

# Principes du ML & Régression Linéaire

Jour 1 — Après-midi

Julien Rolland

Formation M2 Développement Fullstack

Jour 1

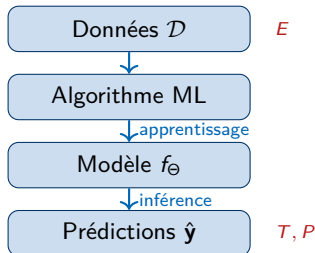
- 1 Vocabulaire du Machine Learning
- 2 Régression Linéaire
- 3 Descente de Gradient

# Qu'est-ce que le Machine Learning ?

## Définition (Tom Mitchell, 1997)

Pour une **tâche**  $T$ , un algorithme **apprend** si sa **performance**  $P$  sur  $T$  s'améliore avec l'**expérience**  $E$ .

	Concept	En pratique
$T$	Tâche	Prédire un prix
$P$	Performance	Erreur MSE
$E$	Expérience	Dataset d'entraînement



## Paradigme clé

On ne **programme** plus les règles.

On **montre des exemples** — le modèle induit les règles.

### Supervisé



$(x_i, y_i)$

Données **étiquetées**  
*Régression, Classification*

### Non-supervisé



$x_i$  seulement

Données **sans labels**  
*Clustering, Embeddings*

### Renforcement



Apprentissage par **récompense**  
*Jeux, Robotique*

### Ce cours

Focus sur l'**apprentissage supervisé** (J1–J4) et les outils de mise en production (J5). Le non-supervisé apparaît en J4 (embeddings).

**Dataset**  $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$

- $\mathbf{x}_i \in \mathbb{R}^D$  — vecteur de **features**
- $y_i \in \mathbb{R}$  (régression) ou  $\{0, 1\}$  (classification)  
⇒ le **label** (ground truth)
- $\mathbf{X} \in \mathbb{R}^{N \times D}$  — matrice de données

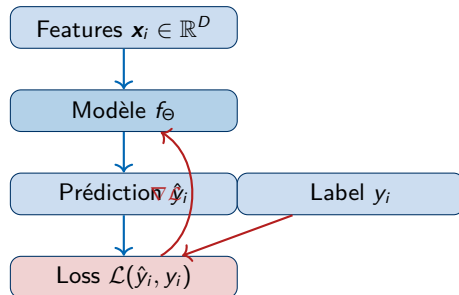
**Modèle paramétrique**  $f_{\Theta}$

$$\hat{y}_i = f_{\Theta}(\mathbf{x}_i)$$

$\Theta$  = paramètres **appris** pendant l'entraînement.

**Apprentissage** = trouver  $\Theta^*$  qui minimise l'écart entre  $\hat{y}_i$  et  $y_i$  :

$$\Theta^* = \arg \min_{\Theta} \mathcal{L}(\Theta, \mathbf{X}, \mathbf{y})$$



**Tâche** : prédire une sortie **continue**  $y \in \mathbb{R}$ .

**Modèle linéaire** ( $D = 1$ ) :

$$f_{w,b}(x) = wx + b$$

**Modèle linéaire** (multivarié) :

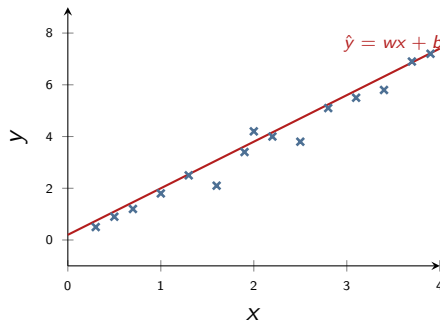
$$f_{w,b}(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b = \sum_{j=1}^D w_j x_j + b$$

**Modèle polynomial** (feature map  $\Phi$ ) :

$$f_{\Theta}(\mathbf{x}) = \boldsymbol{\theta}^\top \Phi(\mathbf{x})$$

Ex :  $\Phi(x) = [1, x, x^2, x^3]$  pour degré 3.

