbe ROC

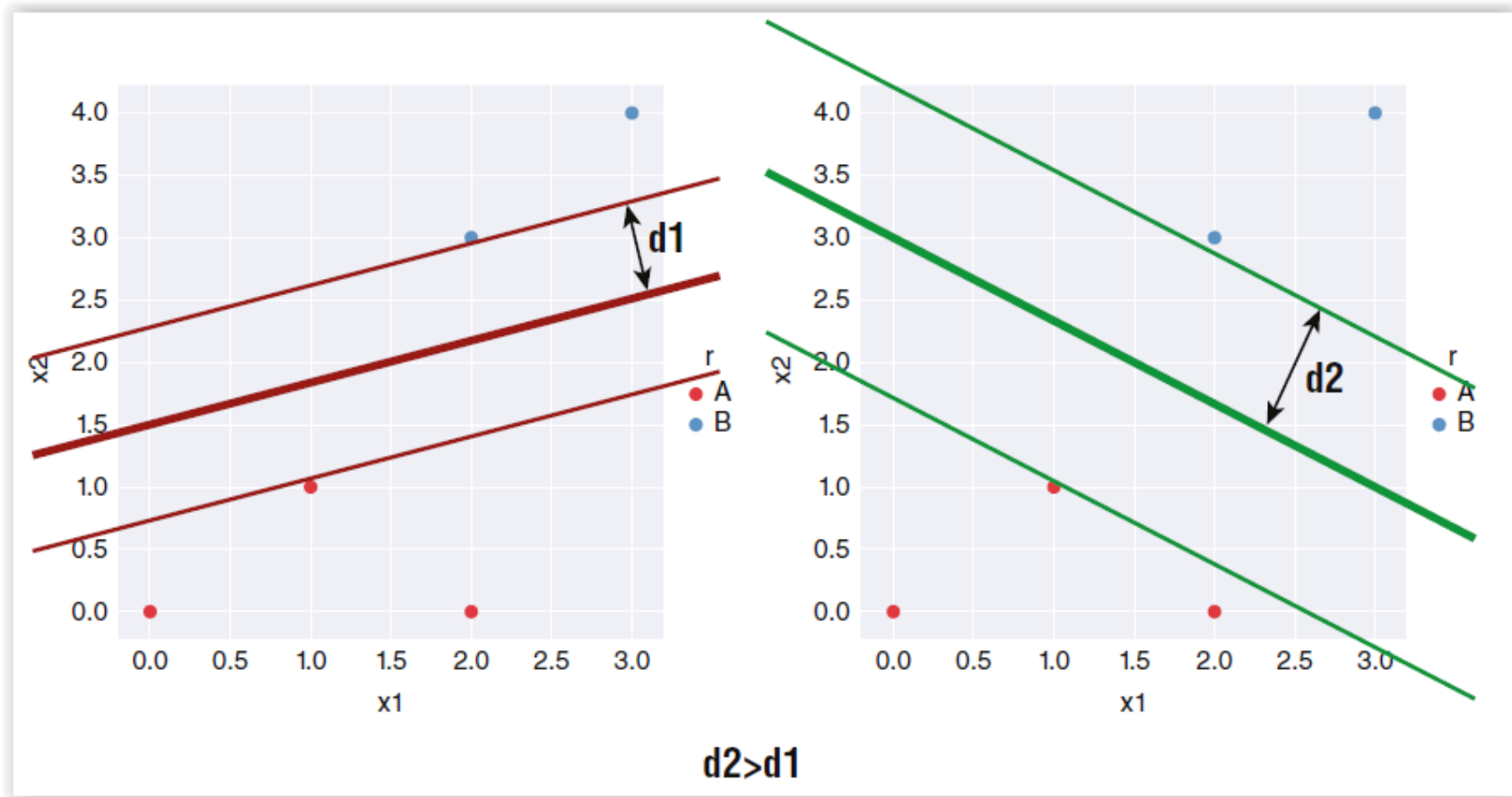
D. Support Vector Machine [SVM] (machine à vecteurs de support)

D.1. Cas des données linéairement séparables

Voici 2 manières différentes pour séparer les données en 2 classes:

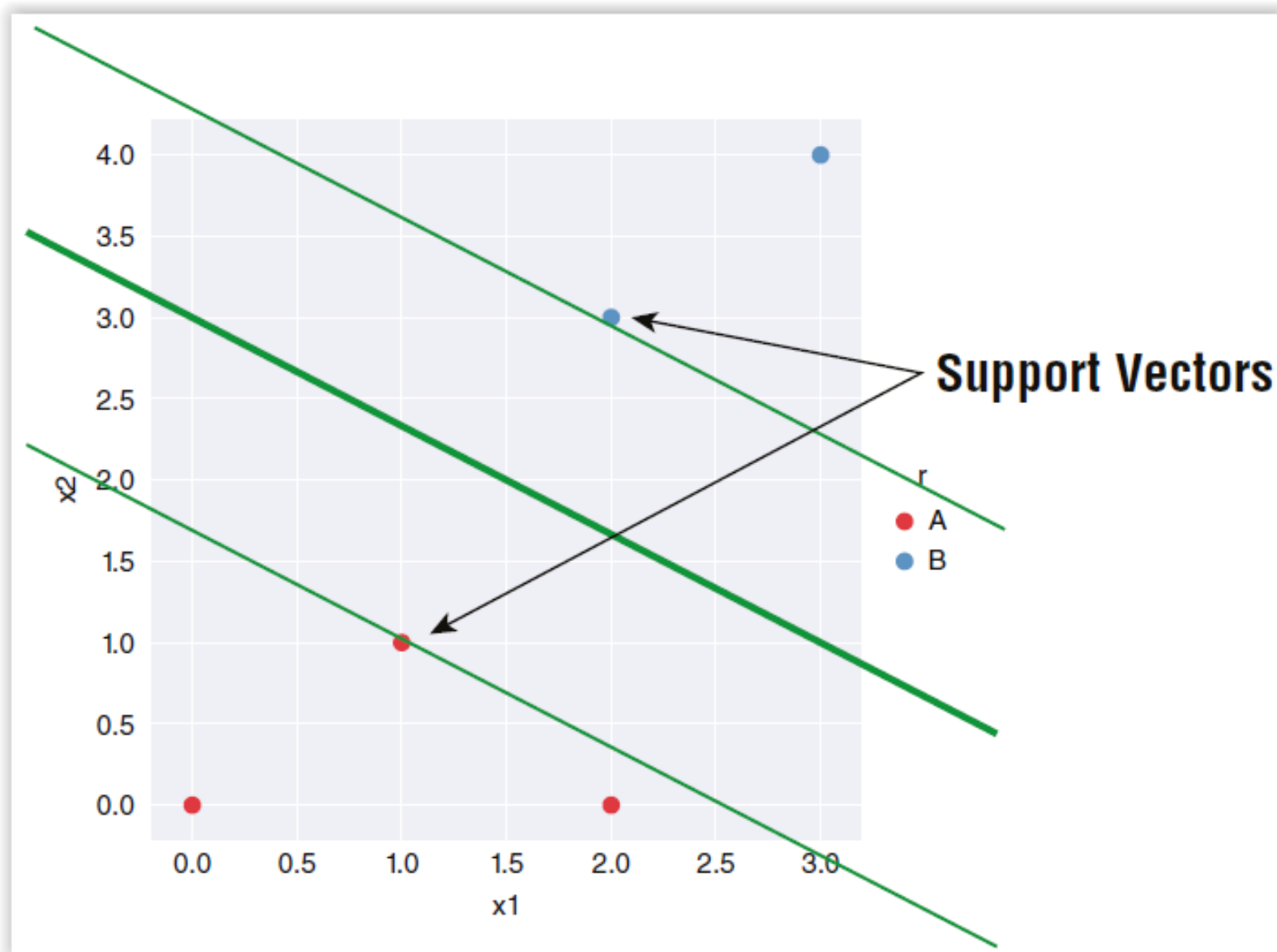


SVM cherche à séparer les 2 classes avec une ligne séparatrice (ou un hyperplan dans un espace multidimensionnel) qui assure la marge maximale ($d_2 > d_1$)

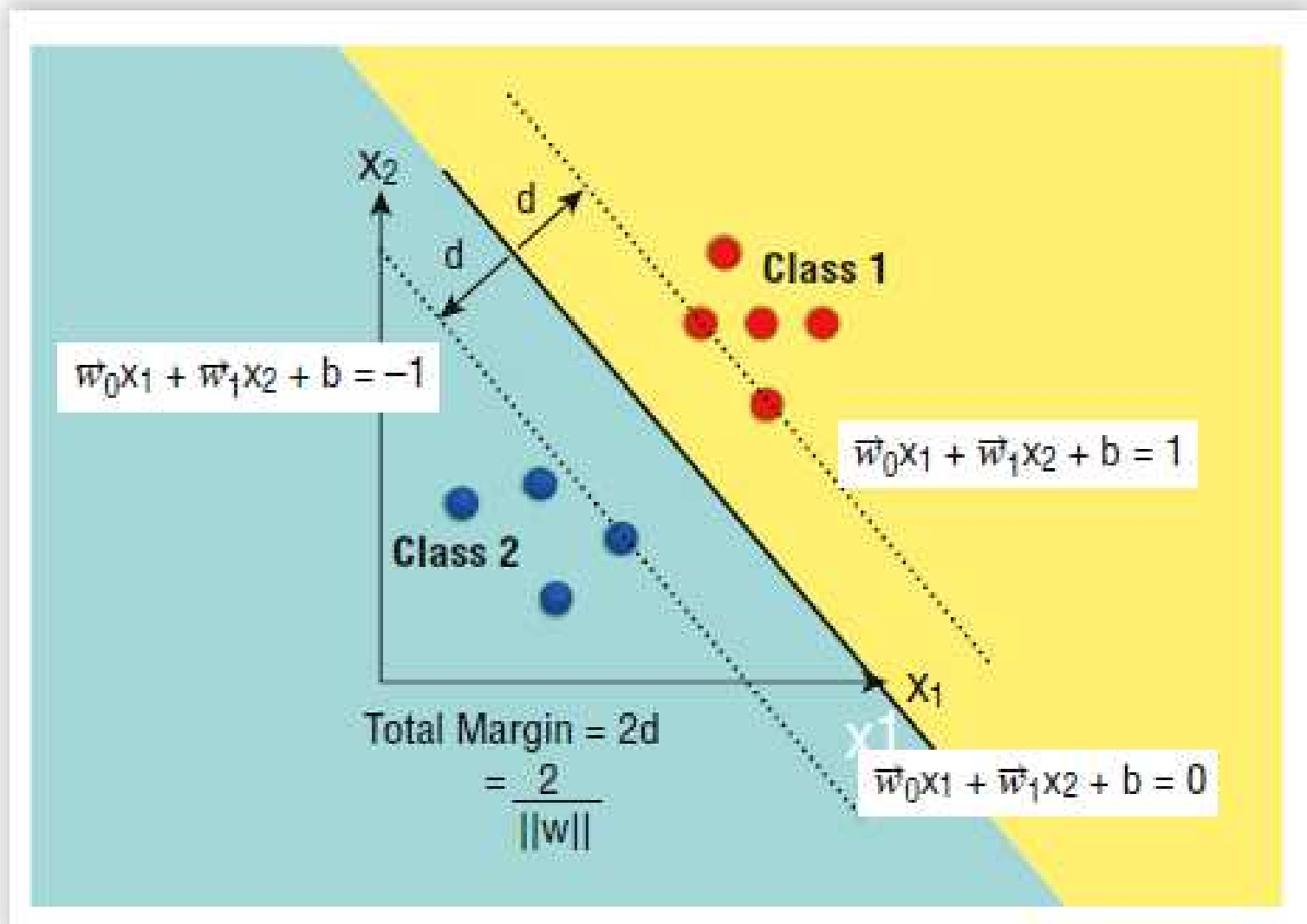


Chacune des 2 lignes marginales touche le point le plus proche d'un groupe de points. Le centre des 2 lignes marginales est la ligne séparatrice.

Les vecteurs support sont des points qui sont sur les 2 lignes marginales.

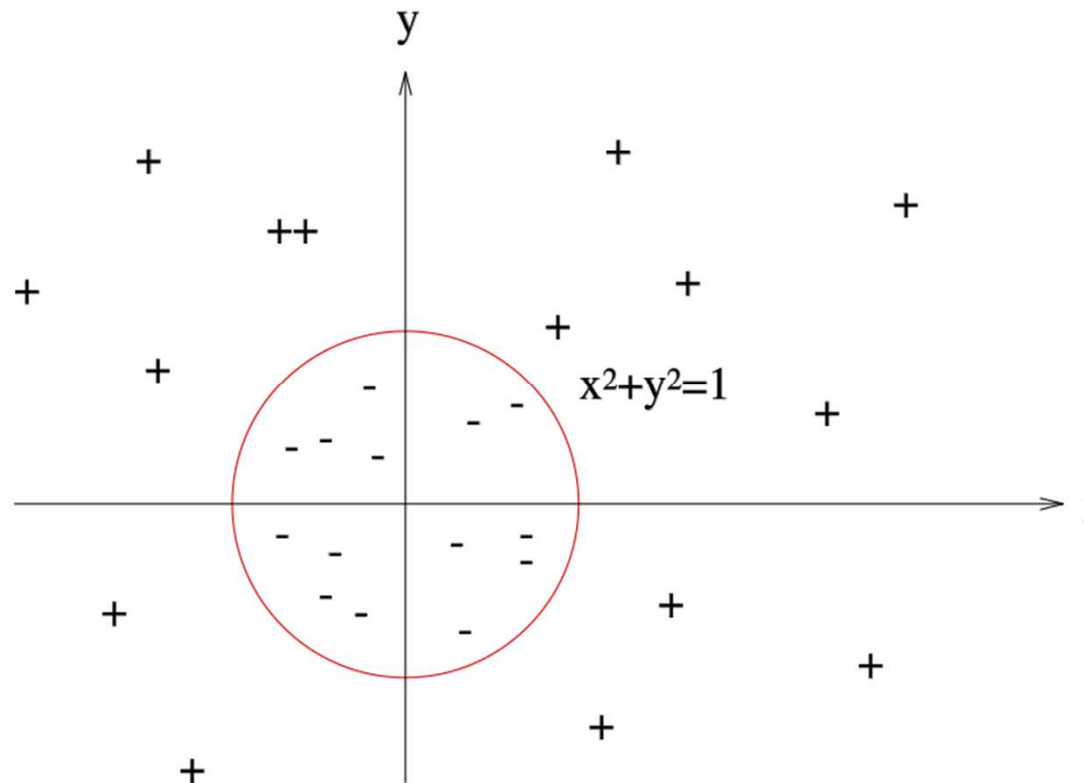


Formule pour l'hyperplan



A partir des données labélisées, on utilise un algorithme d'optimisation sous contrainte, pour déterminer les valeurs optimales des paramètres w_i et b .

D.2. Cas des données linéairement non-séparables



Ici, le problème n'est pas linéairement séparable en coordonnées cartésiennes, par contre en coordonnées polaires, le problème devient linéaire.

Astuce du noyau: on applique aux vecteurs d'entrée \mathbf{x} une transformation non-linéaire ϕ .

L'espace d'arrivée $\phi(\mathbf{x})$ est appelé *espace de re-description*.

Dans cet espace, on cherche alors l'hyperplan séparateur $\mathbf{w}^T \cdot \phi(\mathbf{x}) + b$

E. K-Nearest Neighbors [KNN] (K plus proches voisins - KPPV)

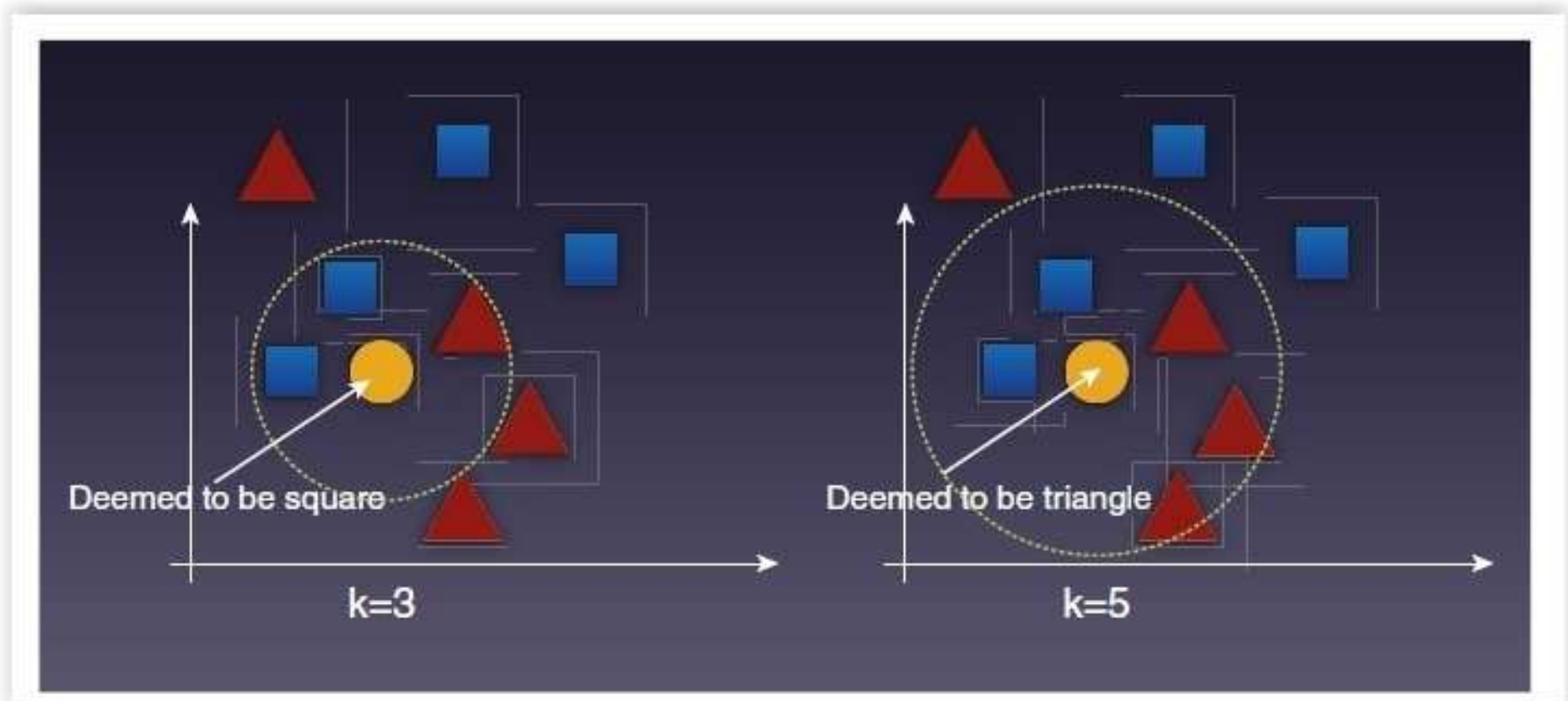
On dispose de n données labélisées (entrée-sortie). Pour estimer la sortie correspondant à une nouvelle entrée (non-labélisée) x , la méthode KNN prend en compte les K données labélisées dont l'entrée est la plus proche de la nouvelle entrée x , selon une distance à définir.

On utilise souvent la distance Euclidienne.

- En classification, on retient la classe la plus représentée parmi les classes des K plus proches voisins de x . Si $K=1$, la nouvel objet est affecté à la classe de son plus proche voisin.
- En régression, la valeur en sortie du nouvel objet est la moyenne des sorties de ses K plus proches voisins.

On peut pondérer l'influence contributive des voisins, par exemple avec une pondération $1/d_i$ où d_i représente la distance entre le nouvel objet et son voisin numéro i .

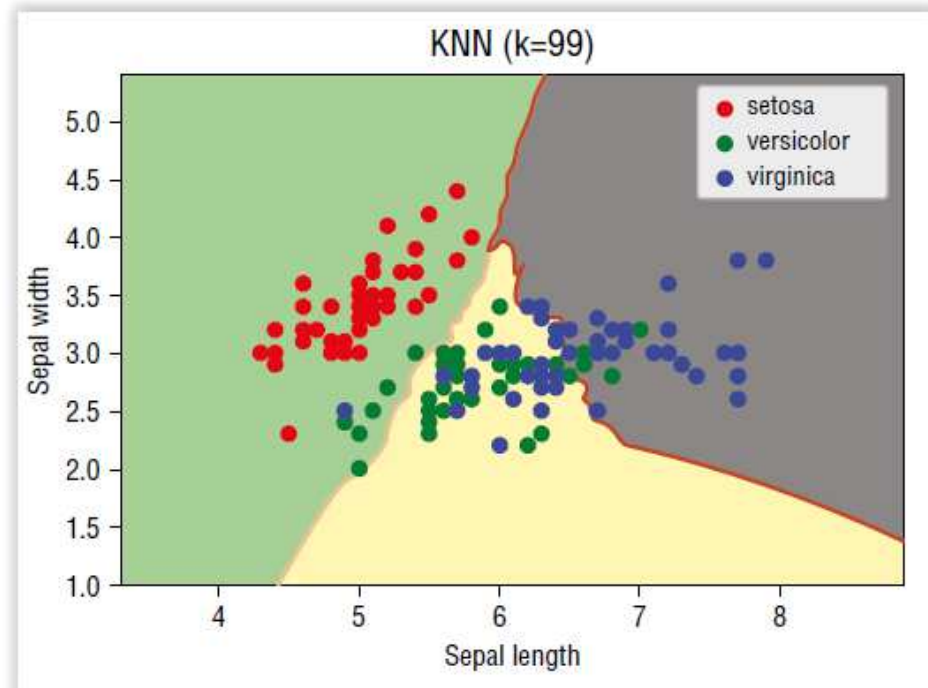
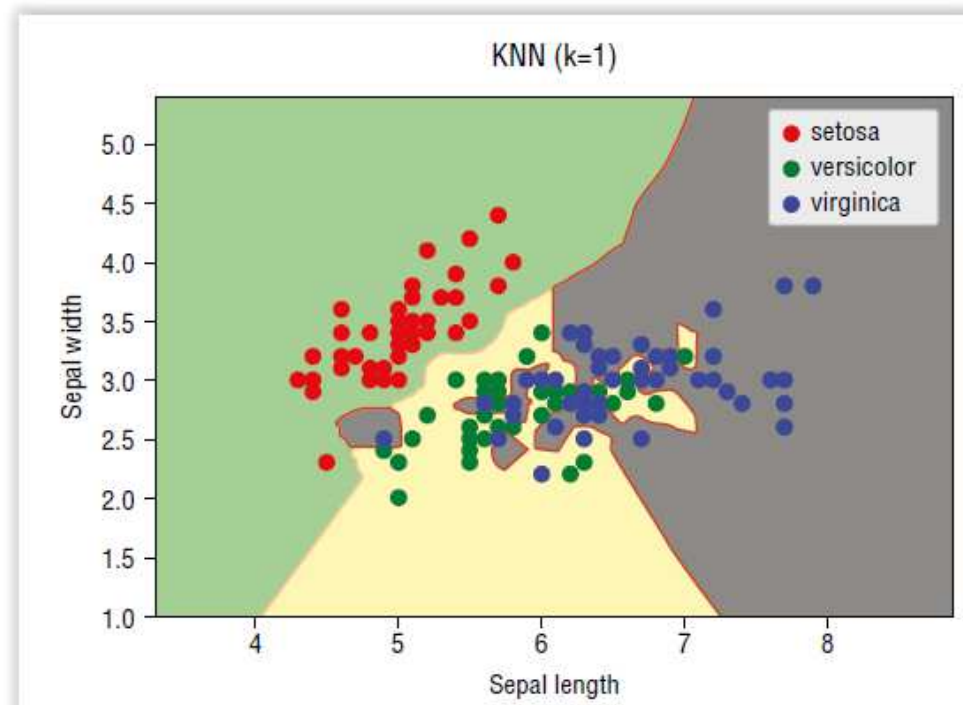
Le résultat dépend du choix de K :



Quand $K=1$, la prédiction est seulement à partir d'un échantillon → elle est très sensible aux distorsions (données aberrantes, mal labélisées, ...)

Quand K est trop grand, la prédiction est robuste aux distorsions mais il y aura davantage de données mal classifiées.

Un exemple:



F. K-means (K-moyennes)

Contrairement aux méthodes précédentes, il s'agit d'une méthode de classification non-supervisée.

