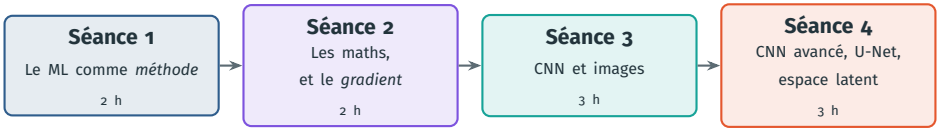
Du Machine Learning aux réseaux convolutifs

CNN, U-Net, auto-encodeurs et VAE

Pierre Lague • pierre.lague@inria.fr

Cours Bachelor 3 — 10 heures

Fil rouge



Une seule idée, déclinée quatre fois

On définit un **modèle** avec des paramètres, une **fonction de coût** qui mesure son erreur, et on **descend le gradient**. Tout le reste — CNN, U-Net, VAE — n'est qu'un choix différent de modèle et de coût.

Mode d'emploi

Définition
Ce qu'il faut savoir énoncer précisément.

Intuition
L'image mentale à garder. Pas rigoureuse, mais c'est elle qui reste.

Rappel mathématique
Le minimum d'outillage mathématique, réexpliqué à chaque fois.

Piège classique
L'erreur que tout le monde fait.

En pratique
Ce qu'on tape réellement dans le code, et les valeurs classiques.

À retenir
Le résumé de fin de section.

Les mots-clés anglais sont notés *comme ceci* : ce sont eux que vous retrouverez dans la documentation et les articles.

Sommaire

Séance 1 — La méthode ML

Objectifs

À la fin de cette séance, vous saurez

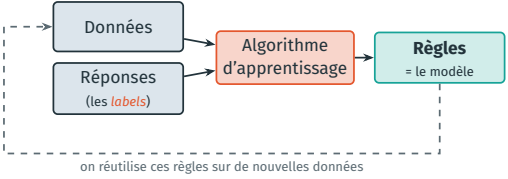
- dire ce qu'est **apprendre** pour une machine ;
- dérouler le **pipeline** d'un projet ML sans en sauter une étape ;
- expliquer **pourquoi** on sépare les données en trois ;
- nommer chaque **hyperparamètre**, dire à quoi il sert et ce qui se passe s'il est mal réglé ;
- choisir la **métrique** adaptée à un problème et lire une matrice de confusion.

Plan

Programmer vs. apprendre

Programmation classique

Machine Learning



Intuition

On **inverse** le sens de la programmation : au lieu d'écrire les règles à la main, on montre des exemples et la machine *en déduit* les règles.

Définition

Apprentissage automatique (Tom Mitchell, 1997)

Un programme **apprend** de l'expérience *E*, pour une tâche *T*, mesurée par une performance *P*, si sa performance *P* sur la tâche *T* **s'améliore** avec l'expérience *E*.

T — la tâche

Ce qu'on veut faire.

« Reconnaître si une image contient une tumeur. »

E — l'expérience

Les données.

« 10 000 IRM déjà annotées par un radiologue. »

P — la performance

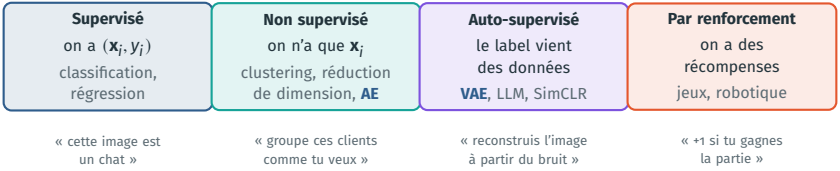
La métrique.

« Le pourcentage de tumeurs correctement détectées. »

Retenez surtout ceci

Si vous ne savez pas énoncer *T*, *E* **et** *P*, vous ne pouvez pas encore prétendre utiliser le Machine Learning. Ce sont les conditions nécessaires (en gros).

Grandes familles



Intuition

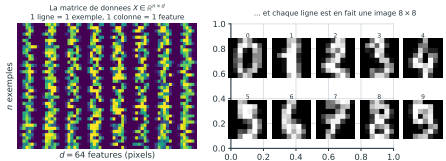
Ce cours parcourt les trois premières. Les **CNN** (séance 3) sont typiquement **supervisés**; les **auto-encodeurs** et **VAE** (séance 4) sont **non** ou **auto**-supervisés : ils apprennent sans étiquettes.

Intuition

Comprenez la supervision comme de la surveillance : non supervisé = on laisse la machine se débrouiller seule,
supervisé = on lui dit ce qu'il faut faire.