Données et modèle linéaire



### Mean Squared Error (MSE) — aussi appelée Least Squares

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^N (\hat{y}_i - y_i)^2 = \frac{1}{N} \sum_{i=1}^N (f_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2$$

#### Pourquoi le carré ?

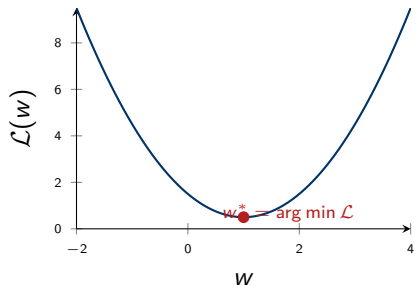
- Pénalise plus fortement les **grandes erreurs**
- Différentiable partout  $\Rightarrow$  gradient bien défini
- Lien avec la vraisemblance gaussienne (vu ce matin)

#### Forme matricielle (plus efficace) :

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$$

avec  $\mathbf{X} \in \mathbb{R}^{N \times D}$ ,  $\mathbf{w} \in \mathbb{R}^D$ ,  $\mathbf{y} \in \mathbb{R}^N$ .

Loss en 1D — convexe !



On dérive  $\mathcal{L}(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^N (\mathbf{w}^\top \mathbf{x}_i - y_i)^2$  par rapport à  $w_j$  :

Règle de la chaîne :

$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{1}{N} \sum_{i=1}^N \underbrace{2(\hat{y}_i - y_i)}_{\text{erreur}} \cdot \underbrace{\frac{\partial \hat{y}_i}{\partial w_j}}_{=x_{i,j}}$$

$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{2}{N} \sum_{i=1}^N (\hat{y}_i - y_i) x_{i,j}$$

Forme vectorielle (tous les  $w_j$  d'un coup) :

$$\nabla_{\mathbf{w}} \mathcal{L} = \frac{2}{N} \mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y})$$

- $\mathbf{X}\mathbf{w} - \mathbf{y} \in \mathbb{R}^N$  : vecteur des erreurs
- $\mathbf{X}^\top(\cdot) \in \mathbb{R}^D$  : pondère par les features

### Propriété cruciale pour la suite

La MSE linéaire est **convexe** en  $\mathbf{w} \Rightarrow$  **un seul minimum global**.

La descente de gradient est garantie de le trouver (avec un bon  $\alpha$ ).



**Problème** : gérer le biais  $b$  séparément est fastidieux.

$$\hat{y}_i = \mathbf{w}^\top \mathbf{x}_i + b \Rightarrow \text{deux termes à maintenir}$$

**Astuce** : ajouter une colonne de 1 dans  $\mathbf{X}$  :

$$\tilde{\mathbf{x}}_i = \begin{pmatrix} 1 \\ x_{i,1} \\ \vdots \\ x_{i,D} \end{pmatrix} \in \mathbb{R}^{D+1} \quad \tilde{\mathbf{w}} = \begin{pmatrix} b \\ w_1 \\ \vdots \\ w_D \end{pmatrix}$$

$$\hat{y}_i = \tilde{\mathbf{w}}^\top \tilde{\mathbf{x}}_i \quad \Leftarrow \text{un seul terme !}$$

**En NumPy :**

```
1 # X : (N, D) -> X_aug : (N, D+1)
2 ones = np.ones((N, 1))
3 X_aug = np.hstack([ones, X])
4 # w_aug contient [b, w1, ..., wD]
5
6 # Prediction
7 y_hat = X_aug @ w_aug
8
9 # Gradient MSE augmentee
10 grad = (2/N) * X_aug.T @ (y_hat - y)
```

### Avantage

Toutes les équations (gradient, solution analytique) s'écrivent **sans cas particulier** pour  $b$ .

On cherche  $\mathbf{w}^* = \arg \min_{\mathbf{w}} \mathcal{L}(\mathbf{w})$ .

**Idée :** le gradient  $\nabla_{\mathbf{w}} \mathcal{L}$  pointe vers la **plus grande pente montante**.

On avance dans la **direction opposée**.