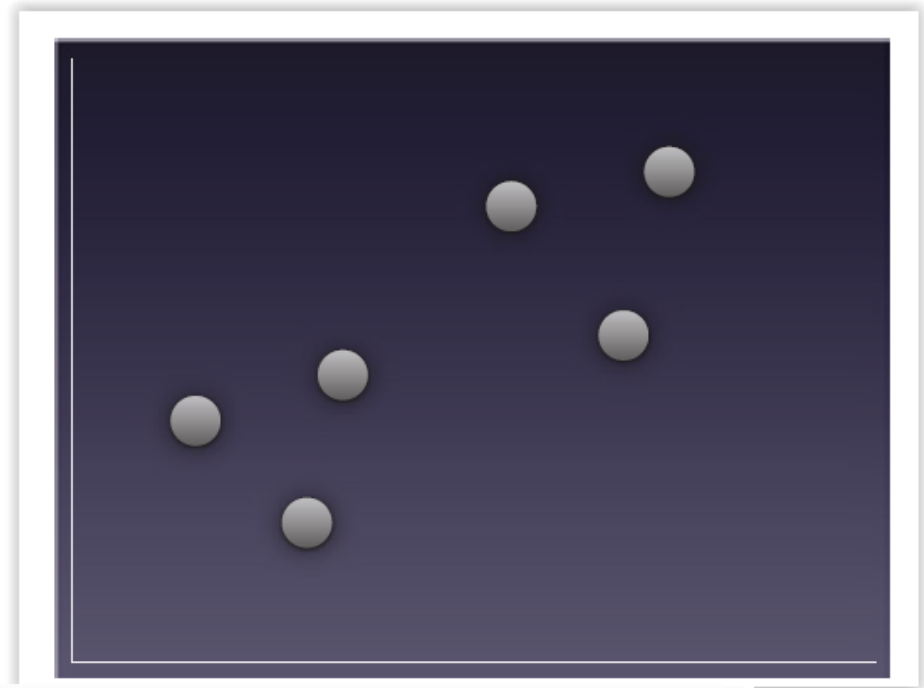
Étant donné un ensemble de n points ($\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$) non-labélisés, on cherche à les partitionner en K classes (appelées *clusters*) en minimisant les distances entre les points à l'intérieur de chaque cluster.

Algorithme:

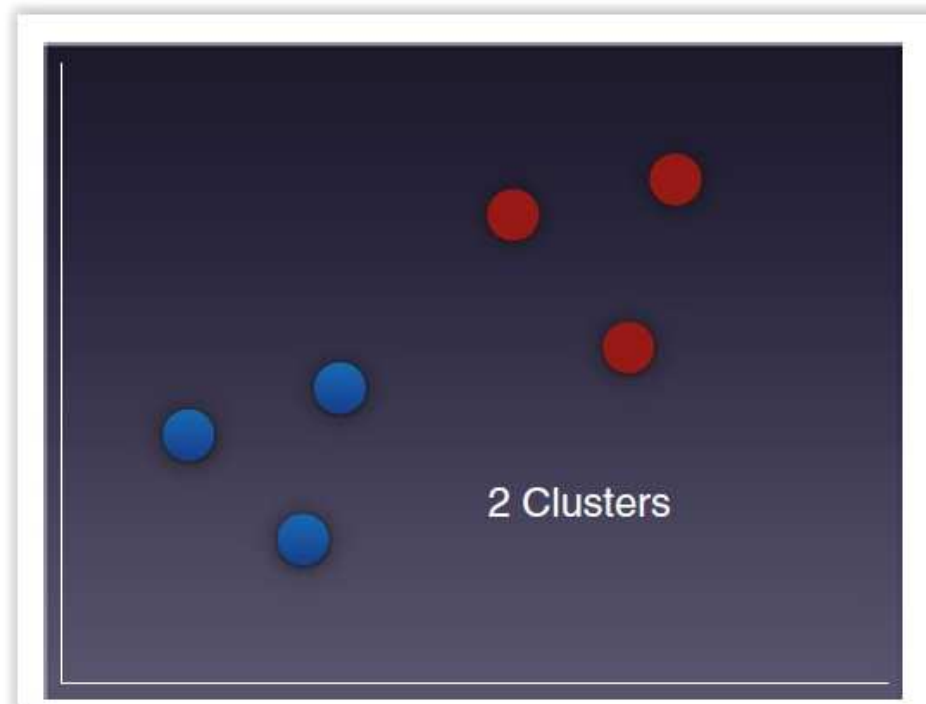
- Choisir K points (au hasard par exemple) qui représentent les positions moyennes (les *centroïdes*) initiales de K clusters.
- Répéter jusqu'à convergence :
 - affecter chaque observation au cluster dont le centroïde est le plus proche.
 - mettre à jour le centroïde de chaque cluster en calculant la moyenne de toutes les observations affectées à ce cluster.

Un exemple:

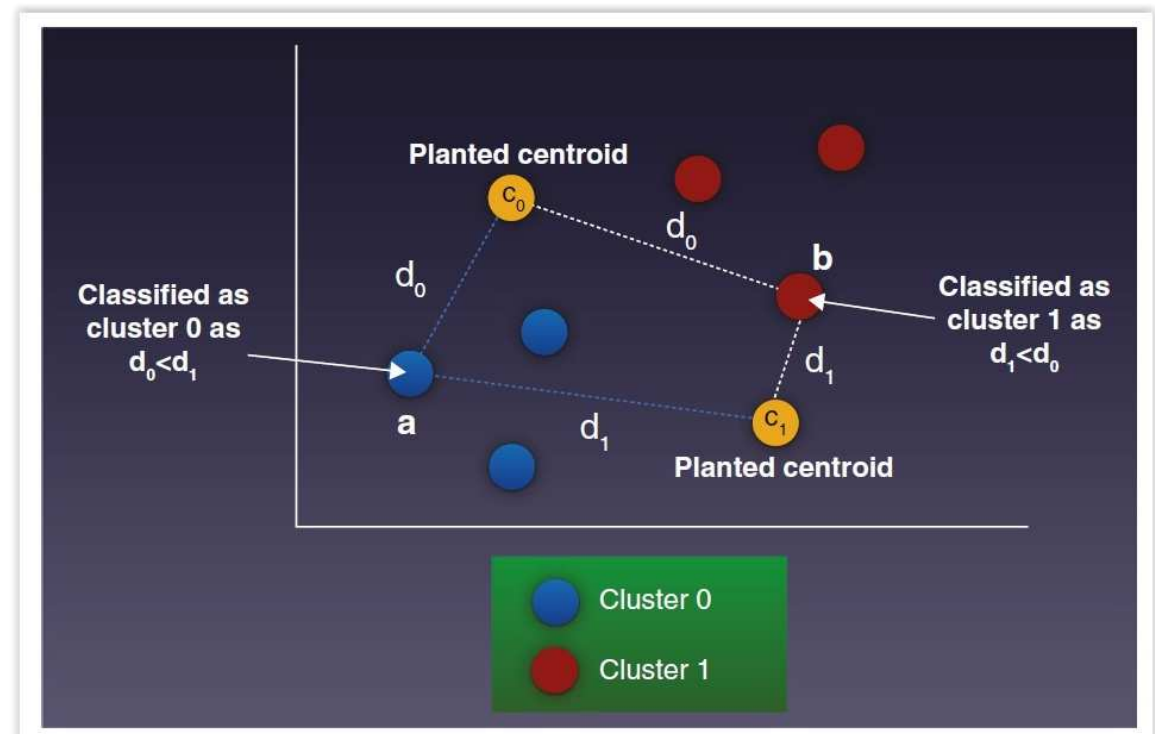
Un ensemble de données non-
labélisées



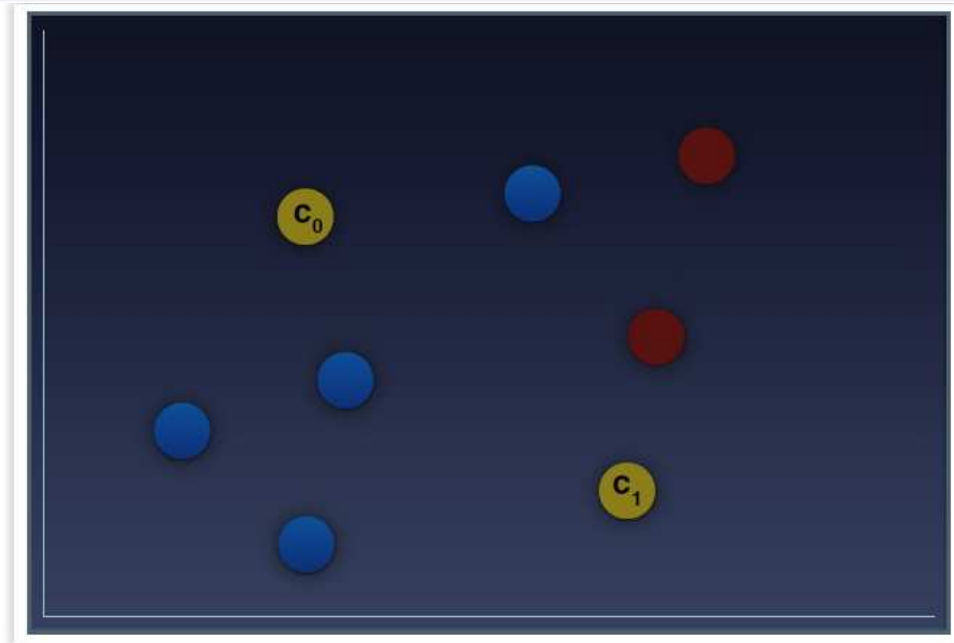
Classification idéale de ces
données en 2 clusters



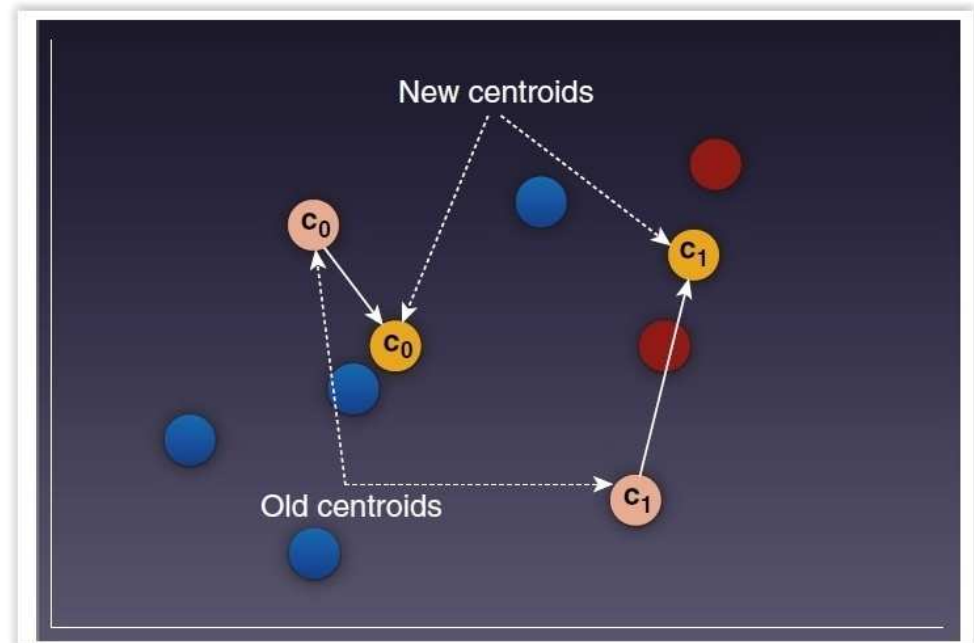
Mesurer la distance de chaque point par rapport à chaque centroïde pour trouver la distance la plus courte



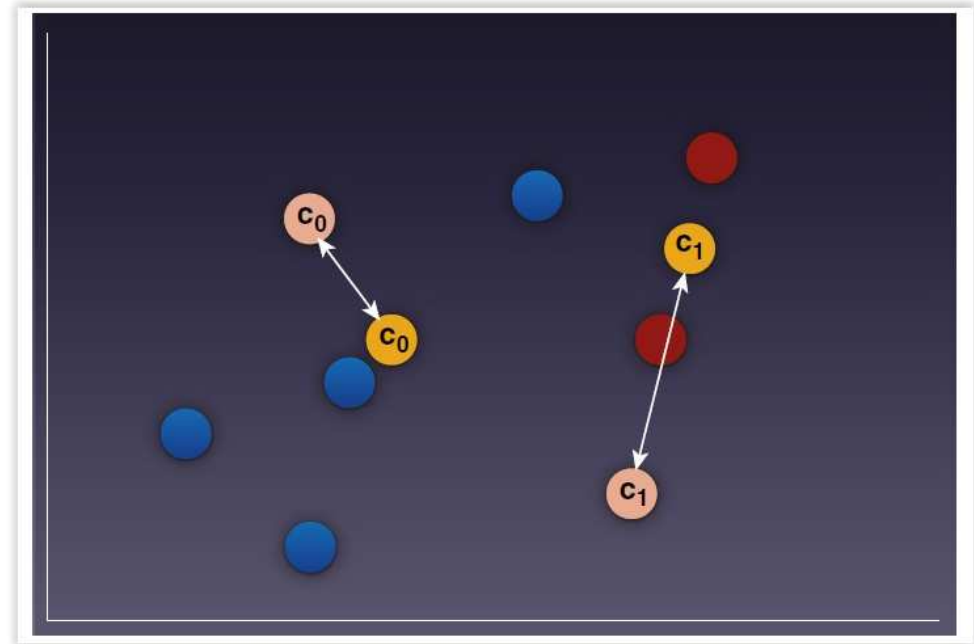
Partionnement des points après la première iteration de l'algorithme



Repositionnement des centroïdes en calculant la moyenne de tous les points de chaque cluster



Mesure de la distance entre le nouveau et l'ancien centroïdes. Si cette distance est nulle pour tous les centroïdes, l'algorithme s'arrête



Chapitre 2: Bibliothèques Python pour l'apprentissage automatique

A. NumPy

Bibliothèque Python pour manipuler les matrices et les tableaux multidimensionnels contenant des fonctions mathématiques qui opèrent sur ces tableaux.

Avant l'utilisation de NumPy, il faut l'importer: `import numpy as np`

Les tableaux NumPy sont de type *ndarray*. Tous les éléments d'un tableau doivent être de même type.

Exemples de création de tableaux:

```
a1 = np.arange(10) # creates a range from 0 to 9
print(a1)          # [0 1 2 3 4 5 6 7 8 9]
print(a1.shape)    # (10,)
```

```
a2 = np.arange(0,10,2) # creates a range from 0 to 9, step 2
print(a2)              # [0 2 4 6 8]
```

```

a3 = np.zeros(5)      # create an array with all 0s
print(a3)             # [ 0.  0.  0.  0.  0.]
print(a3.shape)       # (5,)
-----

a4 = np.zeros((2,3)) # array of rank 2 with all 0s; 2 rows
                      # and 3 columns
print(a4.shape)       # (2,3)
print(a4)
'''
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
'''
-----

a5 = np.full((2,3), 8) # array of rank 2 with all 8s
print(a5)
'''
[[8 8 8]
 [8 8 8]]
'''

```

```

a6 = np.eye(4) # 4x4 identity matrix
print(a6)
'''
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
'''
-----

a7 = np.random.random((2,4)) # rank 2 array (2 rows 4
                             # columns) with random values
                             # in the half-open interval
                             # [0.0, 1.0)

print(a7)
'''
[[ 0.48255806  0.23928884  0.99861279  0.4624779 ]
 [ 0.18721584  0.71287041  0.84619432  0.65990083]]
'''

```

```
list1 = [1,2,3,4,5] # list1 is a list in Python
r1 = np.array(list1) # rank 1 array
print(r1)           # [1 2 3 4 5]
```

Exemples d'indexage de tableaux:

```
print(r1[0]) # 1
print(r1[1]) # 2
-----
list2 = [6,7,8,9,0]
r2 = np.array([list1,list2]) # rank 2 array
print(r2)
'''
[[1 2 3 4 5]
 [6 7 8 9 0]]
'''
print(r2.shape) # (2,5) - 2 rows and 5 columns
print(r2[0,0])  # 1
print(r2[0,1])  # 2
print(r2[1,0])  # 6
```

```
list1 = [1,2,3,4,5]
r1 = np.array(list1)
print(r1[[2,4]])      # [3 5]
```

Indexage Booléen

```
print(r1>2)   # [False False True True True]
```

```
print(r1[r1>2])      # [3 4 5]
```

```
nums = np.arange(20)
```

```
print(nums) # [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
              # 17 18 19]
```

```
odd_num = nums[nums % 2 == 1]
```

```
print(odd_num) # [ 1 3 5 7 9 11 13 15 17 19]
```

Découpage de tableaux

```
a = np.array([[1,2,3,4,5],
              [4,5,6,7,8],
              [9,8,7,6,5]])
b1 = a[1:3, :3]    # row 1 to 3 (not inclusive) and first 3
                  # columns
print(b1)
'''
[[4 5 6]
 [9 8 7]]
'''

b2 = a[-2:,-2:]
print(b2)
'''
[[7 8]
 [6 5]]
'''
```

Attention: Le résultat du découpage dépend de la façon avec laquelle le découpage est réalisé. Par exemple:

```
b4 = a[2:, :]          # row 2 onwards and all columns
print(b4)              # [[9 8 7 6 5]]
print(b4.shape)        # (1,5)
```

```
b5 = a[2, :]          # row 2 and all columns
print(b5)              # [9 8 7 6 5]
                        # b5 is rank 1
print(b5.shape)        # (5,)
```

Exemples de remise en forme de tableaux (array reshaping):

```
b5 = b5.reshape(1,-1)
print(b5)              # [[9 8 7 6 5]]
b4.reshape(-1,)        # [9 8 7 6 5]
```

Attention: Pour convertir un tableau de 2D en un tableau de 1D, on peut utiliser les fonctions `flatten()` ou `ravel()`. `flatten()` renvoie une copie du tableau alors que `ravel()` et `reshape()` renvoient une référence.

Exemples d'opérations mathématiques sur les tableaux:

```
x1 = np.array([[1,2,3],[4,5,6]])
y1 = np.array([[7,8,9],[2,3,4]])
print(x1 + y1)           # same as np.add(x1,y1)
                          # [[ 8 10 12]
                          #  [ 6  8 10]]

print(x1 - y1)           # same as np.subtract(x1,y1)
print(x1 * y1)           # same as np.multiply(x1,y1)
print(x1 / y1)           # same as np.divide(x1,y1)
```

Produit scalaire:

```
x = np.array([2,3])
y = np.array([4,2])
np.dot(x,y)              # 2*4 + 3*2 = 14

x2 = np.array([[1,2,3],[4,5,6]])
y2 = np.array([[7,8],[9,10],[11,12]])
print(np.dot(x2,y2))     # matrix multiplication
                          # [[ 58  64]
                          #  [139 154]]
```

Matrices: NumPy possède aussi une classe `matrix`. La principale différence avec `ndarray` est que `matrix` est toujours 2D alors que `ndarray` peut être d'une dimension quelconque. On peut convertir un tableau NumPy `ndarray` en une matrice avec la fonction `asmatrix()` :

```
x1 = np.array([[1,2],[4,5]])
x1 = np.asmatrix(x1)
```

Une autre différence notable entre `ndarray` et `matrix` concerne la multiplication: le symbole `*` signifie un produit élément par élément pour les `ndarray` mais un produit matriciel pour `matrix`:

```
x1 = np.array([[1,2],[4,5]])
y1 = np.array([[7,8],[2,3]])
print(x1 * y1)          # element-by-element multiplication
                        # [[ 7 16]
                        #  [ 8 15]]

x2 = np.matrix([[1,2],[4,5]])
y2 = np.matrix([[7,8],[2,3]])
print(x2 * y2)          # dot product; same as np.dot()
                        # [[11 14]
                        #  [38 47]]
```

Somme cumulée:

```
print(a.cumsum(axis=0))      # sum over rows
print(a.cumsum(axis=1))      # sum over columns
```

Tri:

```
ages = np.array([34,12,37,5,13])
sorted_ages = np.sort(ages)      # does not modify the
                                  # original array
ages.sort()                      # modifies the array
-----
ages = np.array([34,12,37,5,13])
print(ages.argsort())           # indices: [3 1 4 0 2]
```

Affectation des tableaux: si a1 et a2 sont 2 tableaux:

1) Copying by reference

```
a2 = a1                        # creates a copy by reference
```

a2 pointe toujours sur le a1 original → si l'un des 2 tableaux est modifié, l'autre sera également modifié. Si la forme (la dimension) de l'un des 2 tableaux est modifiée, la forme de l'autre sera également modifiée.

2) Copying by view (Shallow copy)

```
a2 = a1.view() # creates a copy of a1 by reference; but  
               # changes in dimension in a1 will not affect a2
```

La modification des valeurs d'un tableau affecte la modification des valeurs de l'autre, mais la modification de la forme de l'un n'a pas d'effet sur l'autre

3) Copying by value (deep copy)

```
a2 = a1.copy() # create a copy of a1 by value (deep copy)
```

Les modifications de l'un n'ont pas d'effet sur l'autre

B. Pandas

Bibliothèque Python pour la manipulation et l'analyse de données. Il fournit principalement 2 types de structures de données : `Series` et `DataFrame`.

Avant l'utilisation de `pandas`, il faut l'importer: `import pandas as pd`

1) Series

Un « Pandas series » est un tableau à une dimension où chaque élément a un indice (0, 1, ...). Il ressemble plutôt à un dictionnaire de Python (incluant des indices).

```
series = pd.Series([1,2,3,4,5])
```

```
print(series)
```

```
# 0 1
```

```
# 1 2
```

```
# 2 3
```

```
# 3 4
```

```
# 4 5
```

On peut changer les indices par défaut :

```
series = pd.Series([1,2,3,4,5], index=['a','b','c','d','c'])
print(series)
# a 1
# b 2
# c 3
# d 4
# c 5
```

- `print(series[2])` ou `print(series.iloc[2])` rendent un élément de la série à partir de sa position, ici élément numéro 2 qui est 3.
- `print(series['c'])` ou `print(series.loc['c'])` rendent un élément de la série à partir de son indice, ici 3 et 5.
- on peut découper une partie d'une série. `print(series[2:])` rend les 3 derniers éléments de la série ci-dessus avec leurs indices.

2) DataFrame

Un « DataFrame » est un tableau à 2 dimensions avec indices. Chaque colonne d'un dataframe est un « pandas series ».

```
df=pd.DataFrame(np.random.randn(10,4),columns=list('ABCD'))  
print(df)
```

crée un dataframe de taille 10*4 rempli avec random, et chaque colonne a un label : A, B, C, D

	A	B	C	D
0	0.187497	1.122150	-0.988277	-1.985934
1	0.360803	-0.562243	-0.340693	-0.986988
2	-0.040627	0.067333	-0.452978	0.686223
3	-0.279572	-0.702492	0.252265	0.958977
4	0.537438	-1.737568	0.714727	-0.939288
5	0.070011	-0.516443	-1.655689	0.246721
6	0.001268	0.951517	2.107360	-0.108726
7	-0.185258	0.856520	-0.686285	1.104195
8	0.387023	1.706336	-2.452653	0.260466
9	-1.054974	0.556775	-0.945219	-0.030295

On peut charger le contenu d'un fichier CSV dans un dataframe:

```
df = pd.read_csv('data.csv')
```

Exemple de manipulations de dataframes:

- `print(df.index)` affiche les indices, et `print(df.values)` affiche le contenu de dataframe.
- `print(df.describe())` permet d'afficher les statistiques d'un dataframe (nombre, moyenne, std, min, max et différentes quartiles).
- `print(df.mean(0))` affiche la moyenne de chaque colonne, `print(df.mean(1))` la moyenne de chaque ligne.
- `print(df.head(8))` affiche les 8 premières lignes de dataframe, `print(df.tail(8))` affiche les 8 dernières lignes.
- `print(df['A'])` ou `print(df.A)` affiche la colonne de label A (avec ses indices)
- `print(df[['A', 'B']])` affiche les colonnes de labels A et B
- `print(df[2:6])` ou `print(df.iloc[2:6])` affiche les lignes N. 2 ,3,4,5
- `print(df.iloc[[2, 6]])` affiche les lignes 2 et 6 (ne marche pas sans iloc)
- `print(df.iloc[2])` affiche la ligne 2 (ne marche pas sans iloc)
- `print(df.iloc[2:4, 1:4])` affiche les lignes 2 et 3 et les colonnes 1,2,3

- `print(df.iloc[[2,4],[1,3]])` affiche les lignes 2 et 4 et les colonnes 1 et 3
- `print(df[(df.A > 0) & (df.B>0)])` : extraction à partir des conditions sur les colonnes A et B (pour toutes les lignes)
- `print(df.transpose())` ou `print(df.T)` pour transposé
- La fonction 'apply' permet d'appliquer une fonction à un dataframe. Les objets passés à cette fonction sont des séries dont l'indice est soit l'indice de dataframe (index=0) soit les colonnes de dataframe (index=1). Par exemple, nous pouvons définir deux fonctions lambda de la façon suivante :

```
import math
```

```
sq_root = lambda x: math.sqrt(x) if x > 0 else x
```

```
sq = lambda x: x**2
```

Si on écrit ensuite `print(df.B.apply(sq_root))`, la fonction `sq_root` sera appliquée à la colonne B de dataframe.