Les objets de base

- un **exemple** (*sample*) : une ligne, $\mathbf{x}_i \in \mathbb{R}^d$
- une **variable** (*feature*) : une colonne
- une **cible** (*label*, *target*) : y_i
- le **jeu de données** (*dataset*) : $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$



n = nombre d'exemples, d = nombre de features. Pour une image 28×28 « aplatie », $d = 784$.

$$\mathbf{X} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1d} \\ x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nd} \end{pmatrix} \in \mathbb{R}^{n \times d}, \quad \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Convention

n lignes, d colonnes. *Toujours.* La moitié des bugs de shape viennent d'une matrice transposée sans s'en rendre compte : réflexe `print(X.shape)`.

Régression

La cible est **continue** : $y \in \mathbb{R}$.

Prix d'un appartement, température de demain, âge estimé sur une photo.

Classification

La cible est **discrète** : $y \in \{1, \ldots, K\}$.

Chat / chien, spam / non-spam, quel chiffre est écrit.

	Régression	**Classification**
Sortie du modèle	un nombre	un vecteur de probabilités
Dernière couche	aucune (linéaire)	`sigmoid` (2 classes) / `softmax` (K)
Perte usuelle	MSE, MAE	Cross-Entropy
Métrique usuelle	RMSE, MAE, R^2	Accuracy, F1, AUC

Piège classique

Une classification à 3 classes notées 0, 1, 2 n'est **pas** une régression : la classe 1 n'est pas « entre » les classes 0 et 2. Utilisez le *one-hot encoding*.

Question

Vous voulez prédire, pour un patient, **le nombre de jours** avant sa sortie d'hôpital. Régression ou classification ? Et si vous vouliez prédire s'il sort **avant ou après** 7 jours ?

Question

Vous voulez prédire, pour un patient, **le nombre de jours** avant sa sortie d'hôpital. Régression ou classification ? Et si vous vouliez prédire s'il sort **avant ou après** 7 jours ?

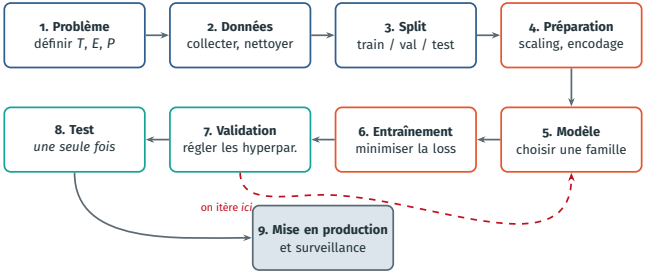
Réponse

1. Régression (une durée est continue).

2. Classification binaire.

C'est vous qui *choisissez* la formulation : le même problème métier peut donner deux tâches ML différentes, avec des métriques différentes.

Une méthode



La règle d'or

La boucle d'itération passe par la **validation**, **jamais** par le test.

Piège classique

80 % du temps d'un projet réel est passé aux étapes 2 et 4. Le `model.fit()` est la partie la plus rapide et la moins intéressante.

Étape 2 : la check-list « données »

Valeurs manquantes ? **doublons** ? classes équilibrées ? valeurs aberrantes ? unités cohérentes ? labels fiables ?

Garbage in, garbage out : aucun modèle ne rattrape des données fausses ou mal étiquetées.

Train, validation, test

L'analogie de l'examen

Le **train** : les exercices d'entraînement corrigés assez variés.
La **validation** : les annales, pour ajuster votre révision.
Le **test** : l'examen. Si vous l'avez vu avant, ça n'a aucun interêt.

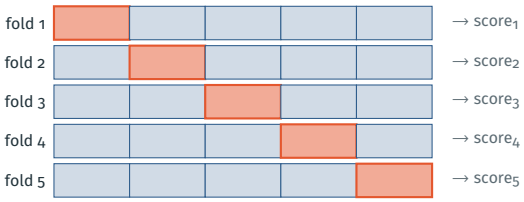
Piège classique

Chaque fois que vous regardez le test pour décider quelque chose, il devient un jeu de validation — et votre estimation devient optimiste.

Intuition

Le risque reste que l'examen ne ne soit pas représentatif de la vraie vie : c'est le **domaine shift**.

Validation croisée



k-fold

On découpe le train en k blocs. On entraîne k fois, en gardant à chaque fois un bloc différent pour valider. Le score final est la **moyenne** des k scores.

$$\text{score} = \frac{1}{k} \sum_{j=1}^{k} \text{score}_j$$

Intuition

Utile quand on a **peu de données** : chaque exemple sert une fois à valider. Coût : on entraîne k fois. Sur un CNN, c'est souvent trop cher → on garde un simple *holdout*.

L'**écart-type** des k scores est presque aussi informatif que la moyenne : il vous dit si votre résultat est stable.