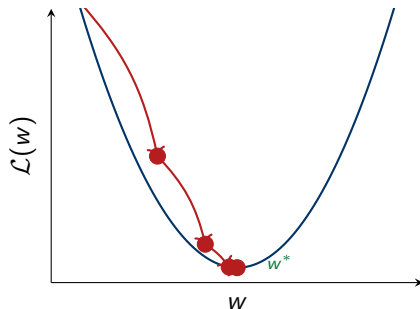
**Règle de mise à jour :**

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w})$$

$\alpha$  learning rate (hyperparamètre)

$\nabla_{\mathbf{w}} \mathcal{L}$  gradient de la loss

Répéter jusqu'à **convergence**.



```
1 def gradient_descent(X, y, lr, n_iters):
2     N, D = X.shape
3     w = np.zeros(D)          # init
4
5     for t in range(n_iters):
6         y_hat = X @ w
7         error = y_hat - y    # (N,)
8
9         grad = (2/N) * X.T @ error    # (D,)
10        w = w - lr * grad
11
12    return w
```

## Complexité

$\mathcal{O}(N \cdot D)$  par itération.

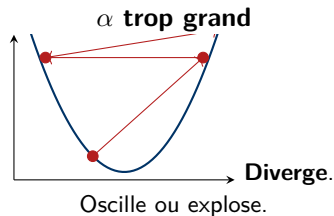
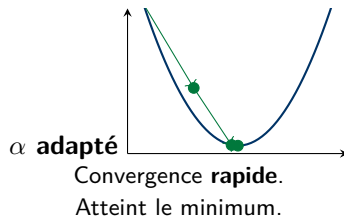
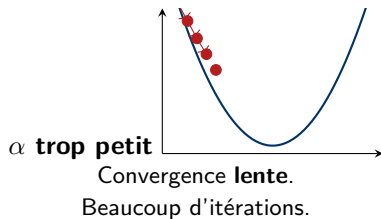
Sur grand dataset : utiliser **Mini-Batch GD**.

## Variantes :

Batch GD	Gradient sur <b>tout</b> le dataset
SGD	Gradient sur <b>1 exemple</b> aléatoire
Mini-Batch	Gradient sur un <b>batch</b> de taille $B$

## En pratique

Mini-Batch ( $B = 32$  à  $256$ ) est le standard.  
C'est ce qu'utilise PyTorch avec DataLoader.



### Règle pratique

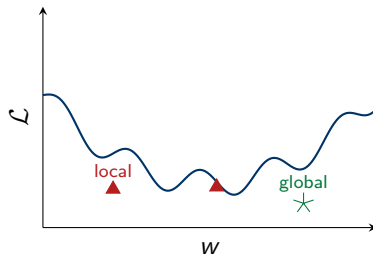
Commencer à  $\alpha = 10^{-3}$ , observer la courbe de loss.

Si la loss **oscille** → diviser par 10. Si elle **stagne** → multiplier par 3.

## Problèmes généraux

- Converge vers un **minimum local** (pas nécessairement global)
- Résultat dépend de **l'initialisation**
- Peut **ne pas converger** si  $\alpha$  trop grand
- Lent sur de très grands datasets (Batch GD)

Minima locaux vs global



## Bonne nouvelle pour la régression linéaire

La MSE est **convexe**  $\Rightarrow$  un seul minimum, GD le trouve toujours.  
C'est une propriété exceptionnelle — les réseaux profonds ne l'ont pas.

## Vocabulaire ML

- Task  $T$ , Performance  $P$ , Experience  $E$
- Supervisé / Non-supervisé / Renforcement
- Features  $\mathbf{X}$ , labels  $\mathbf{y}$ , modèle  $f_{\Theta}$ , loss  $\mathcal{L}$

## Descente de Gradient

- $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} \mathcal{L}$
- Batch / SGD / Mini-Batch
- $\alpha$  trop petit : lent     $\alpha$  trop grand : diverge
- MSE convexe  $\Rightarrow$  minimum global garanti

## Régression Linéaire

- $\hat{y} = \mathbf{w}^{\top} \mathbf{x} + b$
- MSE :  $\mathcal{L} = \frac{1}{N} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$
- Gradient :  $\frac{2}{N} \mathbf{X}^{\top} (\mathbf{X}\mathbf{w} - \mathbf{y})$
- Augmented data : biais intégré dans  $\mathbf{w}$