Si on veut appliquer `sq_root` à tous les éléments de dataframe, il faut écrire :

```
for column in df:
```

```
    df[column] = df[column].apply(sq_root)
```

```
print(df)
```

- `print(df[df.name != 'Nad'])` : supprime la ligne dont dans la colonne 'name' il y a 'Nad'
- `print(df.drop(df.index[1]))` : supprime la ligne N. 1
- `print(df.drop(df.index[-2]))` : supprime l'avant dernière ligne
- `print(df.drop('reports', axis=1))` : supprime la colonne de label 'reports'
- La fonction 'crosstab' permet de créer un tableau, montrant la distribution de 2 variables. Par exemple, si on crée un dataframe de la façon suivante :

```
df = pd.DataFrame(
    {
        "Gender": ['Male', 'Male', 'Female', 'Female', 'Female'],
        "Team"  : [1, 2, 3, 3, 1]
    })
```

en écrivant `print(pd.crosstab(df.Gender, df.Team))`, on obtient :

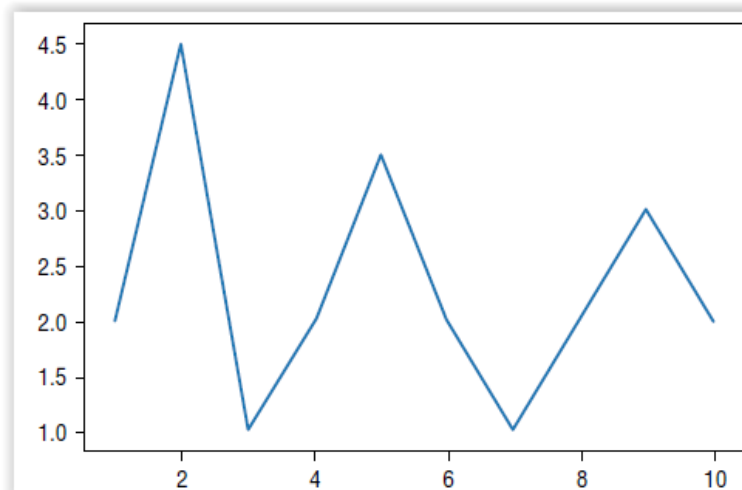
```
Team    1  2  3
Gender
Female  1  0  2
Male    1  1  0
```

C. Matplotlib et Seaborn

Bibliothèques Python pour tracer et visualiser des données sous forme de graphiques. Seaborn est une librairie de visualisation complémentaire basée sur matplotlib.

Quelques exemples:

```
import matplotlib.pyplot as plt  
plt.plot(  
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],  
    [2, 4.5, 1, 2, 3.5, 2, 1, 2, 3, 2]  
)
```



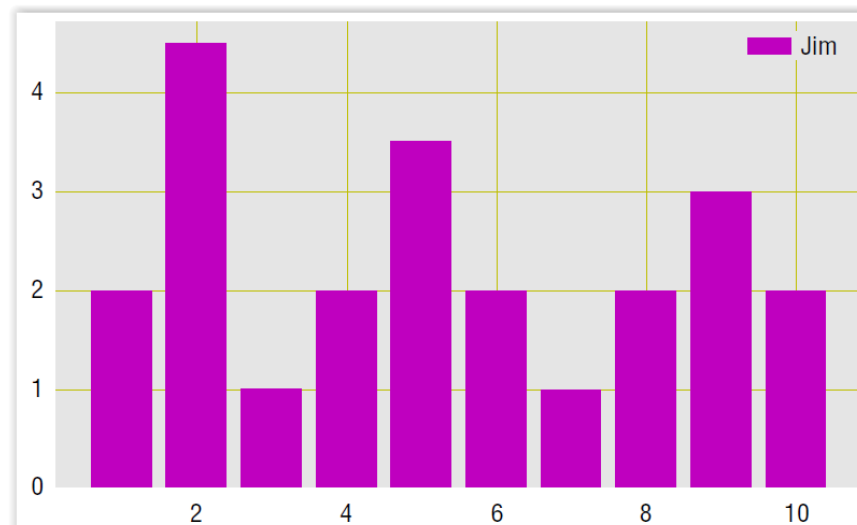
- Pour ajouter titre, xlabel et ylabel :

```
plt.title("Results")           plt.xlabel("Semester")  
plt.ylabel("Grade")
```

- On peut avoir plusieurs plot dans la même figure (*hold on* automatique)

- Pour tracer des diagrammes à barres :

```
plt.bar(  
    [1,2,3,4,5,6,7,8,9,10],  
    [2,4.5,1,2,3.5,2,1,2,3,2],  
    label = "Jim",  
    color = "m", # m for magenta  
    align = "center"  
)
```



- Pour un diagramme circulaire: `plt.pie`
- `plt.savefig("Fig.png", bbox_inches="tight")` enregistre la figure dans un fichier. Le 2ème paramètre (`tight`) enlève tout espace blanc autour de la figure.
- En ajoutant par exemple `'bo'` à la fin d'un `plt.plot`, on peut tracer un nuage de points (*b* pour bleu et *o* pour cercle)
- `plt.subplot(121)` permet de créer un subplot

Seaborn permet d'obtenir des figures plus sophistiquées avec moins de code que matplotlib :

```
import matplotlib.pyplot as plt
import seaborn as sns
```

- Tracer les figures catégorielles : Si les données « data » ont trois colonnes « gender, group, license », avec ce programme:

```
g = sns.catplot(x="gender", y="license", col="group",
               data=data, kind="bar", ci=None, aspect=1.0)
plt.show()
```

on trace 2 figures qui montrent le pourcentage des hommes et femmes possédant un permis, respectivement dans les groupes 1 et 2.

- Avant `plt.show`, on peut ajouter

```
g.set_axis_labels("", "Proportion with Driving license")
```

```
g.set_xticklabels(["Men", "Women"])
```

```
g.set_titles("{col_var} {col_name}")
```

pour régler les paramètres des figures.

- Seaborn a des bases de données internes qu'on peut télécharger avec la commande `load_dataset()`. Pour les connaître, on peut utiliser :
`sns.get_dataset_names()`.
- `sns.lmplot` permet de tracer des nuages de points avec seaborn
- `sns.swarmplot` est un nuage de points catégoriel avec des points sans chevauchement.

D. Scikit-learn

Bibliothèque Python pour l'apprentissage automatique.

1) Régression linéaire

Si on a par exemple les tailles et les poids de plusieurs individus dans deux tableaux *heights* et *weights*, et qu'on cherche un modèle linéaire entre les 2:

```
from sklearn.linear_model import LinearRegression
# Create and fit the model
model = LinearRegression()
model.fit(X=heights, y=weights)

# make prediction
weight = model.predict([[1.75]])[0][0]
print(round(weight,2)) # 76.04
```

Pour obtenir les paramètres du modèle

```
print(model.intercept_)
print(model.coef_)
```

2) Régression polynomiale

Pour faire une régression polynomiale avec une seule variable explicative, on utilise :

```
from sklearn.preprocessing import PolynomialFeatures  
  
degree = 2  
  
polynomial_features = PolynomialFeatures(degree = degree)
```

En utilisant l'objet `polynomial_features`, on peut générer les combinaisons polynomiales des *features* :

```
x_poly = polynomial_features.fit_transform(x)  
print(x_poly)
```

on obtient une matrice dont la 1ère colonne vaut 1, la 2ème colonne vaut les valeurs de x , et la 3ème colonne les valeurs de x^2 . On peut maintenant faire une régression linéaire sur ces valeurs :

```
model = LinearRegression()  
  
model.fit(x_poly, y)  
  
y_poly_pred = model.predict(x_poly)
```

3) Régression logistique

Si l'entrée est la taille d'une tumeur et la sortie la classe (0: maligne, 1: bénigne)

```
x = cancer.data[:,0] # mean radius
y = cancer.target # 0: malignant, 1: benign
from sklearn import linear_model
log_regress = linear_model.LogisticRegression()
#---train the model---
log_regress.fit(X = np.array(x).reshape(len(x),1),
                y = y)
#---print trained model intercept---
print(log_regress.intercept_) # [ 8.19393897]
#---print trained model coefficients---
print(log_regress.coef_) # [[-0.54291739]]
print(log_regress.predict_proba(20)) # [[0.93489354
                                     #  0.06510646]] proba de 0 et de 1
print(log_regress.predict(20)[0]) # 0     classe
```

4) SVM

Si on a 2 entrées x_1 et x_2 et une sortie r :

```
from sklearn import svm
#---Converting the Columns as Matrices---
points = data[['x1','x2']].values
result = data['r']
clf = svm.SVC(kernel = 'linear')
clf.fit(points, result)
print('Vector of weights (w) = ',clf.coef_[0])
print('b = ',clf.intercept_[0])
print('Indices of support vectors = ', clf.support_)
print('Support vectors = ', clf.support_vectors_)
print('Number of support vectors for each class = ',
      clf.n_support_)

# Making prediction
print(clf.predict([[3,3]])[0])    # 'B'
```

5) KNN

Si X contient les 2 entrées et y contient la sortie:

```
from sklearn.neighbors import KNeighborsClassifier
k = 1
#---instantiate learning model---
knn = KNeighborsClassifier(n_neighbors=k)
#---fitting the model---
knn.fit(X, y)
#---min and max for the first feature---
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
#---min and max for the second feature---
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
#---step size in the mesh---
h = (x_max - x_min)/100
#---make predictions for each of the points in xx,yy---
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
Z = knn.predict(np.c_[xx.ravel(), yy.ravel()])
```

6) K-means

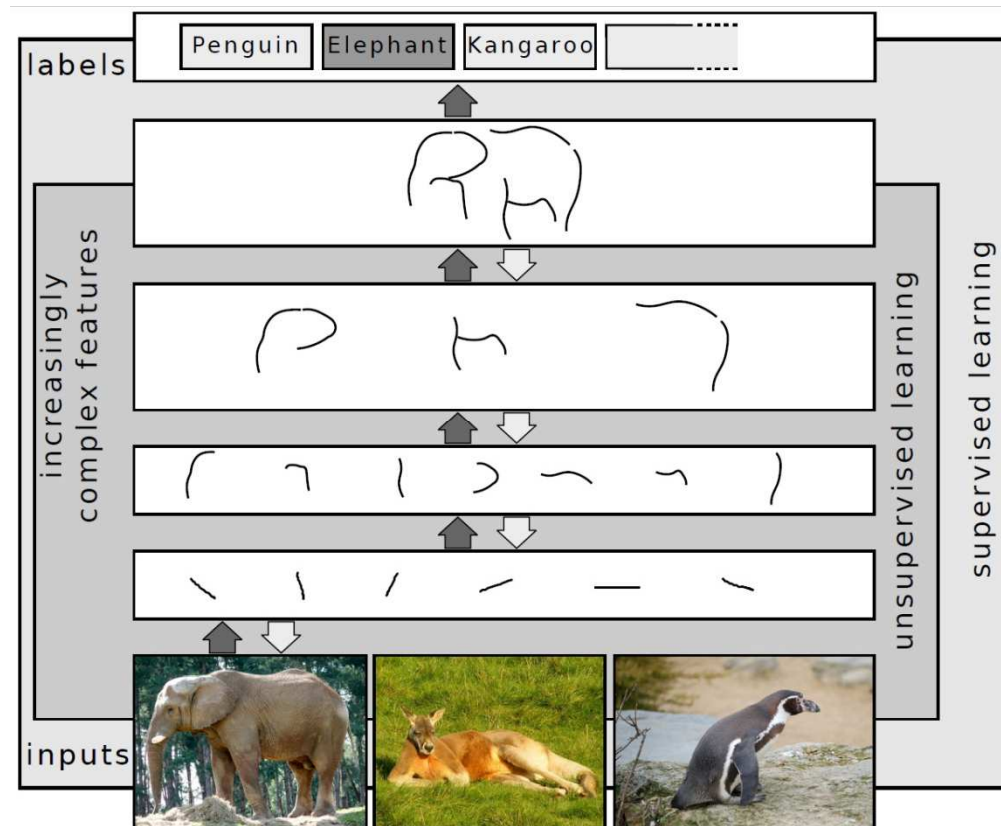
Si X contient les données

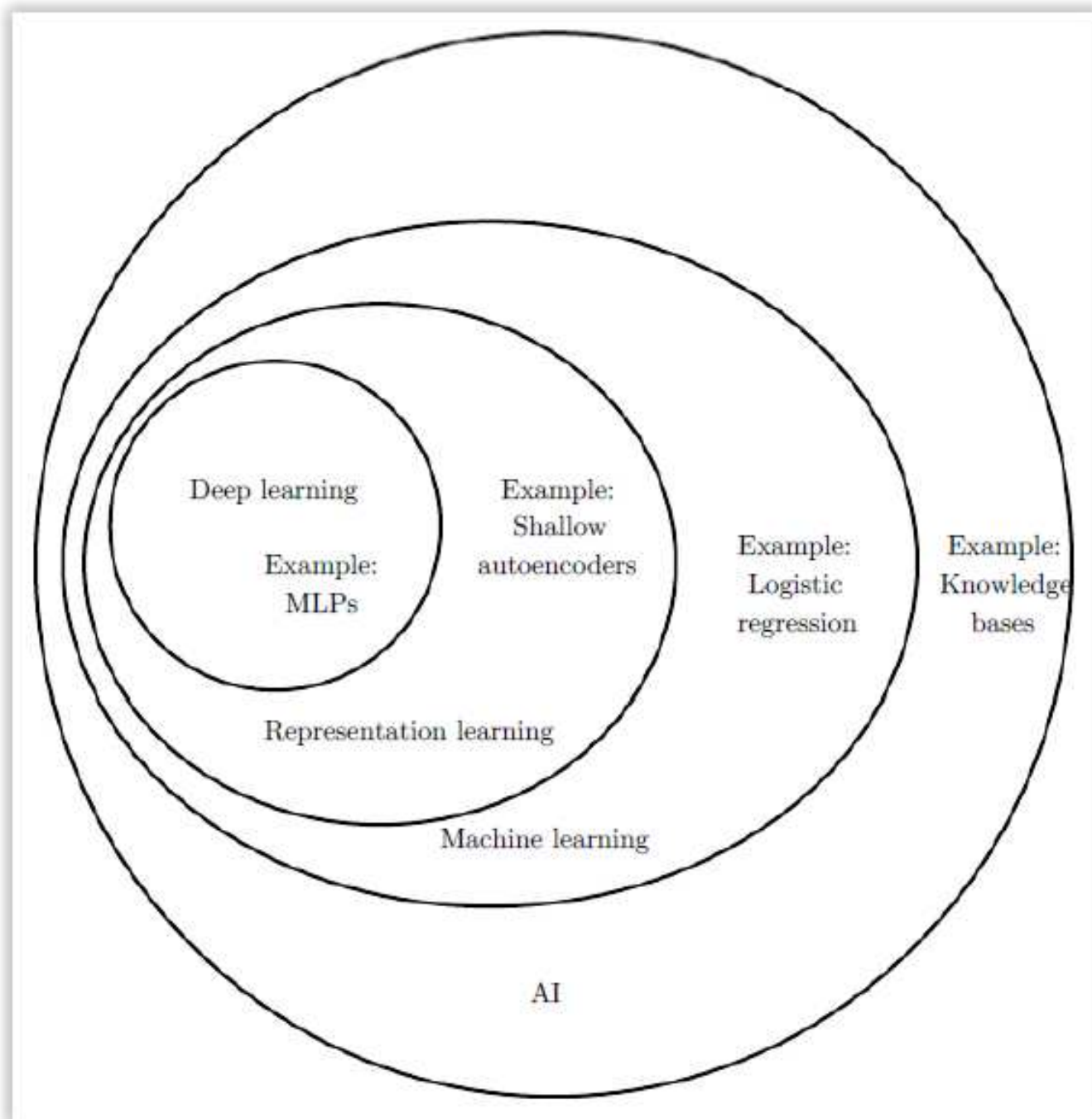
```
#---using sci-kit-learn---  
from sklearn.cluster import KMeans  
k=3  
kmeans = KMeans(n_clusters=k)  
kmeans = kmeans.fit(X)  
labels = kmeans.predict(X)  
centroids = kmeans.cluster_centers_  
print(labels)  
print(centroids)  
  
#---making predictions---  
cluster = kmeans.predict([[3,4]])[0]
```

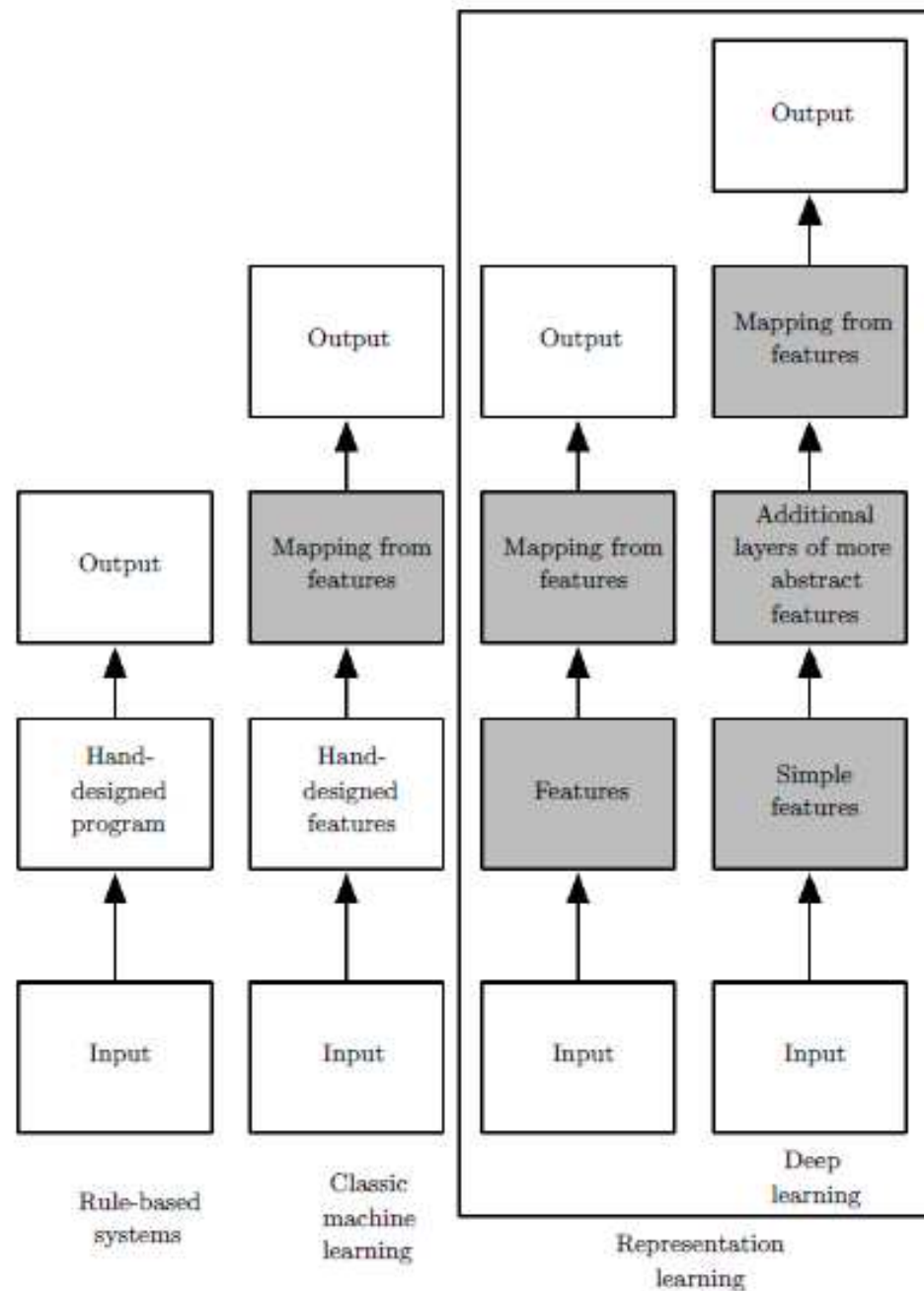
Chapitre 3: Apprentissage profond (Deep learning)

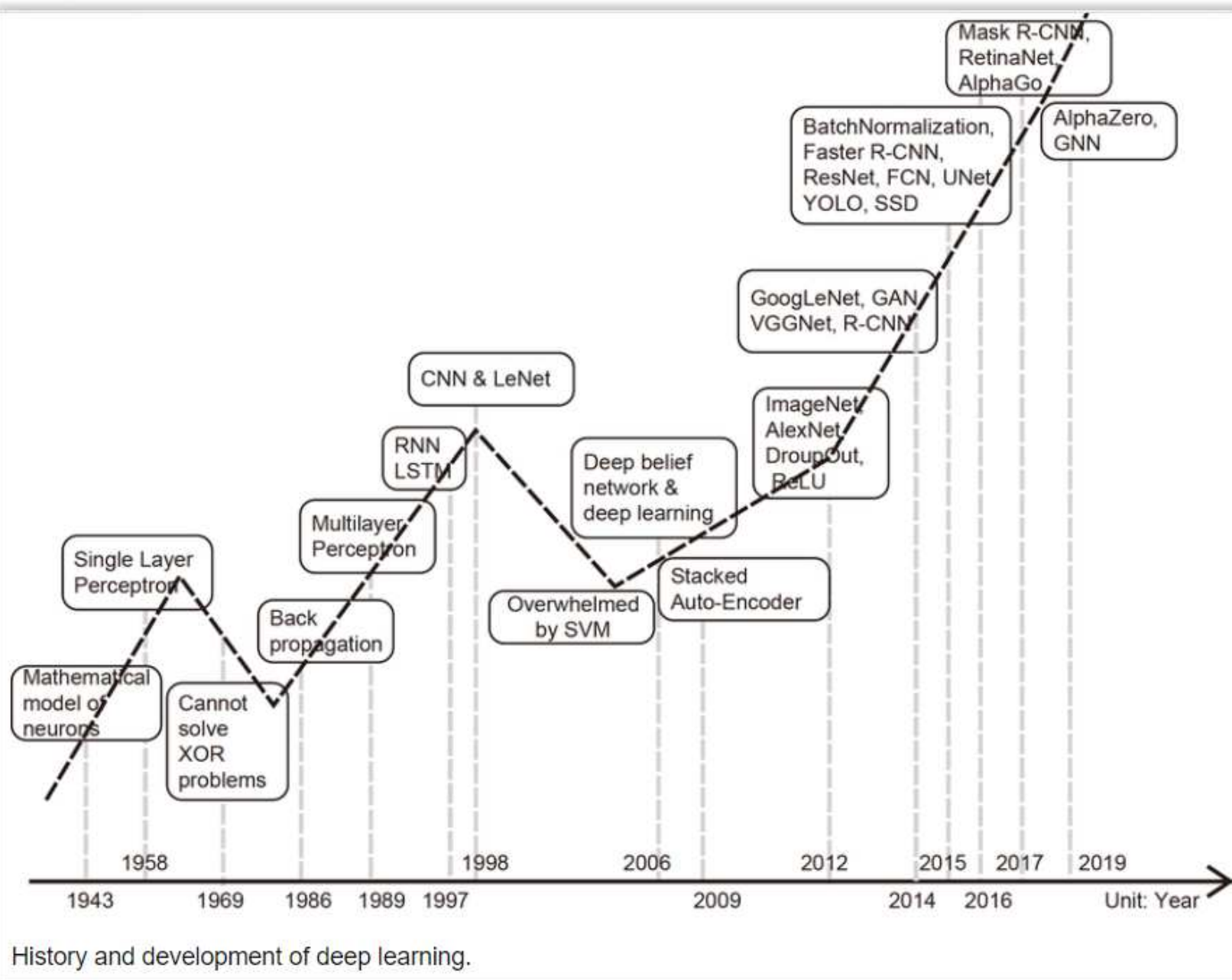
L'apprentissage profond est un type d'apprentissage automatique qui apprend à représenter le monde comme une hiérarchie de concepts, chaque concept étant défini à partir des concepts plus simples.

Principalement: avec les réseaux de neurones artificiels









Les réseaux de neurones artificiels: très populaires dans les années 90, un peu oubliés dans les années 2000.

Regain de popularité depuis 2012 sous l'appellation « Deep learning » (principalement « les réseaux de neurones convolutifs »).

Capables de réaliser la régression non-linéaire et la classification.

Raisons du regain de popularité des « Réseaux de neurones/Deep learning »:

- 1) Augmentation de la taille des bases de données disponibles (Big Data): numérisation de la société, stockage et partage de textes, images, vidéo et audio, ...
- 2) Augmentation de la taille des modèles (nombre de couches et de neurones): mémoire disponible, capacité de calcul (GPU, ...), ...
- 3) Augmentation de la précision, la complexité des tâches et l'impact dans le monde réel: reconnaissance d'objets, de la parole, ...