Fuite de données

Définition

Il y a **fuite** dès qu'une information du jeu de test (ou du futur) se retrouve, directement ou indirectement, dans l'entraînement. Symptôme : des scores *trop beaux* qui s'effondrent en production.

Les 5 fuites les plus fréquentes

1. Normaliser **avant** de couper : la moyenne utilisée contient le test.
2. Des **doublons** présents des deux côtés.
3. Une feature calculée à partir de la cible (« *montant remboursé* » pour prédire un défaut).
4. Séries temporelles coupées au hasard : on prédit le passé avec le futur.
5. Plusieurs images du **même patient** réparties entre train et test.

Le bon réflexe : le `Pipeline`

```python
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import
    StandardScaler
from sklearn.linear_model import
    LogisticRegression

# le scaler est ajusté SUR LE TRAIN
    seulement,
# à chaque fold, automatiquement
pipe = make_pipeline(StandardScaler(),
                     LogisticRegression())
pipe.fit(X_train, y_train)
pipe.score(X_test, y_test)
```

Couper correctement

On isole d'abord le **test** (`train_test_split`, `test_size=.2`), puis on recoupe le reste en **train / validation**.

`stratify=y` si classes déséquilibrées • `GroupKFold` si des exemples sont liés (même patient) • `TimeSeriesSplit` pour des séries temporelles • `random_state` toujours fixé.

Le modèle

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

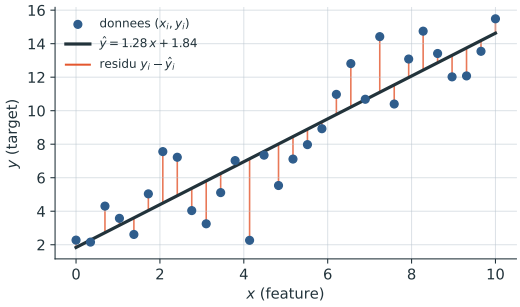
w_j : **poids** du feature j

b : **biais** (*intercept*)

$d + 1$ paramètres à apprendre

Le coût : moindres carrés

$$J(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$



Chaque trait orange est un **résidu**. On cherche la droite qui minimise la *somme des carrés* de ces traits.

La solution exacte existe

$\hat{\mathbf{w}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$. On pourrait s'arrêter là… mais inverser une matrice $d \times d$ coûte $O(d^3)$: impossible dès que d est grand (rappel : une image, c'est $d = 150\,000$). D'où la **descente de gradient**.

Question

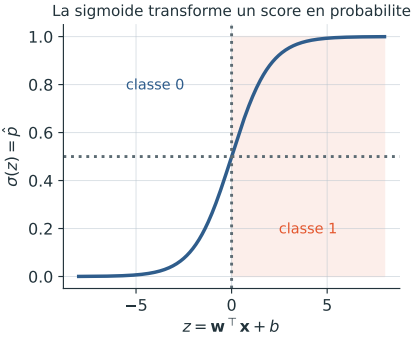
La Régression Logistique est un modèle de **régression** ou de **classification** ?

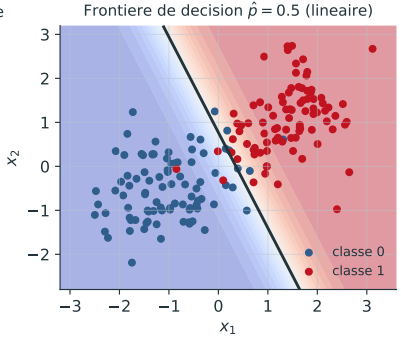
Question

La Régression Logistique est un modèle de **régression** ou de **classification** ?

Réponse

C'est un modèle de **classification** : il prédit la probabilité d'appartenance à une classe. Le nom « régression » vient de l'origine historique du modèle, mais il est utilisé pour des tâches de classification.





Définition

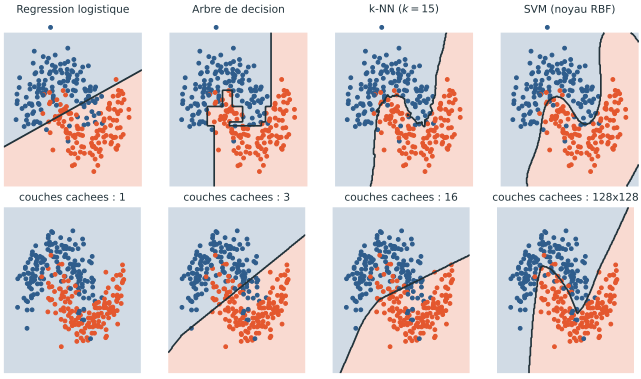
$$\hat{p} = \sigma(\mathbf{w}^\top \mathbf{x} + b), \qquad \sigma(z) = \frac{1}{1 + e^{-z}} \in]0, 1[$$

puis $\hat{y} = 1$ si $\hat{p} \geq 0.5$, sinon 0.