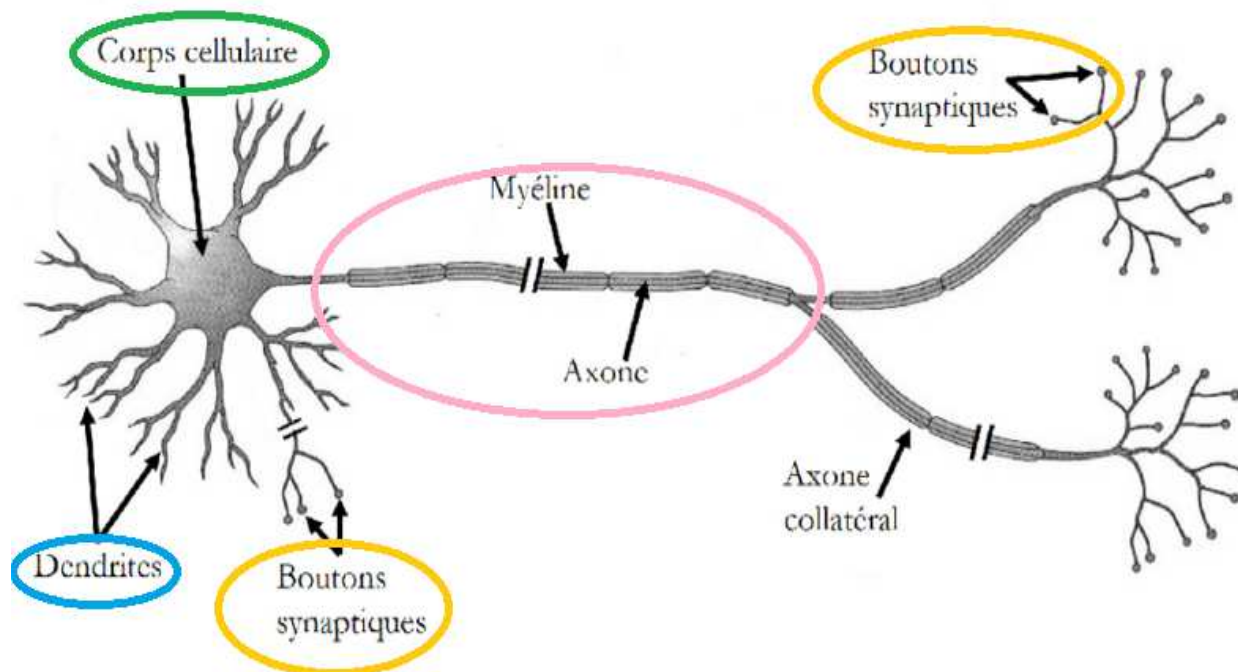
Réseaux de neurones biologiques (cerveaux)

Cerveau humain: 100 milliards de neurones

Chaque neurone reçoit le signal électrique des autres neurones qui y sont connectés, le traite, et le transmet aux neurones suivants.

L'intensité du signal transmis détermine si les neurones suivants sont excités à leur tour.

Les connexions du cerveau évoluent avec le temps, en fonction de multiples facteurs dont l'apprentissage (neuroplasticité)

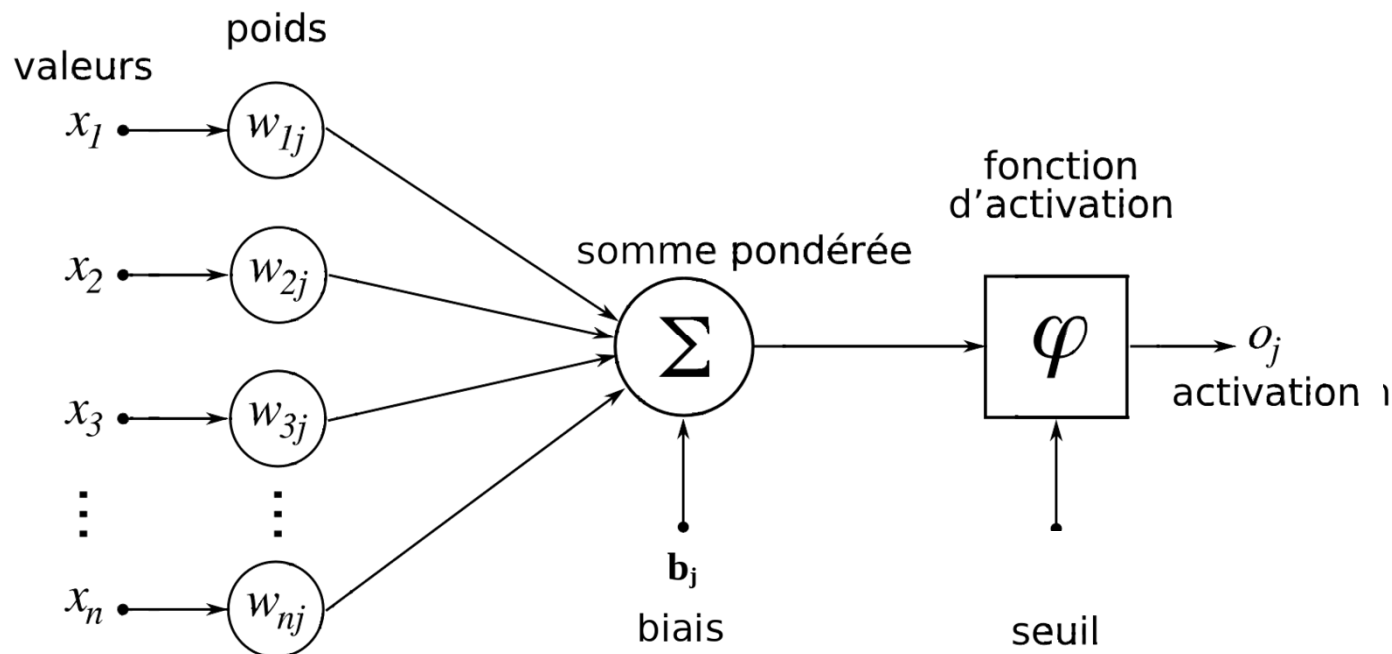


Neurone formel

La sortie d'un neurone (d'indice j) s'écrit:
$$o_j = \varphi \left(b_j + \sum_{i=1}^n w_{ij} x_i \right)$$

où o_j représente la sortie, les x_i sont les n entrées, les w_{ij} sont les n poids, b_j est le biais et φ fonction d'activation.

Les poids et les biais de tous les neurones d'un réseau constituent le vecteur du paramètre θ , qui sera déterminé pendant la phase d'apprentissage à partir des données labélisées.

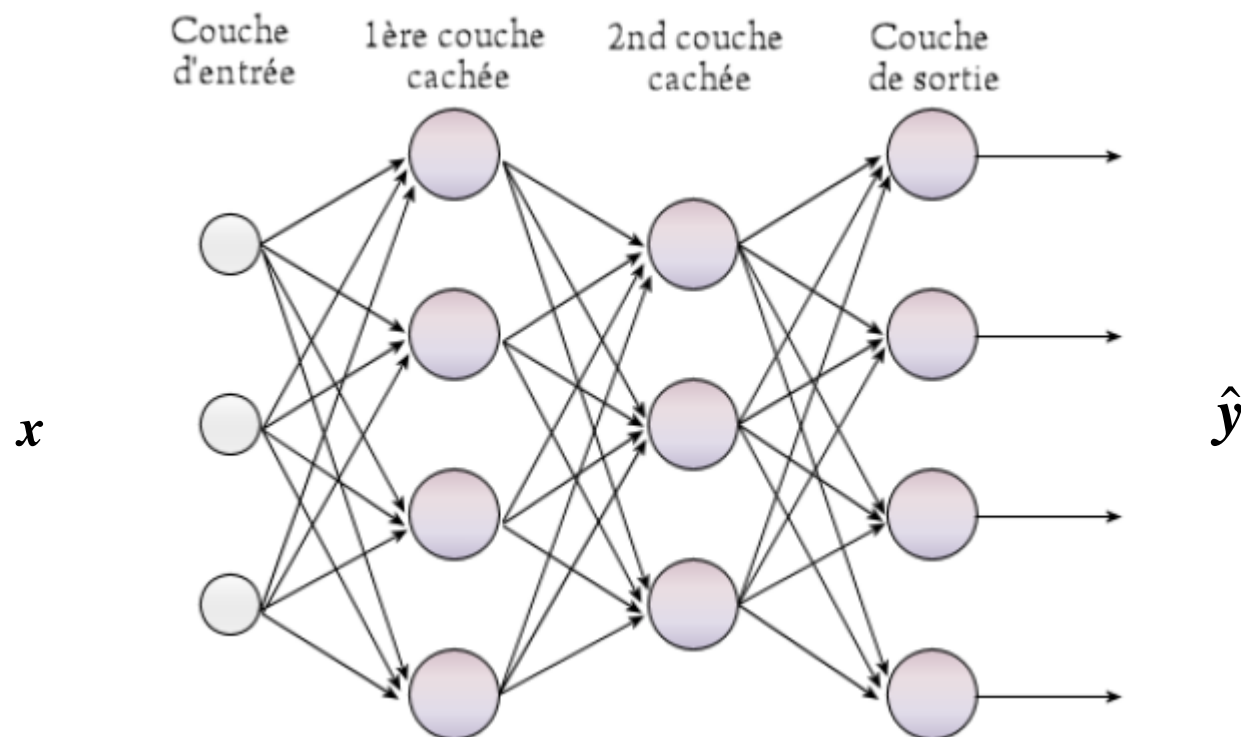


3.1. Perceptron multicouche (Multi-Layer Perceptron, MLP)

Un réseau de neurones artificiel organisé en plusieurs couches, chaque couche étant constituée de plusieurs neurones.

La fonction réalisée (pour régression ou classification): $\hat{y} = f(x, \theta)$

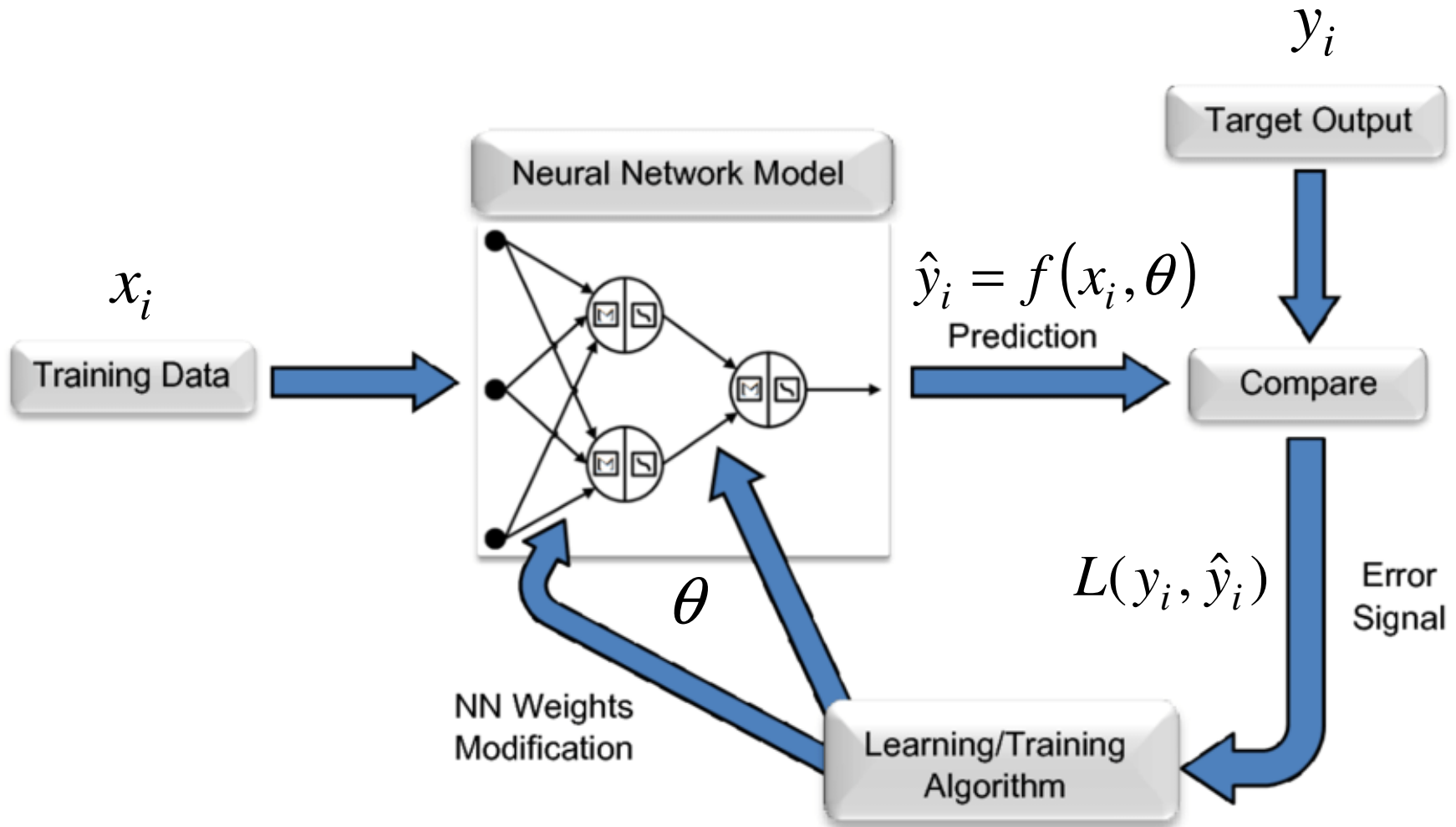
avec x le vecteur des entrées, \hat{y} le vecteur des sorties estimées par le réseau, θ le vecteur des paramètres du réseaux estimés pendant l'apprentissage.



Principe de fonctionnement

- On dispose d'une base de données labélisée: couples (x_i, y_i) avec x_i les entrées et y_i les sorties désirées connues.
- Cette base de donnée est souvent divisée en:
 - **base d'apprentissage (*training*)**: utilisée pour estimer les paramètres,
 - **base de test**: utilisée pour mesurer la capacité de généralisation.
- On choisit la structure du réseau: le nombre et la nature des couches, le nombre de neurones pour chaque couche, les fonctions d'activation, ...
- On définit une **fonction de coût** (*loss function* ou *cost function*) $L(y, \hat{y})$ qui mesure l'erreur du modèle: sa valeur doit être faible quand la sortie du réseau est proche de la sortie désirée sur la base d'apprentissage.
- On utilise un **algorithme d'optimisation** itératif, souvent basé sur le calcul du gradient, pour trouver la valeur optimale du vecteur de paramètres θ , c-à-d celle qui minimise la fonction de coût à partir de la base d'apprentissage (*training* ou *fit*).
- On utilise ce modèle (avec les poids obtenus) pour faire la prédiction à partir des nouvelles entrées.

Apprentissage d'un MLP



Fonctions de coût souvent utilisées pour un problème de régression

Si N représente le nombre d'exemples (individus) dans la base, la fonction de coût est la moyenne d'une « distance » sur les exemples:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^N D(y_i, \hat{y}_i) = \frac{1}{N} \sum_{i=1}^N D(y_i, f(x_i, \theta))$$

1) **Mean Squared Error** (Erreur quadratique moyenne): $D = (y_i - \hat{y}_i)^2$

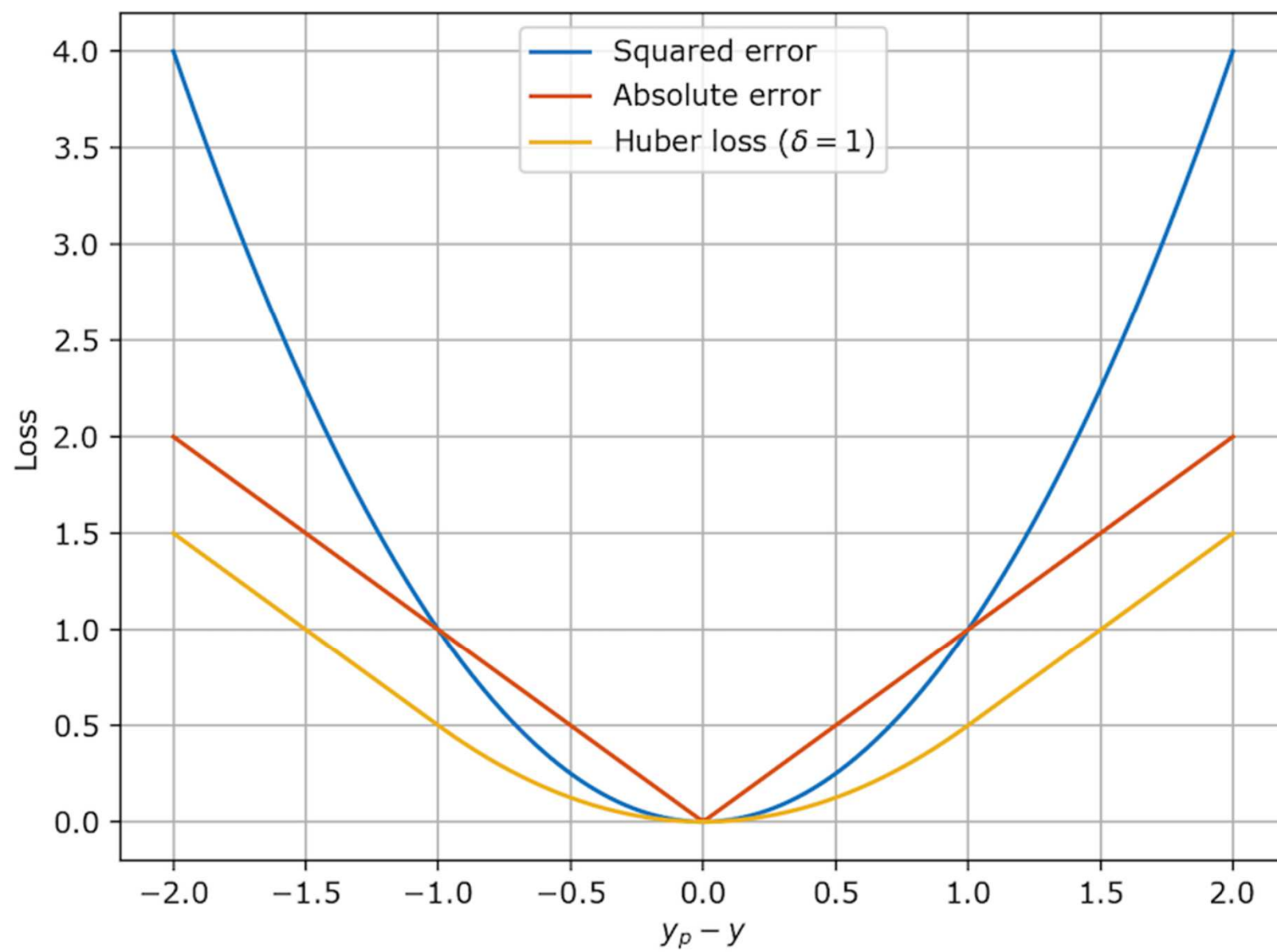
Dérivable partout mais sensible aux grandes erreurs sur peu d'individus.

2) **Mean Absolute Error** (Erreur absolue moyenne): $D = |y_i - \hat{y}_i|$

N'est pas dérivable à l'origine et toutes les erreurs auront le même poids dans une échelle linéaire.

3) **Huber**:
$$D(y_i, \hat{y}_i) = \begin{cases} 0.5(y_i - \hat{y}_i)^2 & \text{si } |y_i - \hat{y}_i| \leq \delta \\ \delta(|y_i - \hat{y}_i| - 0.5\delta) & \text{sinon} \end{cases}$$

Compromis entre les 2: quadratique pour les erreurs faibles et absolue pour les erreurs élevées (dérivable à l'origine)



Fonctions de coût souvent utilisées pour un problème de classification

On suppose que les K sorties du réseaux représentent les probabilités des K classes:

$$\hat{y}_i(k) = p(\text{individu } i \in \text{classe } k) \quad k = 1, \dots, K$$

Categorical Cross entropy (log-loss): Moyenne sur tous les individus de:

$$D(y_i, \hat{y}_i) = - \sum_{k=1}^K y_i(k) \log(\hat{y}_i(k))$$

Binary cross entropy:

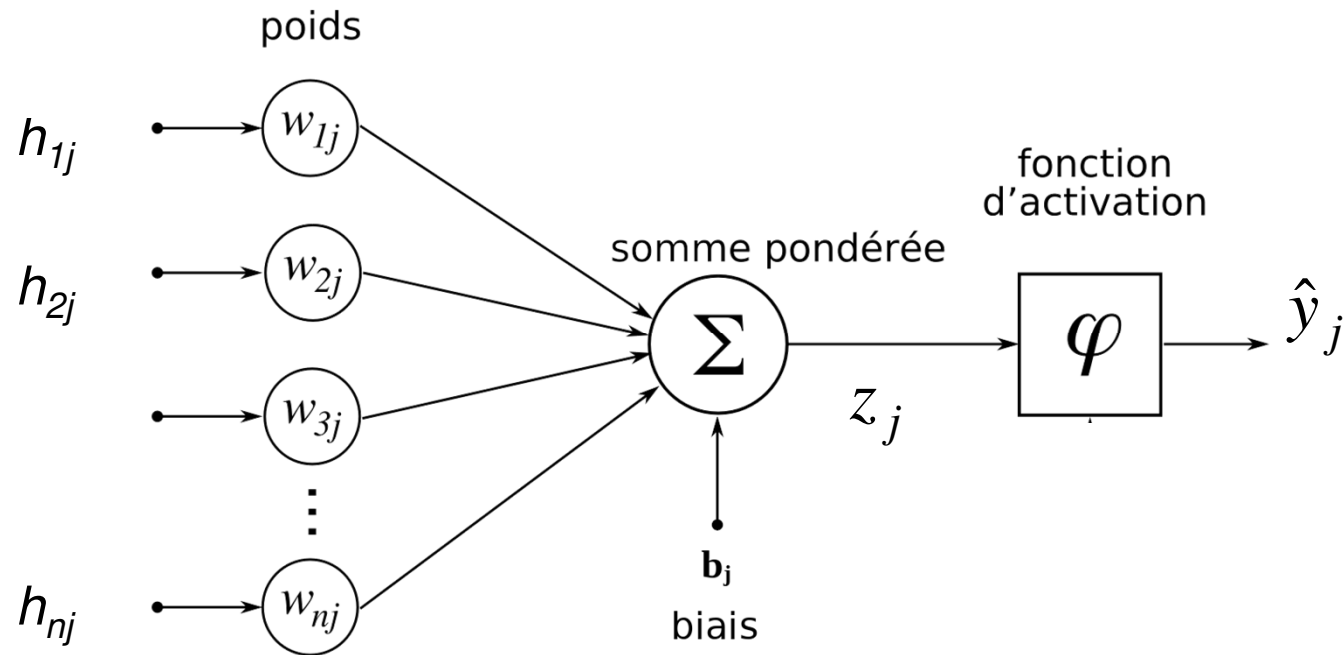
Dans un problème de classification binaire, si la probabilité d'une classe est y , la probabilité de l'autre classe est $1-y$ →

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

Couche de sortie

Les couches cachées fournissent un vecteur de caractéristiques (*features*) $\mathbf{h}=[h_1,\dots,h_n]^T$. La couche de sortie applique une transformation supplémentaire à ces caractéristiques pour réaliser une tâche.

Neurone de sortie:

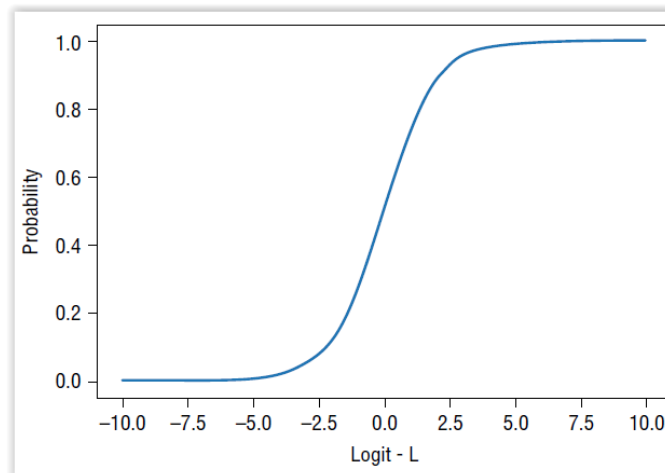


$$\hat{y} = \varphi(z) = \varphi\left(b + \sum_{i=1}^n w_i h_i\right) = \varphi\left(b + \mathbf{w}^T \mathbf{h}\right) \quad \text{avec} \quad \mathbf{w} = [w_1, \dots, w_n]^T.$$

Fonctions d'activation des neurones de la couche de sortie

1) **Unité linéaire.** Utilisée en régression, quand il n'y a pas de contrainte sur les sorties (positivité, ...): $\hat{y} = b + \mathbf{w}^T \mathbf{h}$

2) **Unité sigmoïdale.** Utilisée en classification binaire $\hat{y} = \text{sigm}(b + \mathbf{w}^T \mathbf{h})$ avec « sigm » la fonction sigmoïde définie au chapitre 1.



Inefficace quand la fonction de coût est « l'erreur quadratique moyenne », car le sigmoïde est souvent saturé → le gradient est presque nul.

Efficace quand la fonction de coût est « log-loss », car le logarithme de la fonction de coût se compense avec l'exponentielle de sigmoïde, évitant ainsi la saturation.

3) Unité Softmax. Utilisée en classification multi-classe.

Les K sorties $\hat{y}_{(i)}$, $i=1,\dots,K$, doivent représenter les probabilités de K classes → chacune doit être dans $[0,1]$ et leur somme doit être =1.

On définit:

$$z_{(1)} = b_{(1)} + \mathbf{w}_{(1)}^T \mathbf{h}, \quad \dots \quad , z_{(K)} = b_{(K)} + \mathbf{w}_{(K)}^T \mathbf{h}$$

et ensuite:

$$\hat{y}_{(i)} = \frac{\exp(z_{(i)})}{\exp(z_{(1)}) + \dots + \exp(z_{(K)})}, \quad i = 1, \dots, K$$

La fonction « softmax » est définie de la manière suivante:

$$[\hat{y}_{(1)}, \hat{y}_{(2)}, \dots, \hat{y}_{(K)}] = \text{softmax}([z_{(1)}, z_{(2)}, \dots, z_{(K)}])$$

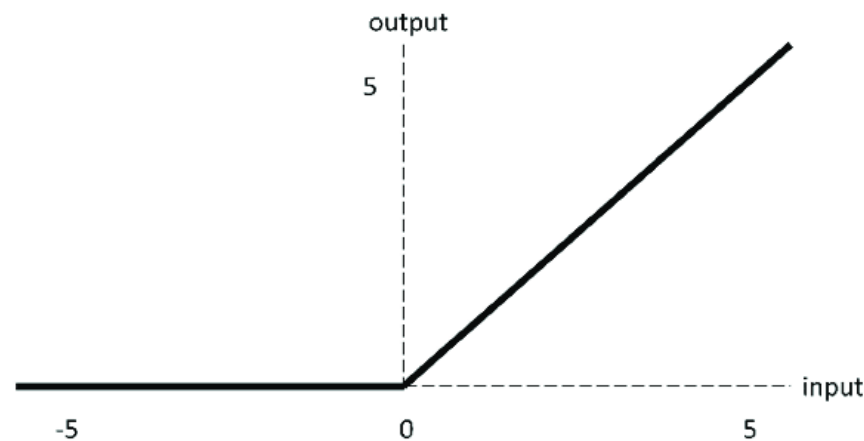
Inefficace quand la fonction de coût est « l'erreur quadratique moyenne ».

Efficace quand on utilise une fonction de coût de type « log-loss ».

Les fonctions d'activation des neurones des couches cachées

Si l'entrée de la fonction d'activation est notée $z = b + \mathbf{w}^T \mathbf{x}$, avec x les entrées d'un neurone caché et z son signal de sortie avant sa fonction d'activation. La sortie de la fonction d'activation sera notée h .

1) Rectified Linear Function (ReLU) $h = \max(0, z)$

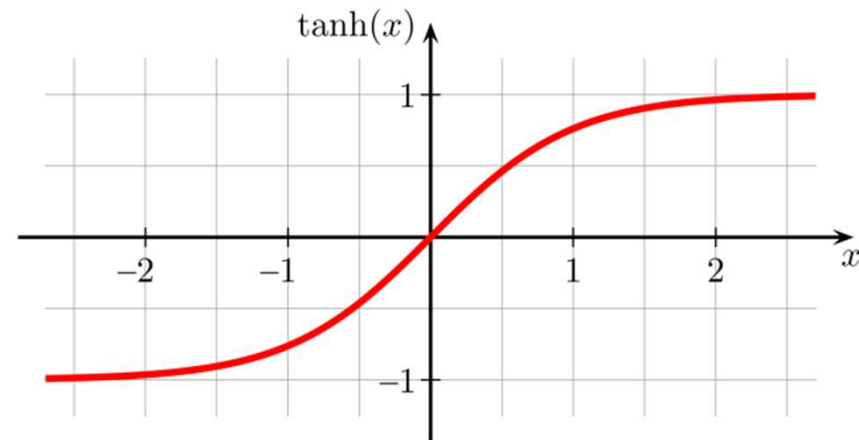


N'est pas dérivable à l'origine mais en pratique cela ne pose pas de problème. Le gradient est facilement calculable, avec des valeurs élevées quand le neurone est actif. Le neurone n'apprend pas quand son activation est nulle.

Plusieurs généralisations disponibles: GeLU, SeLU, PReLU, ...

2) Sigmoid

3) Tangente hyperbolique $h=\tanh(z)$



Inconvénient des sigmoïdes et tanh pour les couches cachées: Saturés sur une plage assez large de leurs entrées → Apprentissage avec les méthodes basées sur le gradient est difficile.

Tanh est souvent meilleur que Sigmoid car au voisinage de 0 il réagit comme une fonction linéaire.

4) **Linéaire** Possible pour quelques couches. Mais: si toutes les couches ont une activation linéaire, l'ensemble du réseau réalise une transformation linéaire (inefficace pour la plupart des problèmes).

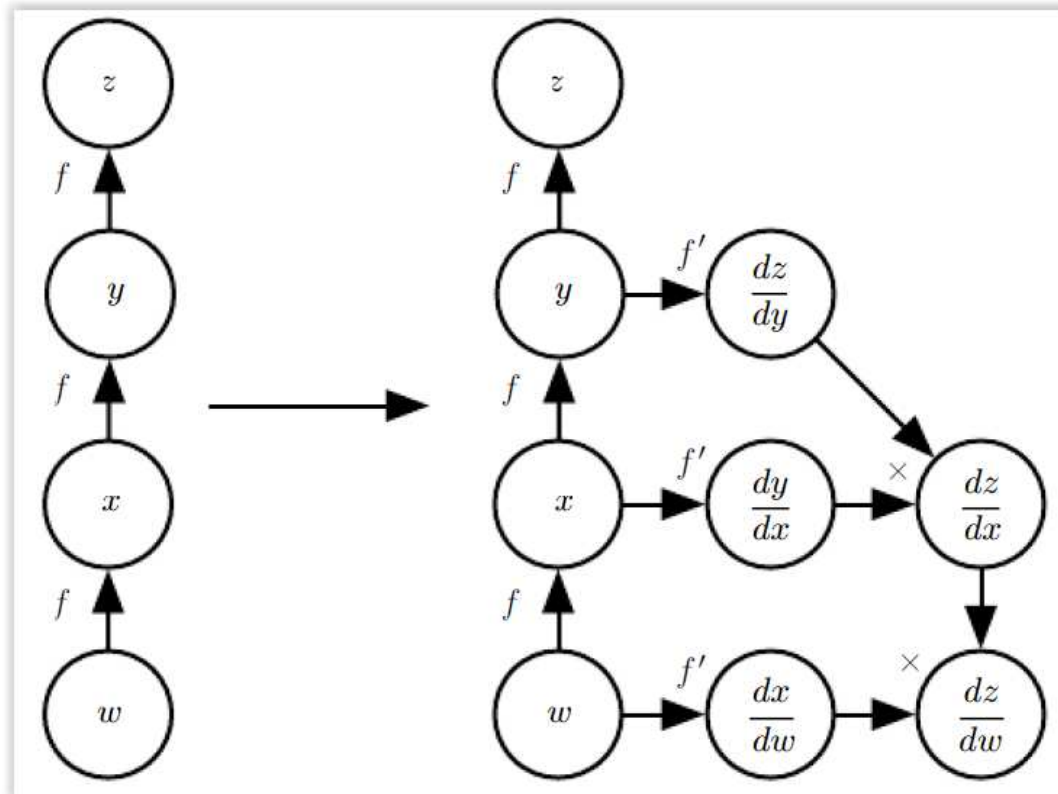
Rétropropagation du gradient (*backpropagation*)

Pour déterminer les paramètres du réseau, il faut utiliser un algorithme d'optimisation (détails dans la suite du cours).

La plupart de ces algorithmes sont basés sur le calcul du gradient: la dérivée de la fonction de coût par rapport à tous les paramètres.

En pratique, on utilise la règle de dérivation en chaîne de façon récursive, en commençant par la dernière couche.

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \frac{\partial x}{\partial w}$$



3.2. Régularisation pour l'apprentissage profond

C'est l'ensemble des stratégies qui permettent de réduire l'erreur de test (et donc d'améliorer la capacité de généralisation du réseau) quitte à augmenter l'erreur d'apprentissage.

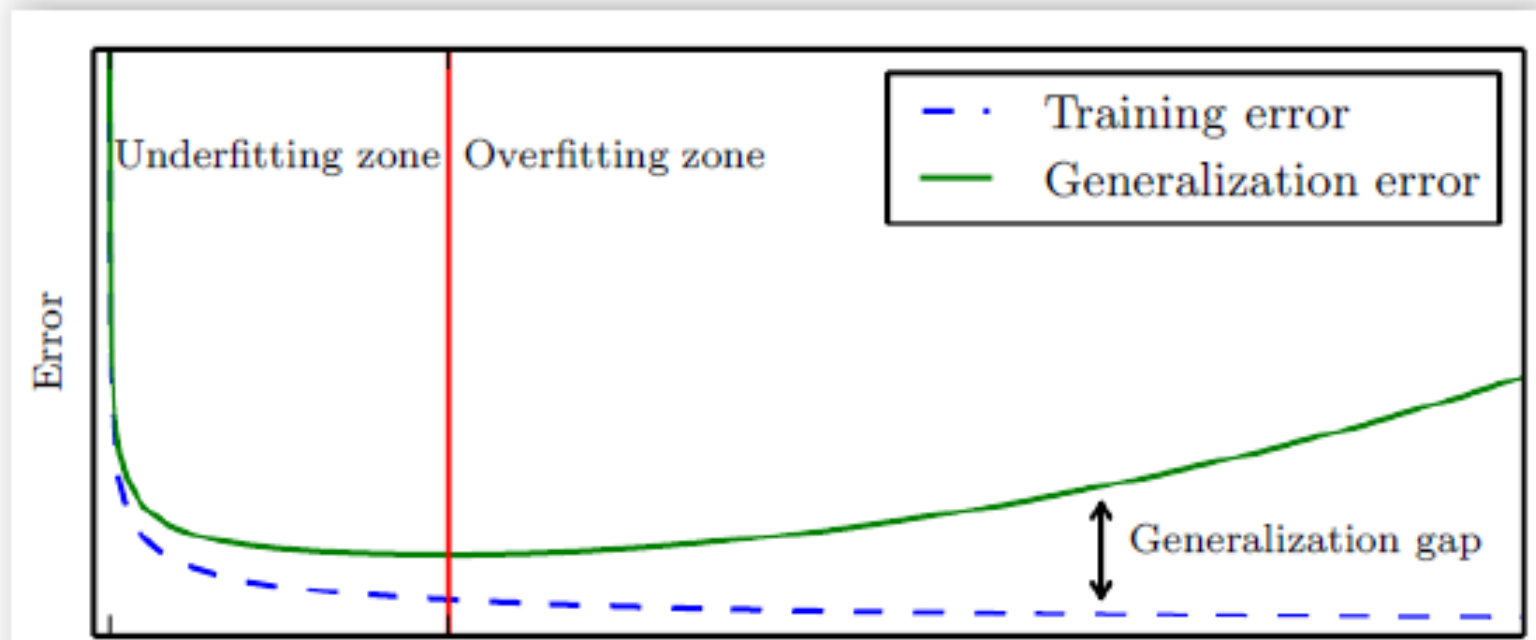
1) *Weight decay* (atténuation des poids): on ajoute une pénalité à la fonction de coût pour empêcher les poids de prendre des valeurs trop grandes. On obtient un réseau moins flexible qui se spécialise moins dans les données utilisées pour l'apprentissage. Généralement, cette pénalité concerne seulement les poids (pas les biais). La nouvelle fonction de coût s'écrit:

$$\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \alpha \|\mathbf{w}\|^2$$

2) *Dataset Augmentation*: Pour améliorer la capacité de généralisation d'un réseau, il faut l'entraîner avec plus de données. Quand on n'a pas assez de données, on peut créer des données fictives. Par exemple, pour la reconnaissance d'objets, on peut ajouter de nouvelles images en appliquant des *translations*, *rotations* ou *mises à l'échelle* aux images existantes. Il faut faire attention à ne pas changer de classe (exemple « 6 » et « 9 » en reconnaissance de chiffres). On peut aussi ajouter du bruit aux données.

3) *Early stopping* (Arrêt précoce):

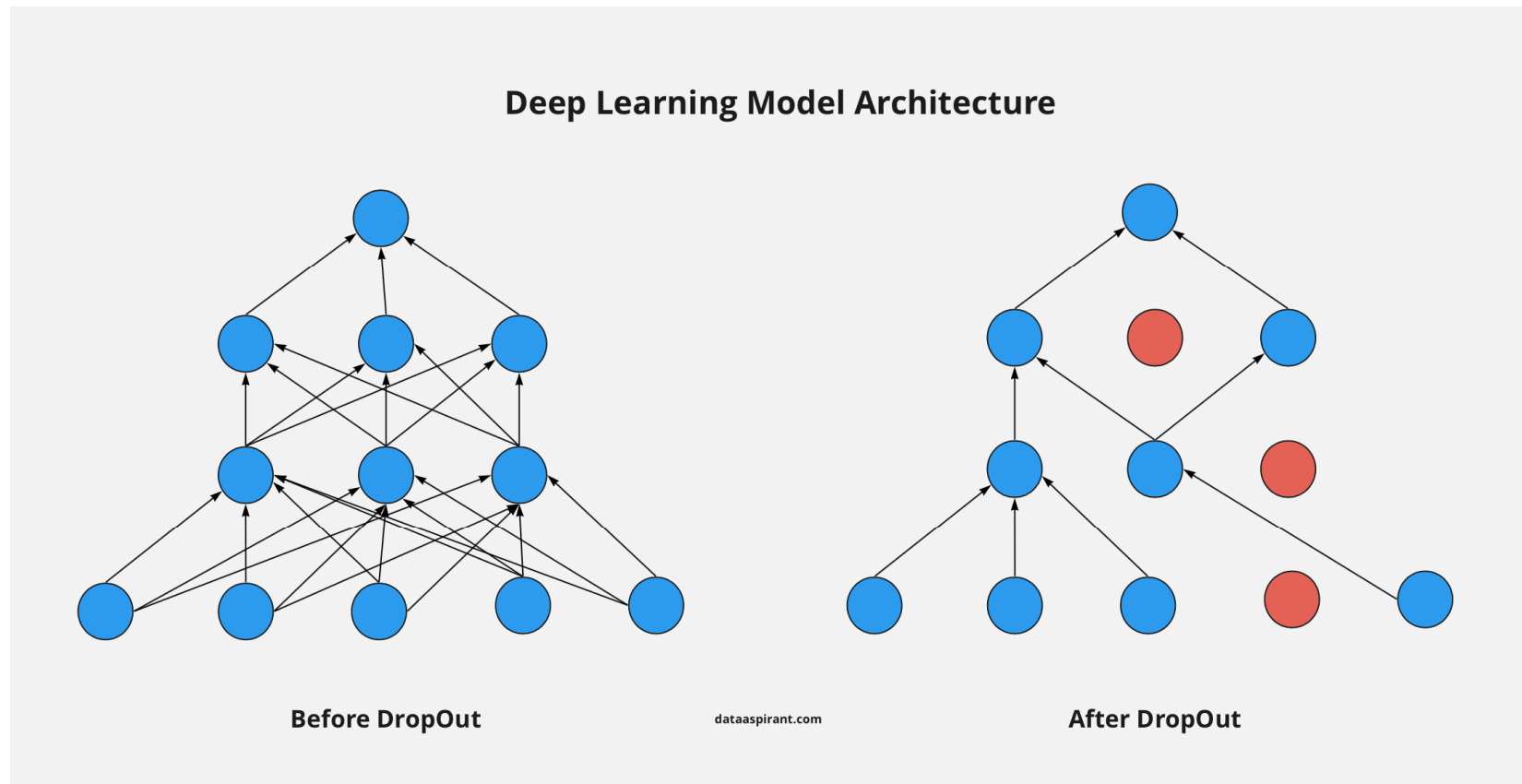
Au cours de l'apprentissage du réseau, l'erreur d'apprentissage diminue généralement de façon régulière avec le temps, mais l'erreur sur les données qui ne sont pas dans la base d'apprentissage diminue d'abord, puis réaugmente.



On peut donc obtenir un modèle avec une meilleure capacité de généralisation en revenant au réglage des paramètres au moment où l'erreur sur les données de validation est la plus faible.

4) *Dropout* (abandon):

On désactive temporairement une partie des neurones cachés et leurs connexions pendant l'apprentissage (par exemple en multipliant la sortie d'un neurone par 0). Le pourcentage des neurones désactivés, p , est le paramètre de *dropout*.



L'objectif de *dropout* est de favoriser l'extraction de caractéristiques de façon indépendante, afin d'apprendre des caractéristiques plus générales et plus diverses.

Le choix est aléatoire. A chaque itération, on applique cette désactivation aléatoire sur un nouveau sous-ensemble des neurones.

5) Bagging (bootstrap aggregating):

On entraîne séparément plusieurs modèles différents.

En phase de test, on détermine la sortie de chaque modèle, puis:

- dans un problème de classification: on applique un système de vote pour choisir la classe la plus présente en sortie des modèles,
- dans un problème de régression: on prend une moyenne des valeurs prédites par les modèles.

Les méthodes employant cette stratégie sont connues sous le nom de **méthodes d'ensemble**.

3.3. Optimisation pour l'apprentissage profond

Base de données généralement répartie en 3 jeux de données:

1) Données utilisées pour obtenir le modèle:

1.A. **Données d'apprentissage (training):** utilisées pour ajuster les paramètres du réseau.

Une *epoch* (époque) désigne un passage complet du jeu de données d'entraînement par l'algorithme. Il faut généralement plusieurs époques avant d'arrêter l'apprentissage.

1.B. **Données de validation:** utilisées pour arrêter l'apprentissage au meilleur moment (avec *early stopping*), ou pour régler les hyper-paramètres du modèle.

2) **Données de test:** utilisées pour mesurer la capacité de généralisation du modèle à la fin de l'apprentissage, et comparer des modèles.

Validation simple

On divise « les données utilisées pour obtenir le modèle » en 2 parties: données d'apprentissage et données de validation. On mesure le score du modèle sur la base de validation:

- De façon périodique (par exemple après chaque époque) si l'objectif est *early stopping*.
- Pour chaque valeur d'un hyper-paramètre si l'objectif est le choix des hyper-paramètres.

Validation croisée

Quand on n'a pas beaucoup de données, on utilise une stratégie de validation, appelée « validation croisée »:

On divise les données en K blocs: 1 pour validation et $K - 1$ pour l'apprentissage. On répète K fois cette opérations avec K choix différents du bloc de validation. La moyenne du score sur les K expériences est considérée comme le score de validation.

Optimisation basée sur le gradient (à pas fixe)

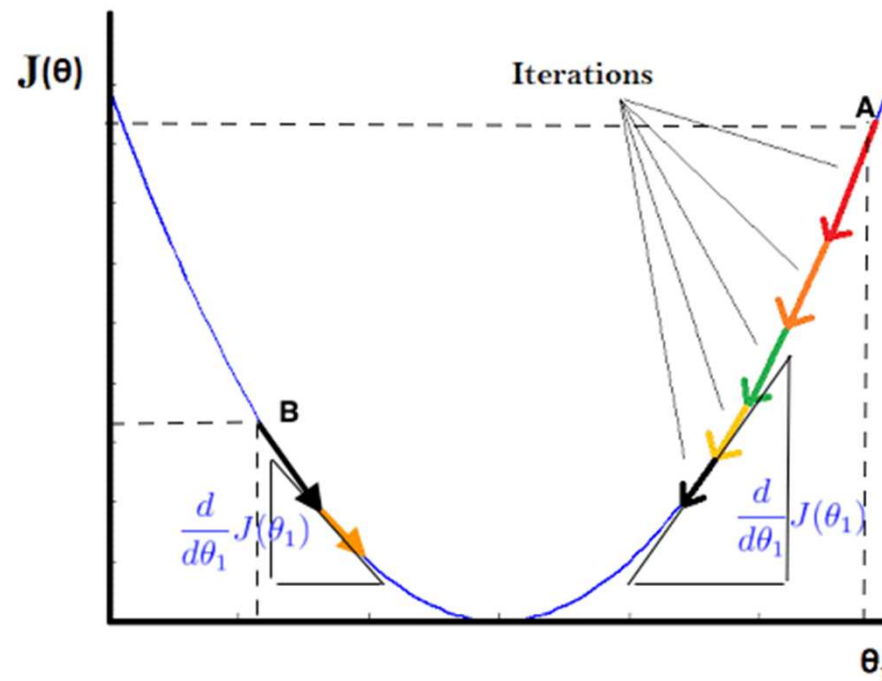
Initialiser le vecteur de paramètres du modèle, θ .

Tant que la condition d'arrêt n'est pas satisfaite

$$\theta = \theta - \varepsilon \nabla_{\theta} J(\theta)$$

Fin

$\nabla_{\theta} J(\theta)$ est le gradient (le vecteur des dérivées) de la fonction de coût J par rapport à θ , et ε est un taux d'apprentissage positif (hyper-paramètre).



Trois versions possibles:

Si $\hat{y}_i = f(\mathbf{x}_i, \boldsymbol{\theta})$, $i = 1, \dots, N$ et D représente la distance choisie:

1) Méthode du gradient *déterministe* ou *batch*

A chaque itération de l'algorithme, le gradient est calculé à partir de l'ensemble des données d'apprentissage, puis utilisé pour la mise à jour des paramètres:

$$J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N D(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}))$$

La convergence est lente mais régulière avec peu de variations.

2) Méthode du gradient *stochastique* ou *en ligne*

A chaque itération de l'algorithme, le gradient est calculé à partir d'une seule donnée → On remplace $J(\boldsymbol{\theta})$ par $D(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}))$, $i = 1, \dots, N$

3) Méthode du gradient *mini-batch*

A chaque itération de l'algorithme, le gradient est calculé à partir d'un sous-ensemble des données d'apprentissage (appelé un *mini-batch* ou parfois par abus de langage *batch*) → On remplace $J(\boldsymbol{\theta})$ par

$$\frac{1}{M} \sum_{i \in \{\text{un sous-ensemble de taille } M\}} D(y_i, f(\mathbf{x}_i, \boldsymbol{\theta}))$$

La taille de mini-batch (appelée *batch size*) est un hyperparamètre.

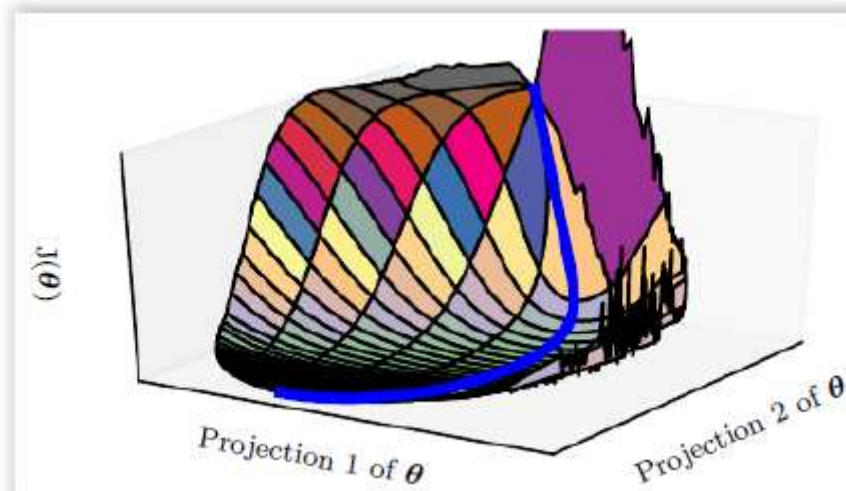
Défis dans l'optimisation des réseaux de neurones

1) Minima locaux

L'algorithme du gradient peut converger vers un minimum local au lieu du minimum global. On pense aujourd'hui que pour des réseaux de neurones suffisamment grands, la plupart des minima locaux ont une faible valeur de fonction de coût, et qu'il n'est pas important de trouver un vrai minimum global.

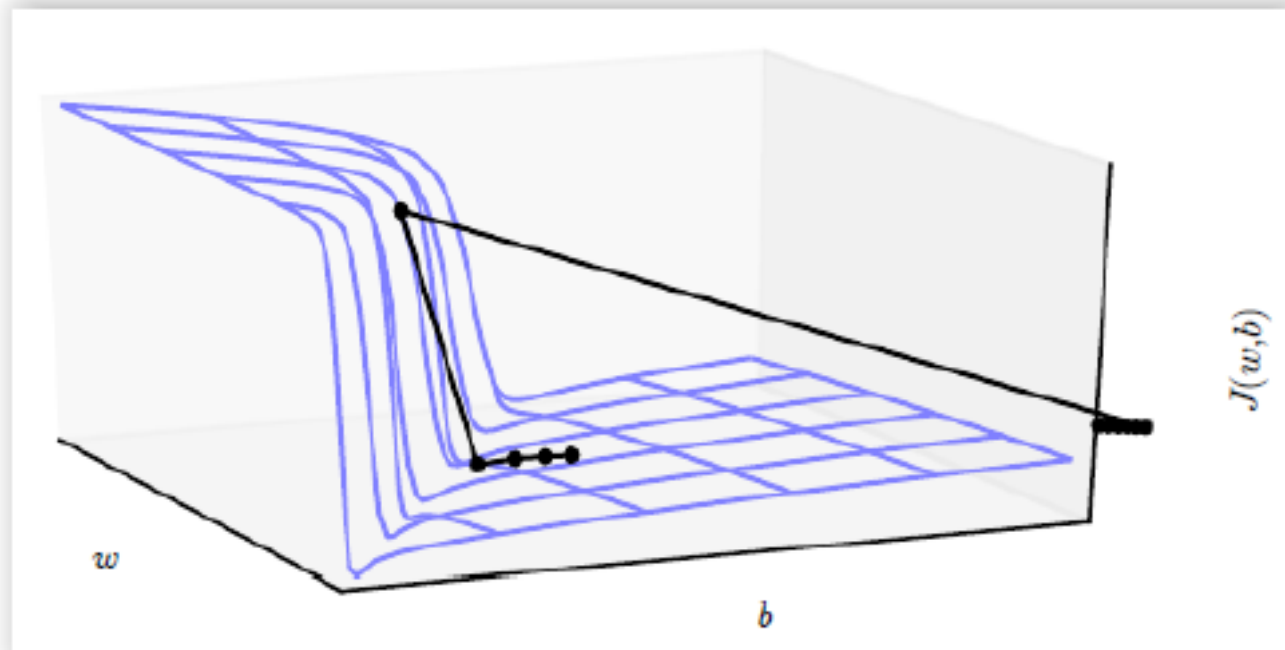
2) Plateaux et points selle

Sont plus fréquents dans les espaces de très grande dimension. Le gradient est très faible dans ces régions → une convergence lente de l'algorithme.



3) Explosion du gradient dans les « falaises »

Dans les réseaux très profonds, la fonction de coût contient souvent des régions en pente raide ressemblant à une falaise. Le gradient est très élevé dans ces régions et peut modifier largement la valeur des paramètres, ce qui peut faire perdre la plupart du travail d'optimisation qui avait été fait.



Une solution est la saturation du gradient (*gradient clipping*):

$$\text{Si } \|\mathbf{g}\| > \nu : \mathbf{g} = \frac{\mathbf{g}\nu}{\|\mathbf{g}\|}$$

4) Choix du taux d'apprentissage (ε)

Dans la version *batch*, on peut choisir une valeur constante sur toute les itérations. Dans les versions stochastique et *mini-batch*, il faut souvent réduire la valeur de ε au fil des itérations.

En pratique, il est courant de décroître le taux d'apprentissage linéairement jusqu'à l'itération τ : $\varepsilon_k = (1-\alpha)\varepsilon_0 + \alpha\varepsilon_\tau$ avec $\alpha=k/\tau$. Ensuite, on continue avec une valeur constante. Généralement, $\varepsilon_\tau=0.01\varepsilon_0$.

Si ε_0 est trop **grand**, la courbe d'apprentissage a des oscillations violentes. S'il est trop **petit**, l'apprentissage est lent.

Méthode du *Momentum*: Pour accélérer la convergence, on peut stocker en mémoire la mise-à-jour des paramètres à l'étape n ($\Delta\theta_n$). La mise à jour à l'étape suivante est calculée comme une combinaison linéaire du gradient actuel et de la mise à jour précédente :

$$\Delta\theta_{n+1} = \alpha\Delta\theta_n - \varepsilon\nabla_{\theta}J(\theta) \quad , \quad \theta_{n+1} = \theta_n + \Delta\theta_{n+1}$$

5) Initialisation des paramètres (en général)

Les biais avec des valeurs constantes choisies au préalable,

Les poids avec des valeurs aléatoires gaussiennes ou uniformes.

D'autres algorithmes d'optimisation (gradient à pas variable ou second-ordre)

1) Adagrad

Require: Global learning rate ϵ

Require: Initial parameter θ

Require: Small constant δ , perhaps 10^{-7} , for numerical stability

Initialize gradient accumulation variable $\mathbf{r} = \mathbf{0}$

while stopping criterion not met do

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

 Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

 Accumulate squared gradient: $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$

 Compute update: $\Delta \theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}$. (Division and square root applied element-wise)

 Apply update: $\theta \leftarrow \theta + \Delta \theta$

end while

2) RMSProp

Require: Global learning rate ϵ , decay rate ρ .

Require: Initial parameter θ

Require: Small constant δ , usually 10^{-6} , used to stabilize division by small numbers.

Initialize accumulation variables $r = 0$

while stopping criterion not met do

 Sample a minibatch of m examples from the training set $\{x^{(1)}, \dots, x^{(m)}\}$ with corresponding targets $y^{(i)}$.

 Compute gradient: $g \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$

 Accumulate squared gradient: $r \leftarrow \rho r + (1 - \rho) g \odot g$

 Compute parameter update: $\Delta\theta = -\frac{\epsilon}{\sqrt{\delta+r}} \odot g$. ($\frac{1}{\sqrt{\delta+r}}$ applied element-wise)

 Apply update: $\theta \leftarrow \theta + \Delta\theta$

end while

3) Adam (*Adaptive moments*)

Require: Step size ϵ (Suggested default: 0.001)

Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0,1)$.
(Suggested defaults: 0.9 and 0.999 respectively)

Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})

Require: Initial parameters θ

Initialize 1st and 2nd moment variables $\mathbf{s} = \mathbf{0}$, $\mathbf{r} = \mathbf{0}$

Initialize time step $t = 0$

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

 Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

$t \leftarrow t + 1$

 Update biased first moment estimate: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1) \mathbf{g}$

 Update biased second moment estimate: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2) \mathbf{g} \odot \mathbf{g}$

 Correct bias in first moment: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}$

 Correct bias in second moment: $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$

 Compute update: $\Delta \theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}} + \delta}}$ (operations applied element-wise)

 Apply update: $\theta \leftarrow \theta + \Delta \theta$

end while

4) Méthode de Newton

Require: Initial parameter θ_0

Require: Training set of m examples

while stopping criterion not met do

 Compute gradient: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

 Compute Hessian: $\mathbf{H} \leftarrow \frac{1}{m} \nabla_{\theta}^2 \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

 Compute Hessian inverse: \mathbf{H}^{-1}

 Compute update: $\Delta\theta = -\mathbf{H}^{-1} \mathbf{g}$

 Apply update: $\theta = \theta + \Delta\theta$

end while

5) Méthode du gradient conjugué

Require: Initial parameters θ_0

Require: Training set of m examples

Initialize $\rho_0 = 0$

Initialize $g_0 = 0$

Initialize $t = 1$

while stopping criterion not met do

Initialize the gradient $g_t = 0$

Compute gradient: $g_t \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

Compute $\beta_t = \frac{(g_t - g_{t-1})^\top g_t}{g_{t-1}^\top g_{t-1}}$ (Polak-Ribière)

(Nonlinear conjugate gradient: optionally reset β_t to zero, for example if t is a multiple of some constant k , such as $k = 5$)

Compute search direction: $\rho_t = -g_t + \beta_t \rho_{t-1}$

Perform line search to find: $\epsilon^* = \operatorname{argmin}_{\epsilon} \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \theta_t + \epsilon \rho_t), \mathbf{y}^{(i)})$

(On a truly quadratic cost function, analytically solve for ϵ^* rather than explicitly searching for it)

Apply update: $\theta_{t+1} = \theta_t + \epsilon^* \rho_t$

$t \leftarrow t + 1$

end while

Normalisation par lots (*Batch normalization*)

Une des innovations récentes les plus utilisées en deep learning, qui améliore la vitesse et la stabilité de l'algorithme du gradient.

En phase d'apprentissage, pour chaque mini-batch, les entrées d'une couche sont **centrées et réduites** en utilisant la moyenne et la variance empiriques calculées sur le mini-batch: $\mathbf{H}' = (\mathbf{H} - \mu) / \sigma$ avec $\mu = \frac{1}{M} \sum \mathbf{H}_i$, $\sigma = \sqrt{\delta + \frac{1}{M} \sum (\mathbf{H} - \mu)_i^2}$ où δ est une très petite valeur.

En phase de test, μ et σ peuvent être remplacés par des moyennes mobiles, obtenues pendant l'apprentissage. Cela permet d'évaluer le modèle sur un seul exemple.

La normalisation d'une unité peut réduire la puissance expressive du réseau. Il est donc coutume de remplacer \mathbf{H} par $\gamma \mathbf{H}' + \beta$, où γ et β sont deux paramètres à apprendre qui permettent aux activations d'avoir n'importe quelle moyenne et variance.

la moyenne est donc déterminée uniquement par β , facile à apprendre, alors que sans *batch normalization*, la moyenne est déterminée par une interaction compliquée entre les paramètres dans les couches avant \mathbf{H} .

Certains auteurs proposent d'appliquer le *Batch normalization* au signal avant la fonction d'activation, c-à-d à $\mathbf{z} = \mathbf{b} + \mathbf{W}^T \mathbf{x}$.

3.4. Réseaux de neurones convolutifs (CNN: *Convolutional Neural Networks*)

Les réseaux les plus populaires actuellement, utilisant la convolution au moins dans une couche. Composés généralement de plusieurs couches de convolution pour l'extraction automatique des caractéristiques, suivies des couches totalement connectées pour classification/régression.

Dans le cas des images, la **convolution discrète** avec une entrée I , un noyau de convolution (filtre) K et une sortie S s'écrit:

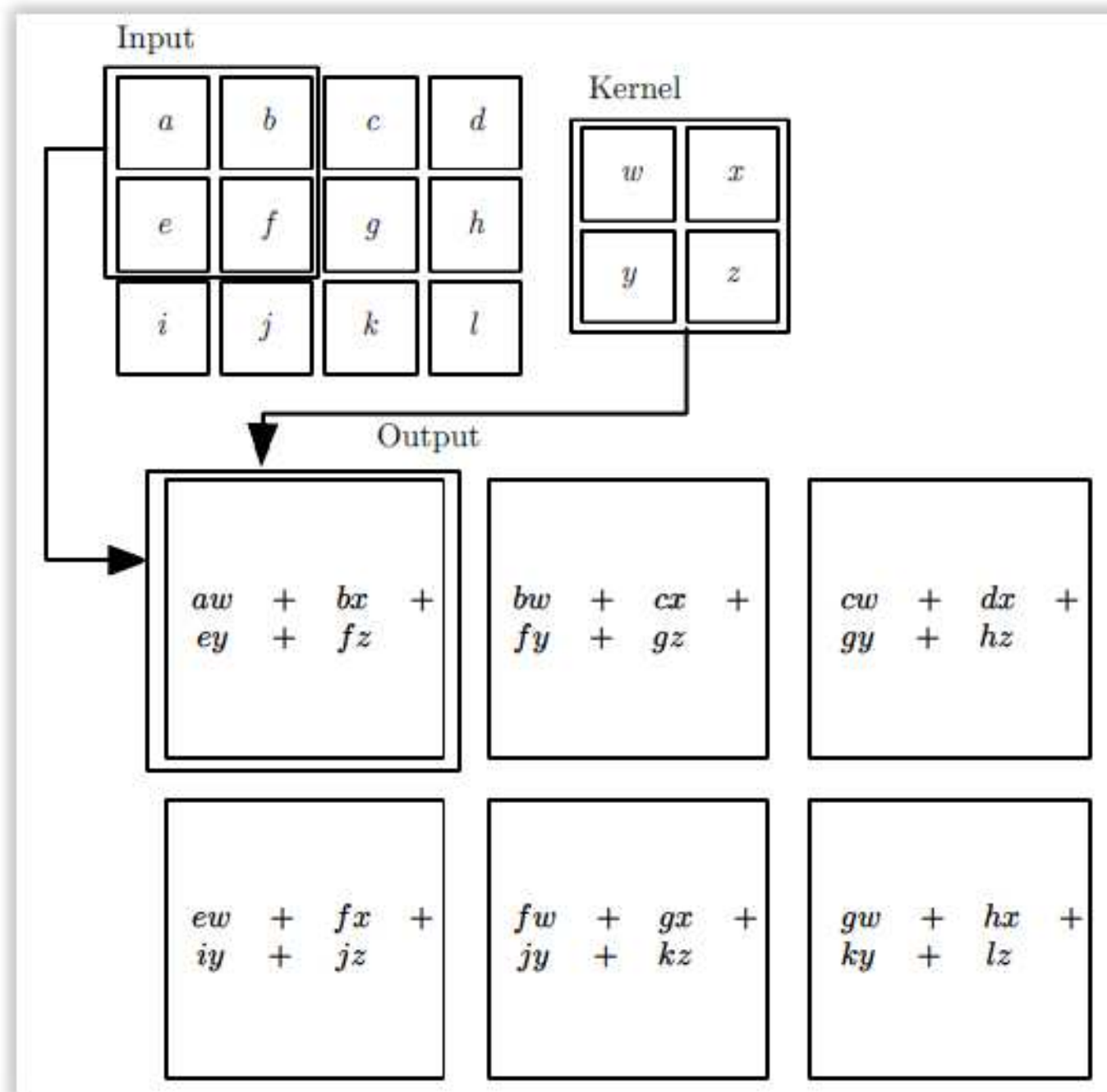
$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n) K(m, n)$$

Dans la formule classique d'une convolution, les indices du signal et du filtre varient dans le sens inverse: il en résulte la propriété de commutativité de la convolution.

La plupart des bibliothèques d'apprentissage automatique réalisent la relation suivante, qui est en réalité une « **intercorrélacion** », en l'appelant « **convolution** »:

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i + m, j + n) K(m, n)$$

La propriété de commutativité n'est plus vérifiée, mais nous n'en avons pas besoin en apprentissage automatique.



Une convolution discrète, appliquée aux signaux de taille finie, peut être modélisée comme une multiplication par une matrice de Toeplitz ou circulante. Quand le noyau de convolution est petit par rapport à l'image d'entrée, la matrice est creuse (la plupart de ses éléments sont nuls).

➔ Tout algorithme d'apprentissage automatique basé sur la multiplication matricielle peut traiter la convolution.

Les éléments des noyaux de convolution seront estimés pendant l'apprentissage du réseau.

Motivations pour l'utilisation de la convolution en deep learning

1) Connectivité parcimonieuse (*sparse*)

Dans un réseau totalement connecté (MLP classique), chaque neurone est connecté à tous les neurones de la couche précédente.

Dans un réseau convolutif, comme la taille du noyau de convolution est petite, chaque neurone est connecté à un nombre limité des neurones de la couche précédente. Ceci réduit le nombre de paramètres à stocker et le temps de calcul.

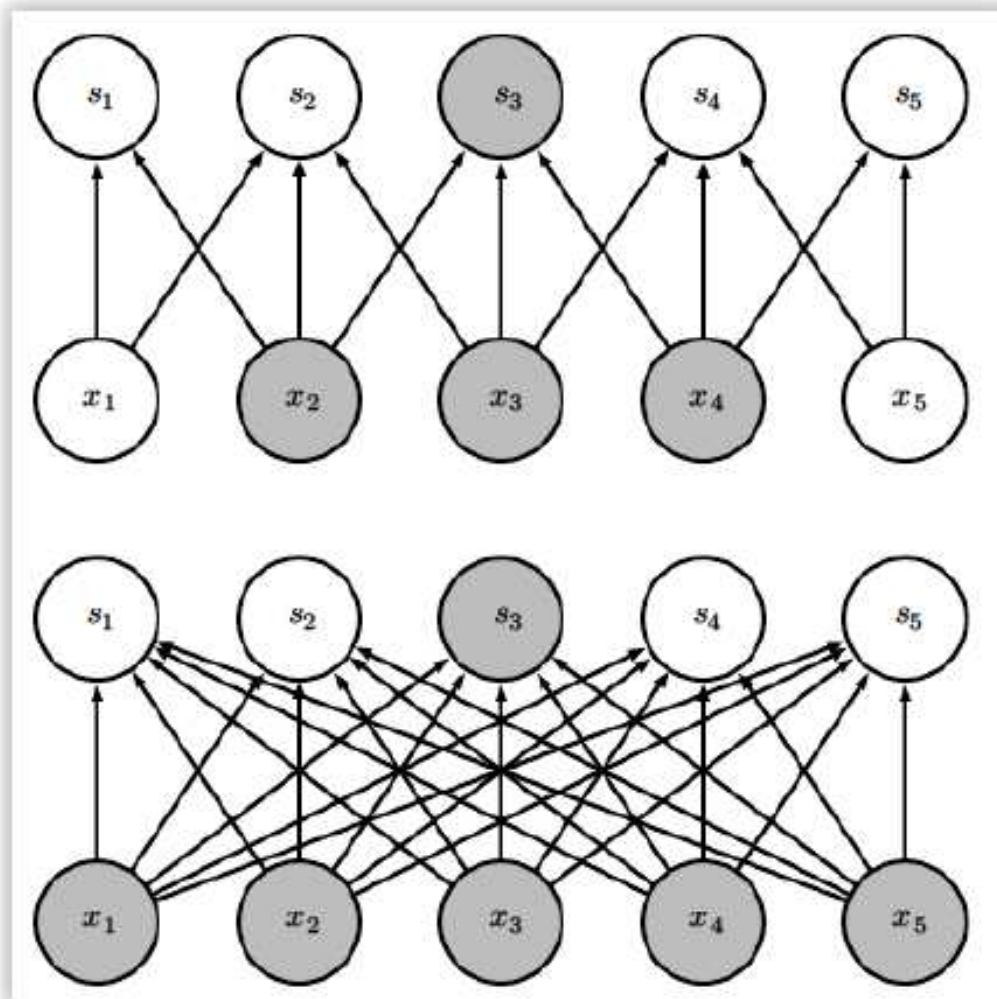


Illustration de la connectivité parcimonieuse:

En haut: s est calculé à partir d'un noyau de convolution de taille 3 dans un réseau convolutif.

En bas: s est calculé par un produit matriciel dans un réseau classique (totalement connecté).

2) Partage des paramètres

Dans un réseau traditionnel, chaque poids est utilisé exactement une fois lors du calcul de la sortie d'une couche.

Dans un réseau convolutif, chaque élément d'un noyau est appliqué à chaque élément de l'entrée → plutôt que d'apprendre un ensemble séparé de paramètres pour chaque emplacement, le réseau n'apprend qu'un seul ensemble. La convolution est donc nettement plus efficace que la multiplication matricielle dense en termes de besoins en mémoire.

3) Représentation équivariante

Grâce au partage des paramètres, les réseaux convolutifs ont la propriété d'équivariance à la translation. Si le signal ou l'image est décalé, le résultat de la convolution ne change pas: il est juste décalé.

→ L'utilisation du même noyau de convolution (filtre) sur des zones similaires d'une image permet d'extraire les mêmes caractéristiques (par exemple des contours).

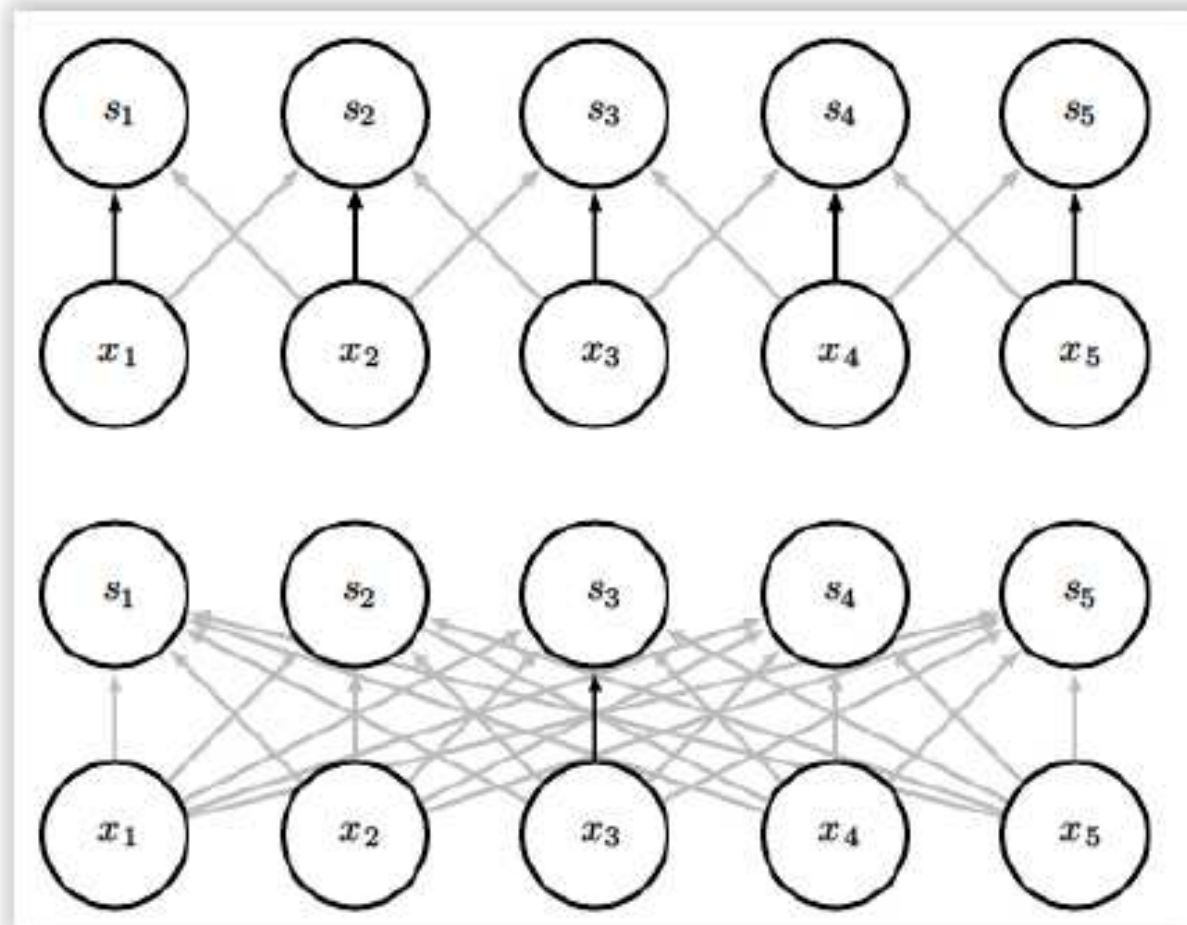


Illustration du partage des paramètres:

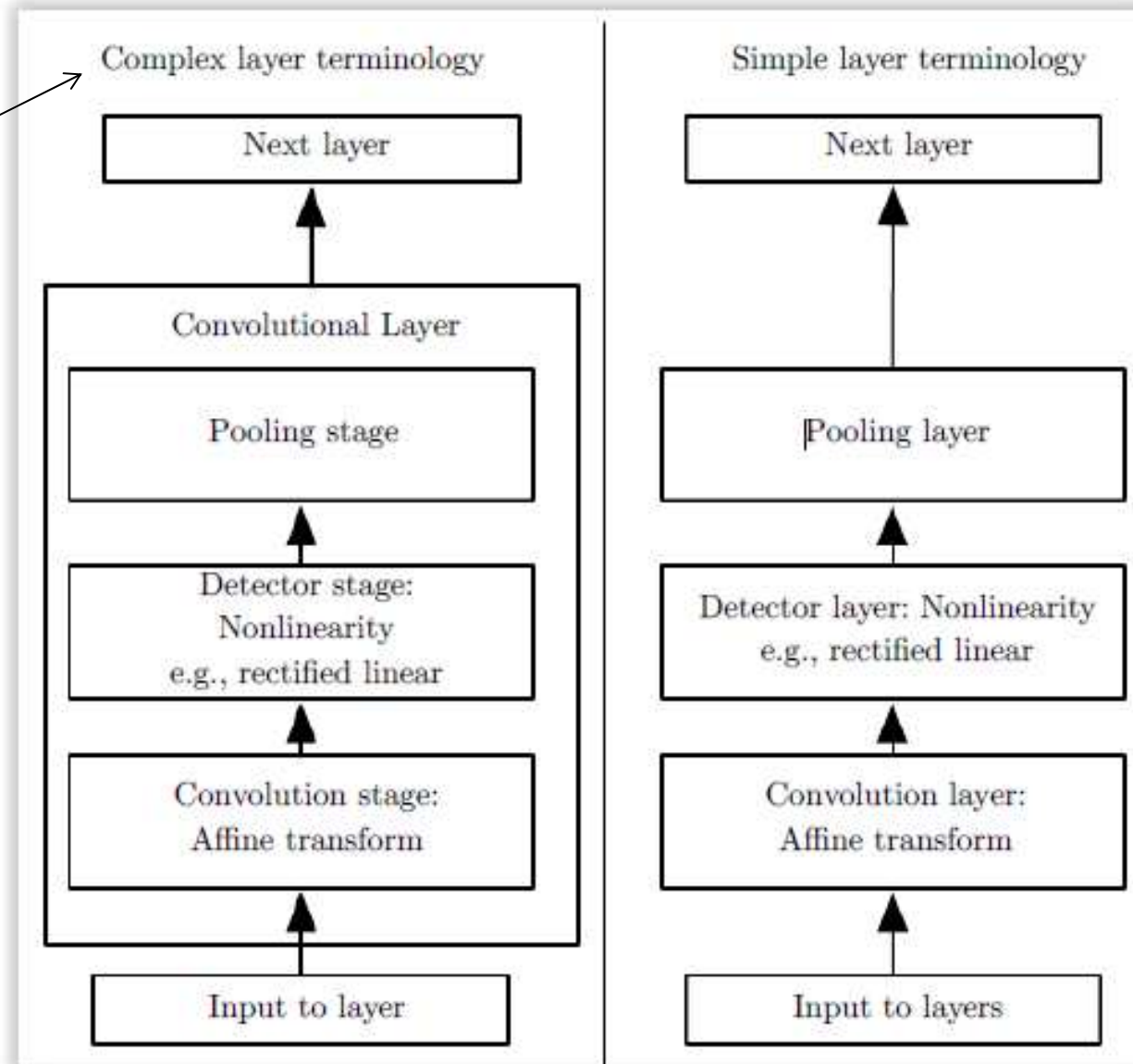
En haut: La flèche verticale est la valeur de l'élément central d'un noyau de convolution de taille 3. Elle a la même valeur pour tous les neurones.

En bas: La flèche verticale est le poids central dans un réseau traditionnel: chaque flèche verticale a une valeur différente.

Structure d'une « couche » convolutive

Deux terminologies différentes couramment utilisées:

Celle utilisée
dans ce cours



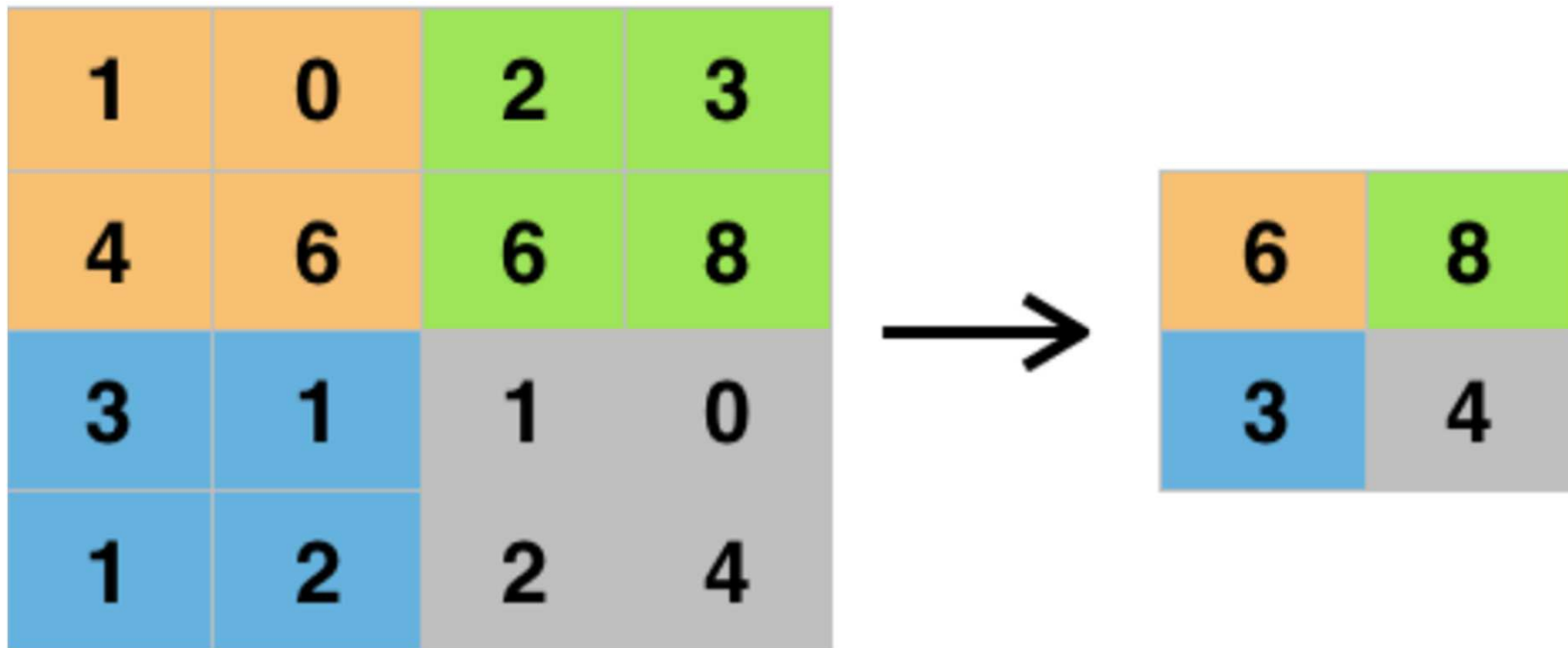
Une couche convolutive est composée de 3 étapes:

- 1) Plusieurs convolutions sont réalisées en parallèle pour produire un ensemble d'activations linéaires.
- 2) Chaque activation linéaire passe dans une fonction d'activation non linéaire (souvent ReLU). Cette étape est parfois appelée l'étape de détection.
- 3) *Pooling* (mise en commun).

Pooling

Une fonction de pooling calcule une statistique récapitulative sur une zone.

La plus utilisée est *max pooling*, qui prend la valeur maximale dans un voisinage rectangulaire. Les autres fonctions populaires sont la moyenne ou la moyenne pondérée ou la norme Euclidienne dans un voisinage rectangulaire.



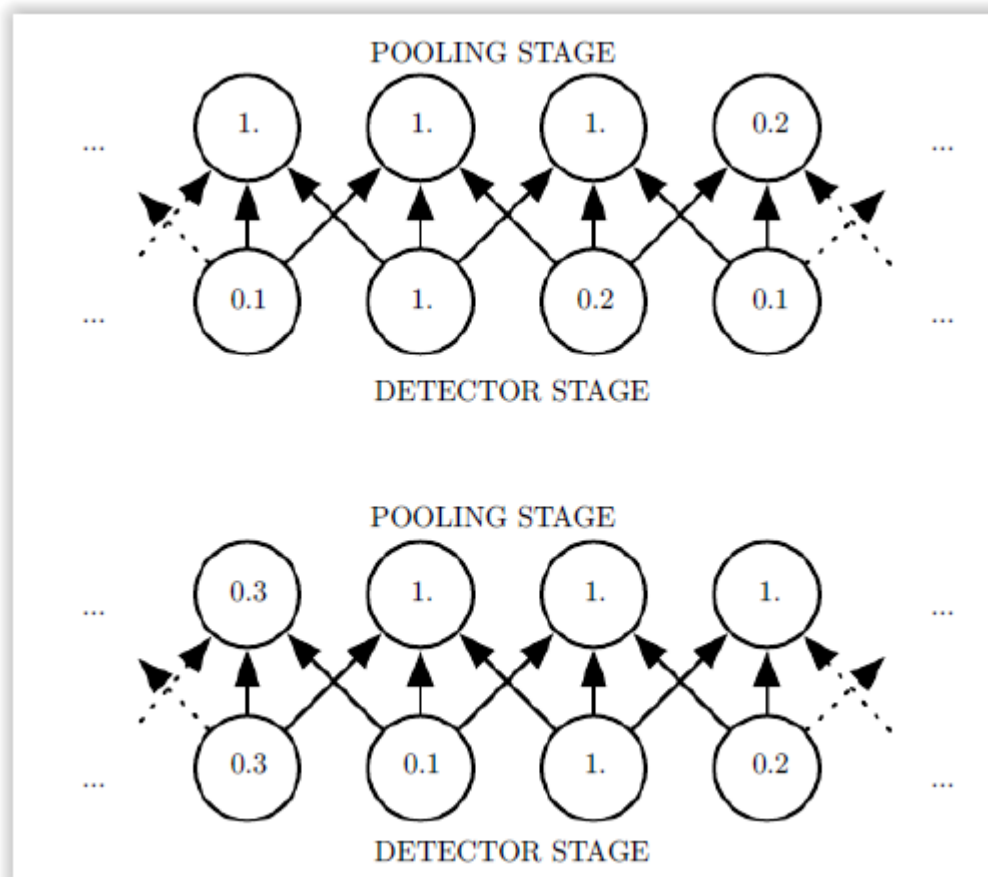
Exemple de *max pooling* avec un pas de 2 sur un rectangle 2x2

Avantages de Pooling

1) Invariance aux petites translations

Si l'entrée est un peu translatée, les valeurs de la plupart des sorties de pooling ne changent pas. Utile quand on veut savoir si une caractéristique est présente plutôt que connaître son emplacement précis.

Illustration: En bas, l'entrée est décalée d'un pixel à droite. Toutes les sorties de l'étape de détection ont changé, mais seulement la moitié des sorties de l'étape de pooling sont modifiées. C'est parce que le pooling est sensible à la « valeur » du maximum, pas à son emplacement exact.



2) Sous-échantillonnage

Les rectangles utilisés pour pooling peuvent se chevaucher ou pas. Le pas de pooling (*stride*) est un paramètre à choisir.

Si $\text{pas} = 1$: en sortie de l'étape de pooling on aura autant de valeurs qu'en entrée (voir l'exemple de la page précédente).

Si $\text{pas} > 1$: en sortie on aura moins de valeurs qu'en entrée (sous-échantillonnage) → la couche suivante aura moins d'entrées à traiter.

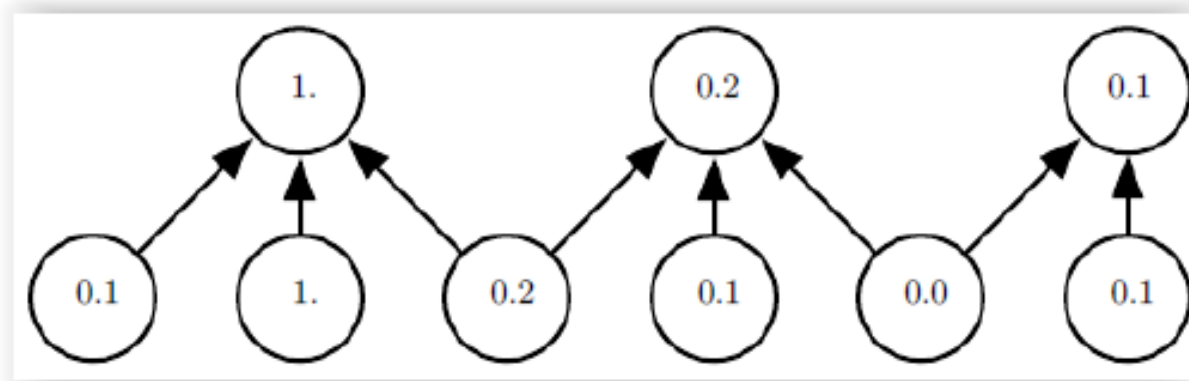


Illustration: un max pooling avec une largeur de pool de trois et une foulée entre les pools de deux. Cela réduit la taille de la représentation d'un facteur deux.

3) Traitement des entrées de taille variable

Si nous voulons classifier des images de taille variable, l'entrée de la couche de classification doit avoir une taille fixe. Ceci est généralement accompli en faisant varier la taille d'un décalage entre les régions de pooling afin que la couche de classification reçoive toujours le même nombre de statistiques récapitulatives quelle que soit la taille de l'entrée.

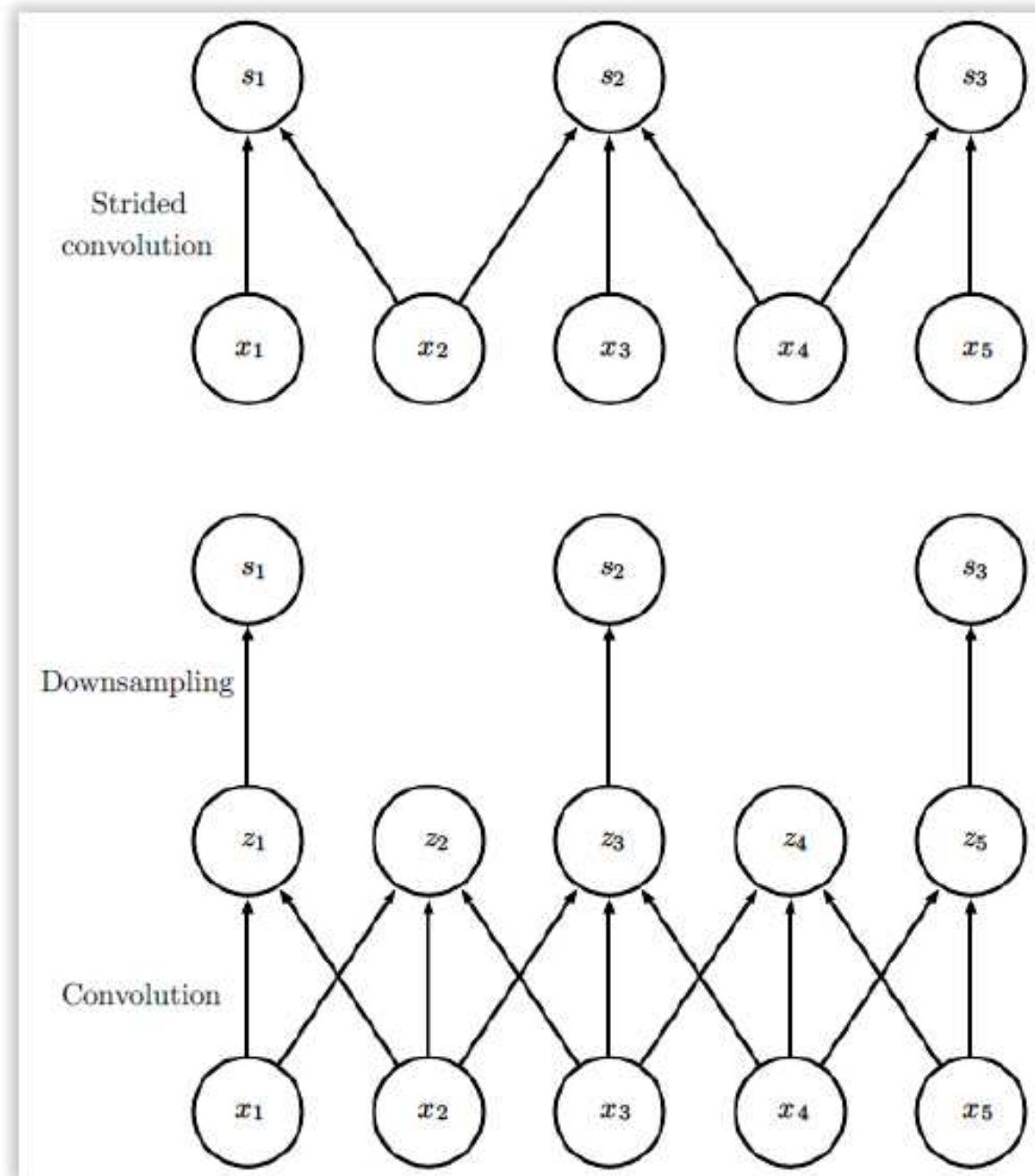
Remarques concernant la convolution

- En général, plusieurs convolutions avec des noyaux différents sont réalisées en parallèle. Ceci permet à chaque couche du réseau d'extraire plusieurs caractéristiques différentes.
- Si les données sont des images couleur, l'entrée et la sortie sont des tenseurs à 3 dimensions: il y a 2 dimensions spatiales + 1 dimension pour les 3 canaux rouge-vert-bleu.
- Pour réduire le coût de calcul, on peut calculer la convolution sur l'image avec un pas de s (appelé *stride*) dans chaque dimension spatiale. Ceci est l'équivalent au sous-échantillonnage du résultat d'une convolution complète.

Convolution
calculée avec un
pas de 2

=

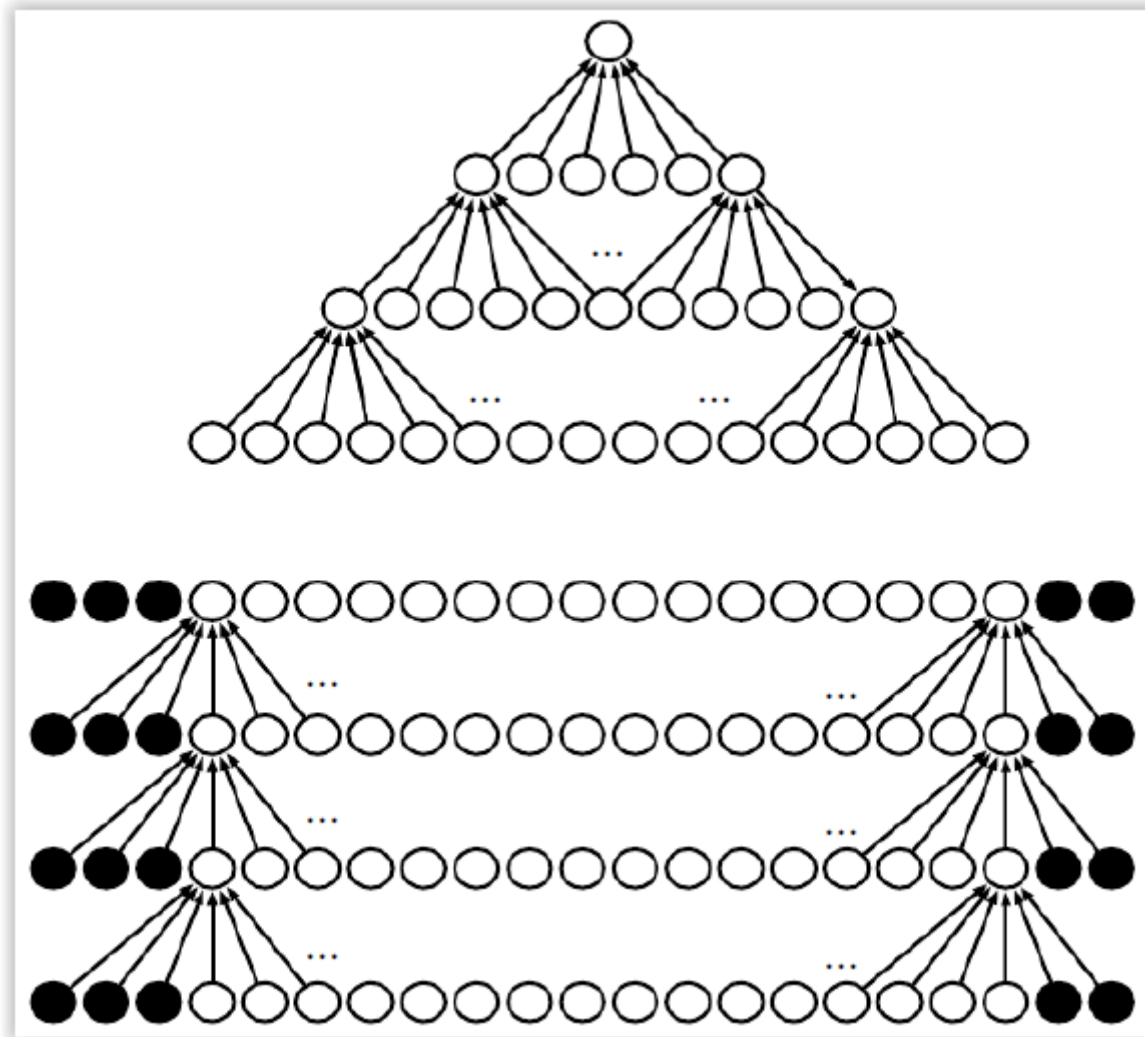
Convolution
complète suivie
d'un sous-
échantillonnage
de 2 (perte de
temps)



- Si on réalise la convolution sans *zero-padding* (appelé « *valid padding* »), la largeur de la représentation diminue d'une couche à l'autre.

Le zero-padding de l'entrée nous permet de contrôler la largeur du noyau et la taille de la sortie indépendamment. On peut utiliser 2 types de zero-padding:

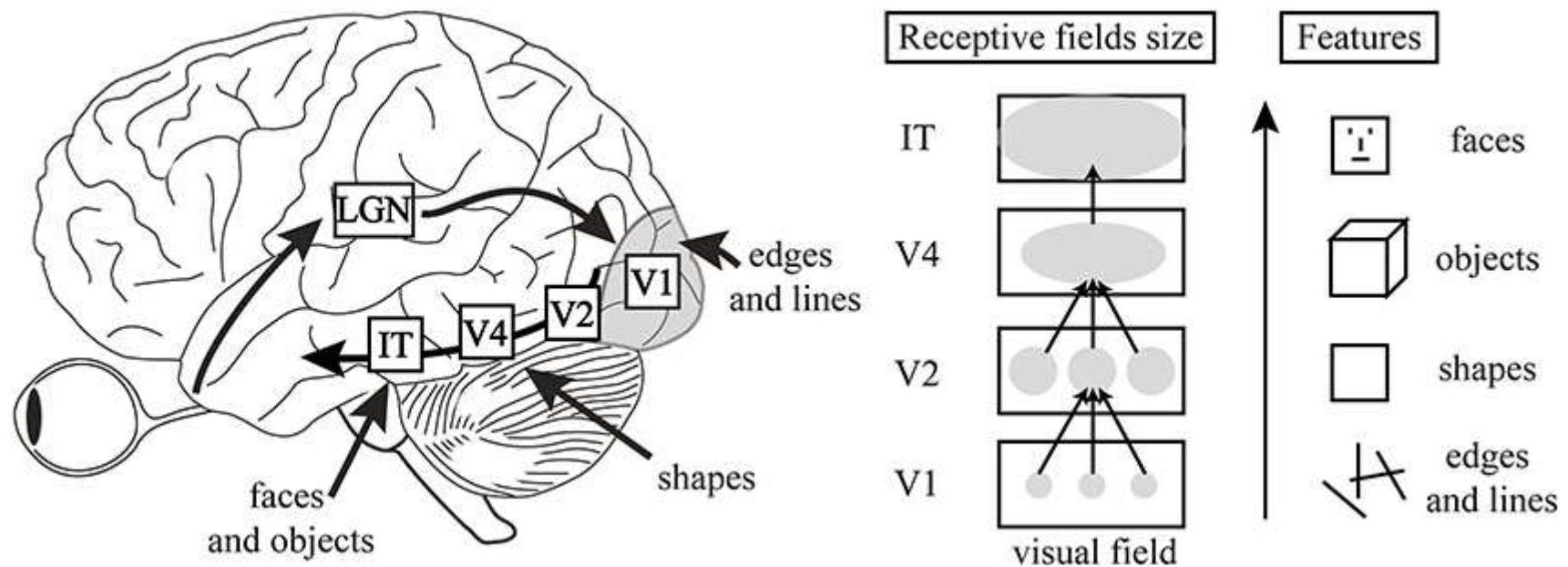
- 1) Suffisamment de zéros sont ajoutés pour que la taille de la sortie soit égale à la taille de l'entrée (appelé « *same padding* ») → les pixels d'entrée près des frontières auront moins d'impact sur la sortie que les pixels d'entrée qui sont au centre → ils seront sous-représentés dans le modèle.
- 2) Suffisamment de zéros sont ajoutés pour que chaque pixel soit visité k fois dans chaque direction, où k est la largeur du noyau (appelé « *full padding* ») → la taille de la sortie sera plus large que la taille de l'entrée → les pixels de sortie près des frontières sont créés par moins de pixels d'entrée que les pixels de sortie près du centre.



En haut: convolution sans zero-padding
En bas: convolution avec zero-padding

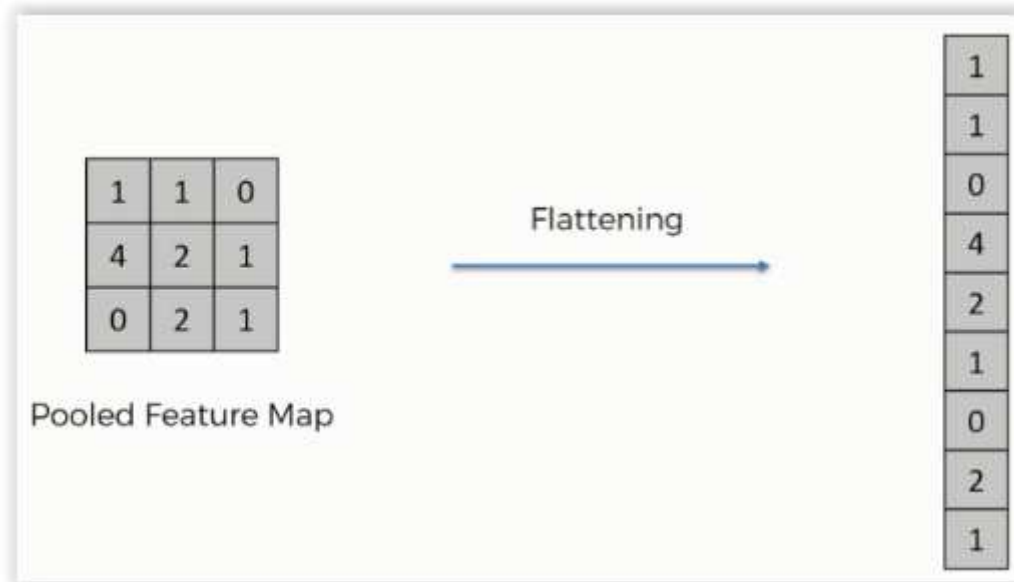
Inspiration biologique des réseaux convolutifs

Les réseaux de neurones convolutifs sont inspirés du mécanisme de fonctionnement du cortex visuel.

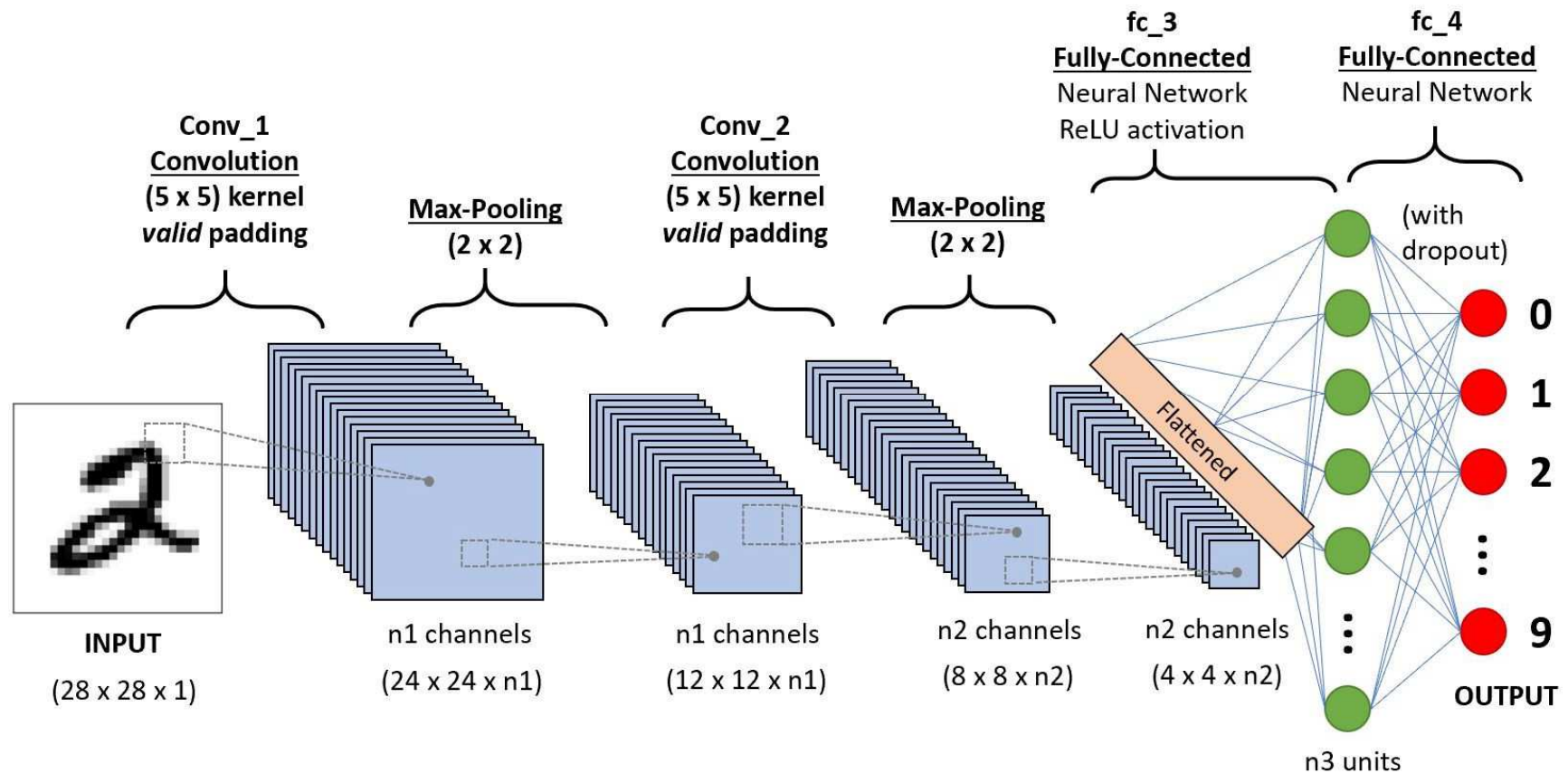


Mise à plat (*flattening*)

Dans un réseau de neurones convolutifs, les premières couches sont des couches convolutives qui se chargent de l'extraction automatique des caractéristiques sous forme de tableaux à 2 ou plusieurs dimensions. Les dernières couches sont des couches denses (*fully connected MLP*) pour classification ou régression à partir de ces caractéristiques: leur entrée est un tableau à 1 dimension. Il faut donc une opération de *flattening* entre les couches convolutives et les couches denses, pour le changement de dimensions.



Exemple d'architecture d'un réseau de neurones convolutifs

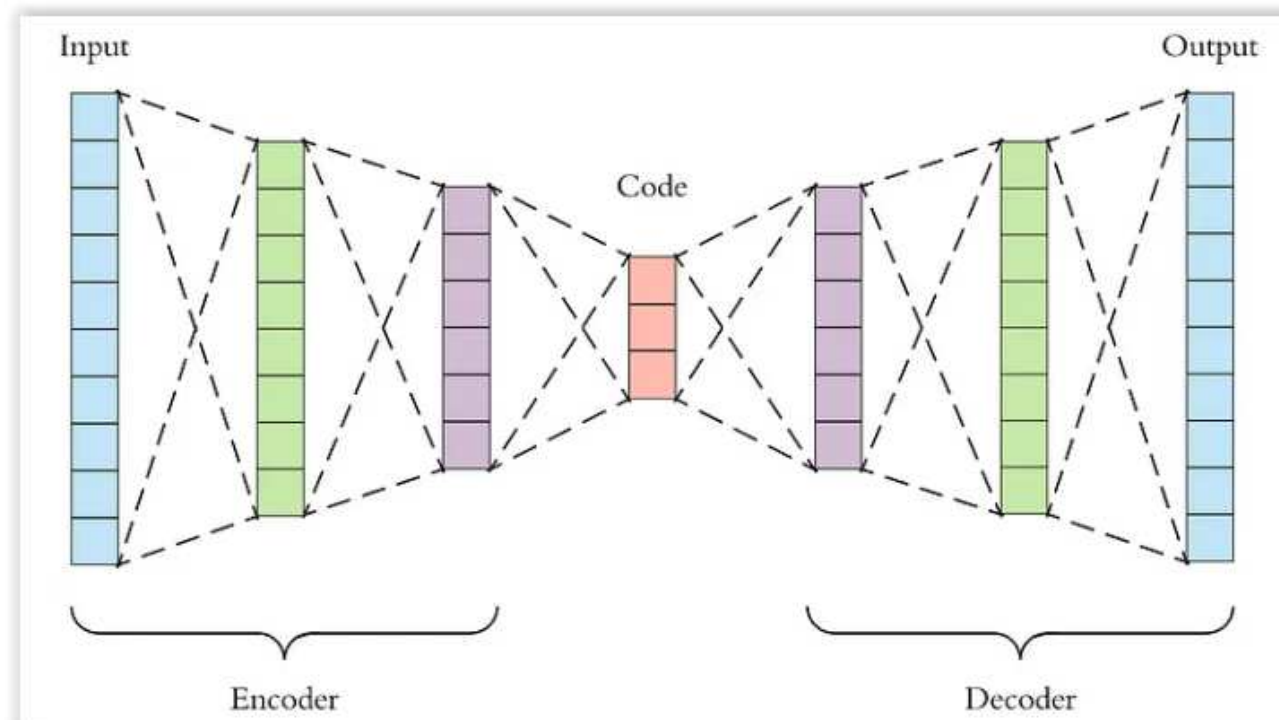


3.5. D'autres types de réseaux

1) Auto-encodeur

Un réseau principalement utilisé pour l'extraction de caractéristiques et la réduction de dimension, mais aussi pour le débruitage.

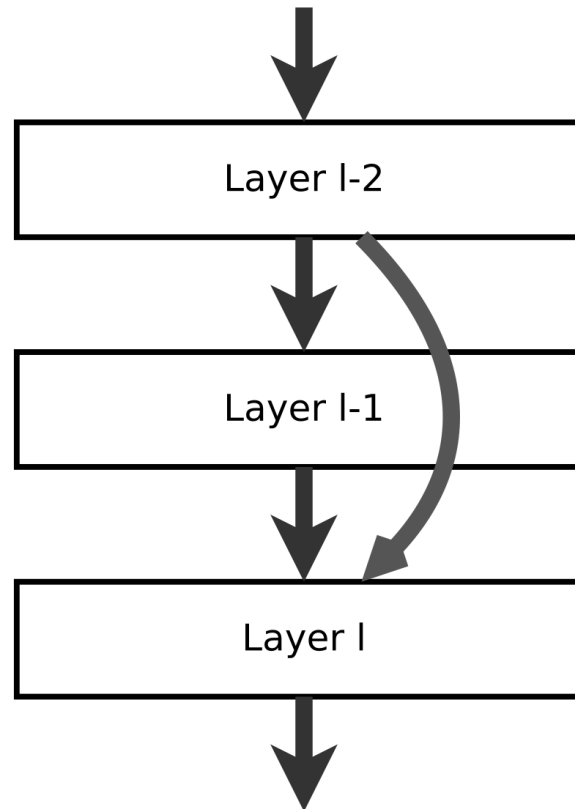
La couche de sortie a le même nombre de neurones que la couche d'entrée. Les couches cachées ont moins de neurones. Le réseau est souvent entraîné pour que $\text{sortie} = \text{entrée} = X$. La couche cachée la plus interne produit alors des caractéristiques qui représentent X dans un espace avec moins de dimensions (appelées « code »).



2) Réseau résiduel (*Residual neural network* : ResNet)

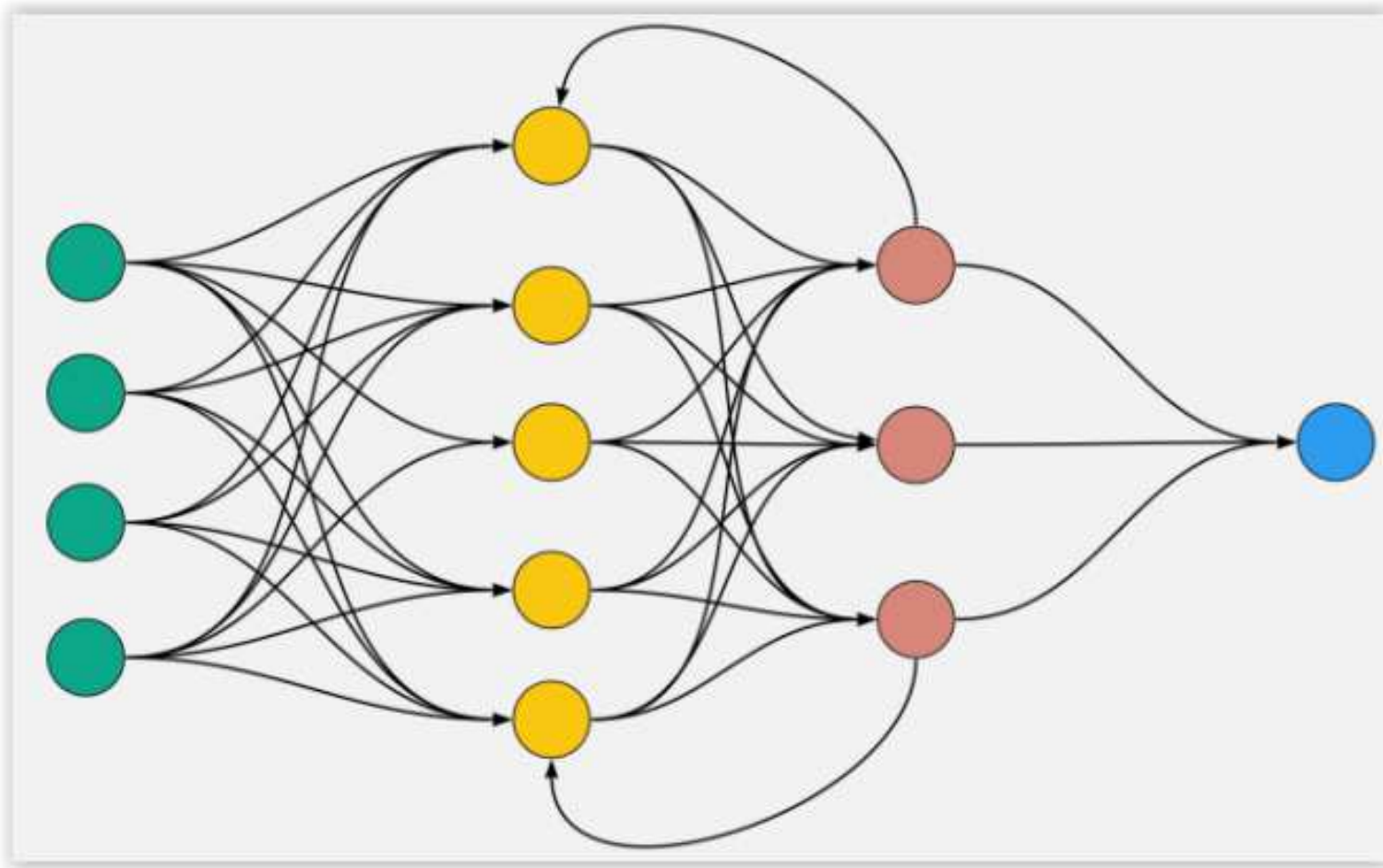
Idée: permettre aux couches antérieures d'être directement introduites dans les couches plus profondes, en utilisant ce qu'on appelle connexions de saut (*skip connections*).

Objectif: minimiser le risque de disparition ou d'explosion des gradients.



3) Réseau récurrent (*Recurrent neural network: RNN*)

Un réseau qui possède des connexions récurrentes. Utilisé pour traiter des séquences de taille variable, comme des séries temporelles.



Apprentissage par transfert (*Transfer learning*)

Exploiter les connaissances acquises pendant le traitement d'un problème pour traiter un autre problème.

Dans le contexte de l'apprentissage profond, au lieu de commencer l'apprentissage d'un modèle à partir de zéro, on prend un réseau (ou une partie d'un réseau) pré-entraîné sur un problème, et on continue son apprentissage sur un nouveau problème.

Deux approches:

- 1) Utilisation de modèles pré-entraînés comme extracteurs de caractéristiques. Par exemple, utiliser seulement les premières couches convolutives d'un réseau entraîné pour la classification des voitures, y ajouter d'autres couches, et l'entraîner pour la classification des camions.
- 2) Ajustement de modèles pré-entraînés: certains paramètres de l'ancien réseau sont gelés alors que d'autres sont entraînés.

Chapitre 4: Outils informatiques pour l'apprentissage profond

TensorFlow

Un outil d'apprentissage automatique basé sur le calcul tensoriel, développé par Google:

- Open source
- Fonctionne sur les CPU/GPU/TPU
- Contient des modèles pré-entraînés
- Supporte les architectures les plus utilisées telles que CNN et RNN
- Possède ses propres outils de la dérivation automatique (*auto-differentiation*)
- Fonctionne avec Python, C++, Java, R, et Go.

Ses concurrents: PyTorch (Meta), Caffe (Berkeley puis Facebook), MXNet (Apache et Amazon), ...

Keras: Une API (Interface de programmation d'application) qui fonctionne comme une interface python conviviale pour la bibliothèque TensorFlow.

Après l'installation de TensorFlow (Keras sera automatiquement installé avec), pour pouvoir les utiliser dans un programme python, il faut les importer:

```
import tensorflow as tf
from tensorflow import keras
```

On peut aussi importer uniquement quelques modules de Keras dont on a besoin. Par exemple:

```
from tensorflow.keras import datasets, layers, models,
    regularizers, optimizers
```

Créer un modèle: Dans le cas le plus simple, on peut utiliser la classe `sequential` de Keras qui permet d'y ajouter ensuite une succession de couches.

Création du modèle et l'ajout des couches convolutives

Exemple:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), padding='same',
                        input_shape=(28, 28, 1), activation='relu'))
model.add(layers.MaxPooling2D(pool_size=(2, 2),
                               strides=(2, 2)))
```

On crée un modèle séquentiel, appelé `model`.

On y ajoute ensuite une couche de convolution avec:

- Entrée: une image 28x28 pixels sur un canal
- Etape de convolution: 32 noyaux de convolution de taille 3x3 (→ 32 canaux en sortie), avec zero-padding (option 'same')
- Etape de détection: Fonction d'activation ReLU appliquée aux résultats de convolution
- Etape de pooling: Maxpooling sur un rectangle de 2x2 avec un pas de 2 dans chaque direction (→ sous-échantillonnage avec un facteur de 2)

Ajout d'une étape de *flatten* et des couches denses avec *dropout*

Exemple:

```
Classes=10
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(classes, activation='softmax'))
```

Après la conversion des données en tableau 1-D avec la fonction `Flatten`, on crée une couche totalement connectée (dense) avec 512 neurones et la fonction d'activation `relu` et un taux de *dropout* de 50%.

On ajoute une deuxième couche dense (couche de classification en sortie) avec 10 classes et la fonction d'activation `softmax`.

Batch normalization: Peut être réalisée entre 2 couches. Exemple:

```
model.add(layers.Conv2D(64, (3, 3), padding='same',
                        activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D(pool_size=(2, 2)))
```

Compilation du modèle

Après avoir défini l'architecture du réseau, il faut le configurer avec le module `compile`.

Exemple:

```
model.compile(optimizer='SGD',  
              loss='categorical_crossentropy',  
              metrics=['accuracy'])
```

`optimizer` définit la méthode d'optimisation. Quelques choix: SGD (méthode du gradient avec momentum), Adagrad, RMSprop, Adam, ...

`loss` définit la fonction de coût à minimiser. Quelques choix: `binary_crossentropy`, `categorical_crossentropy`, `mean_squared_error`, `mean_absolute_error`, `huber`, ...

`metrics` définit le critère utilisé pour juger les performances du modèle. Quelques choix: `accuracy`, `Precision`, `Recall`, `TruePositives`, `TrueNegatives`, `FalsePositives`, `FalseNegatives`, `AUC`, ...

Apprentissage du modèle

Après avoir configuré le modèle, il faut réaliser l'apprentissage avec une base de données labélisée en utilisant le module `fit`.

Exemple:

```
model.fit(X_train, Y_train, batch_size=128, epochs=200,  
          verbose=1, validation_split=0.2)
```

`X_train` et `Y_train` sont les données utilisées pour estimer le modèle.

`batch_size` définit la taille de mini-batch utilisé pour l'apprentissage.

`epochs` définit le nombre de fois où l'ensemble des données d'apprentissage sont présentées au réseau.

`verbose` contrôle l'affichage pendant l'apprentissage. Si `=0` aucun affichage, si `=1` affiche une barre qui progresse, si `=2` affiche le numéro d'époque.

`validation_split` définit la proportion de données utilisées pour la validation.

Evaluation du modèle

Après avoir entraîné le modèle, on évalue sa capacité de généralisation avec les données de test et le module `evaluate`.

Exemple:

```
test_loss, test_acc = model.evaluate(X_test, Y_test)
print('Test accuracy:', test_acc)
```

Prédiction avec un modèle

On peut finalement utiliser notre modèle pour faire la prédiction à partir des nouvelles entrées.

Exemple:

```
predictions = model.predict(samples_to_predict)
```

* Ce lien fournit une liste des bases de données disponibles de Keras:

https://www.tensorflow.org/api_docs/python/tf/keras/datasets

Exemple d'une base de données des chiffres manuscrites:

```
mnist = keras.datasets.mnist
(X_train, Y_train), (X_test, Y_test)=mnist.load_data()
```

Data augmentation

Pour augmenter la taille de notre base de données, on peut appliquer des transformations aux données existantes.

Exemple:

```
from tensorflow.keras.preprocessing.image import  
    ImageDataGenerator  
  
datagen = ImageDataGenerator(  
    rotation_range=30,  
    width_shift_range=0.2,  
    height_shift_range=0.2,  
    horizontal_flip=True,  
)  
  
datagen.fit(x_train)
```

Sauvegarder un modèle et ses paramètres

```
model_json = model.to_json()
with open('model.json', 'w') as json_file:
    json_file.write(model_json)
    model.save_weights('model.h5')
```

Charger un modèle et ses paramètres enregistrés

```
from tensorflow.keras.models import model_from_json
model_architecture = 'model.json'
model_weights = 'model.h5'
model =
    model_from_json(open(model_architecture).read())
model.load_weights(model_weights)
```

Quelques autres fonctionnalités utiles

`Y_train = tf.keras.utils.to_categorical(Y_train, NB_CLASSES)`
convertit les données entières en données binaires codées en format 1 parmi n (*one hot*). La sortie est une matrice dont chaque colonne correspond à une classe.

`model.summary()` montre un résumé des caractéristiques du modèle.

TensorBoard est un outil de visualisation de Tensorflow
Callbacks permet de réaliser une action de façon régulière (par exemple après chaque Epoch ou chaque Batch). Par exemple:

- Ecrire le journal de TensorBoard pour surveiller les métriques
- Enregistrer périodiquement le modèle sur le disque
- Early stopping

Exemple:

`Callbacks=`

```
[tf.keras.callbacks.TensorBoard(log_dir='./logs')]
```

```
model.fit(..., ..., callbacks=callbacks)
```

écrit le journal de TensorBoard dans le dossier './logs'

Apprentissage profond avec GPU et TPU

GPU (Graphics Processing Unit): est une unité de calcul, principalement utilisée dans les cartes graphiques pour le traitement d'images et vidéos.

- Un GPU est doté de centaines ou de milliers de cœurs dédiés simultanément à une tâche unique.
 - La plupart des opérations mathématiques utilisées en deep learning sont facilement parallélisables.
- ➔ L'utilisation des GPU permet d'accélérer considérablement l'apprentissage des réseaux de neurones.

CUDA est une interface de programmation qui permet aux logiciels d'utiliser certains GPU de Nvidia pour le traitement à usage général.
Tensorflow utilise CUDA pour exécuter ses fonctions sur GPU.

TPU (Tensor Processing Unit): est un circuit intégré, spécialement conçu pour Tensorflow par google, afin accélérer l'apprentissage profond.

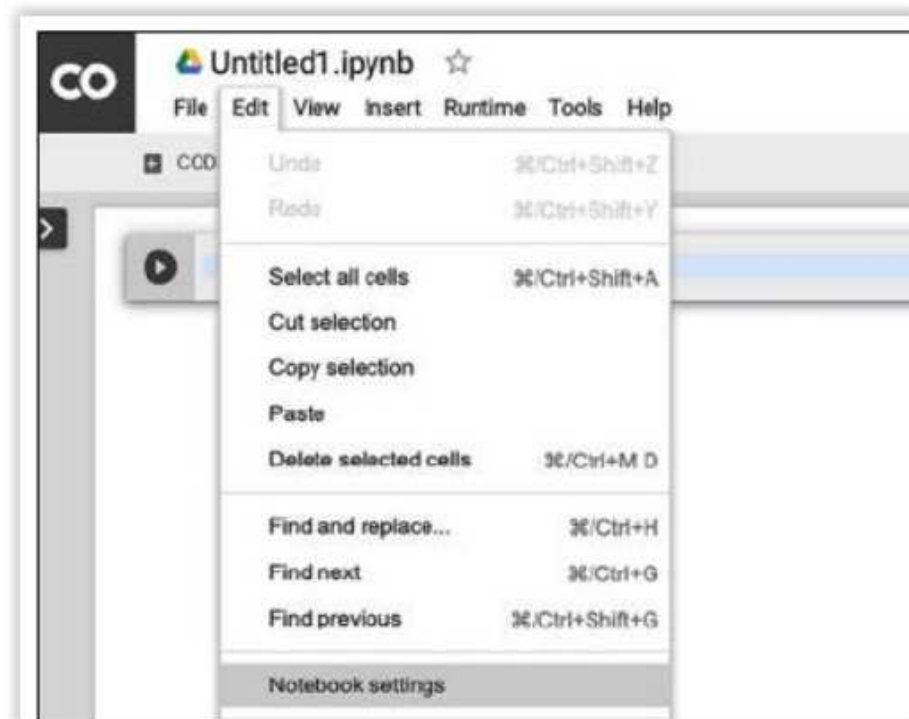
Google Colab

un service cloud, basé sur Jupyter Notebook et offert gratuitement par Google qui permet d'entraîner des modèles de machine learning directement dans le cloud.

Après la connexion à un compte google (gmail), on peut créer un nouveau notebook.

A partir de Edit->Notebook settings, on peut choisir le type du processeur utilisé: CPU, GPU ou TPU.

Il faut ensuite insérer son code python dans le notebook et l'exécuter.



Principales références

- [1] W.-M. Lee, Python Machine Learning, Wiley, 2019.
- [2] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT, 2016.
- [3] A. Kapoor, A. Gulli, S. Pal, Deep Learning with TensorFlow and Keras, Packet Publishing, 3rd Edition, 2022.
- [4] S. W. Knox, Machine Learning : a Concise Introduction, Wiley, 2018.
- [5] X. S. Yang, Optimization Techniques and Applications with Examples, Wiley, 2018.
- [6] <https://keras.io/api/>