Intuition

Le modèle calcule un **score** linéaire z, et la sigmoïde l'écrase dans $[0, 1]$ pour en faire une **probabilité**. La frontière $\hat{p} = 0.5$ correspond à $z = 0$: c'est un hyperplan, donc une frontière **droite**.

Limites du linéaire



« No free lunch »

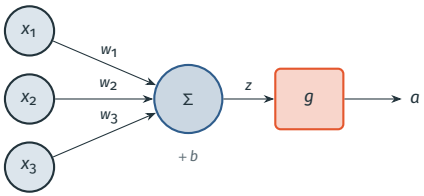
Aucun modèle n'est le meilleur partout. Le bon modèle dépend de la **forme des données**, de leur **quantité** et de vos contraintes.

Capacité

Le nombre de neurones cachés contrôle la **richesse des formes** que le modèle peut dessiner. C'est un hyperparamètre.

Piège classique

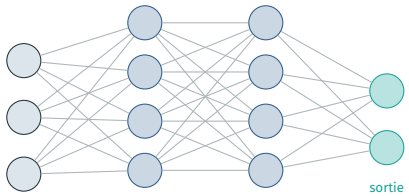
Plus de capacité ≠ meilleur modèle : sur la dernière figure, la frontière fait des détours pour attraper des points isolés. C'est l'**overfitting** qui commence. Partez toujours d'une **baseline** simple.



Un neurone artificiel

$$a = g(\mathbf{w}^\top \mathbf{x} + b)$$

Une **combinaison linéaire**, puis une **non-linéarité** g (ReLU, sigmoïde…).



Multi-Layer Perceptron

Empiler des couches de neurones permet d'approcher *n'importe quelle* fonction continue. La non-linéarité g est indispensable : sans elle, empiler des couches linéaires redonne… une couche linéaire.

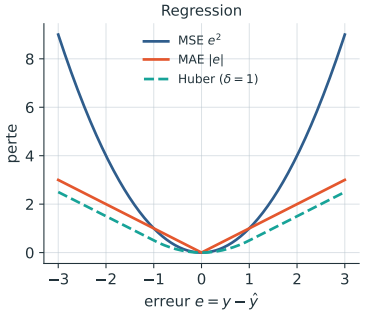
Fonction de coût

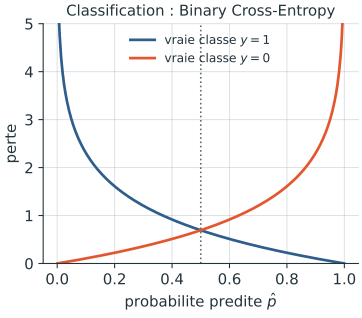
Loss / cost function

Une fonction $\mathcal{L}(\hat{y}, y) \geq 0$ qui vaut 0 si la prédiction est parfaite et grandit avec l'erreur. Le **coût total** est sa moyenne sur le jeu de données :

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f_\theta(\mathbf{x}_i), y_i)$$

Apprendre, c'est résoudre $\hat{\theta} = \arg\min_{\theta} J(\theta)$.





Pertes usuelles

Tâche	Perte	Pourquoi / quand
Régression	MSE $\frac{1}{n}\sum(\hat{y}-y)^2$	le défaut ; pénalise fort les grosses erreurs
Régression	MAE $\frac{1}{n}\sum\lvert\hat{y}-y\rvert$	robuste aux *outliers*
Régression	Huber	compromis : MSE près de 0, MAE au loin
Classif. binaire	BCE $-[y\log\hat{p}+(1-y)\log(1-\hat{p})]$	sortie `sigmoid`
Classif. K classes	Cross-Entropy $-\sum_k y_k \log\hat{p}_k$	sortie `softmax`
Classes déséquilibrées	CE pondérée, Focal Loss	donne plus de poids aux classes rares
Segmentation	BCE + Dice	*séance 4*
Auto-encodeur	MSE ou BCE pixel à pixel	*séance 4*
VAE	reconstruction + β·KL	*séance 4*

⚠ Piège classique

La **loss** est ce que le modèle *minimise* ; la **métrique** est ce que *vous* regardez. Elles sont souvent différentes : on optimise la cross-entropy (dérivable) mais on juge sur l'accuracy ou le F1 (non dérivables).

L'image mentale à garder pour tout le cours

Vous êtes dans le brouillard sur une montagne, et vous voulez descendre. Vous ne voyez pas la vallée, mais vous sentez **la pente sous vos pieds**. Alors vous faites un pas dans la direction qui descend le plus, et vous recommencez. Cette pente, c'est le **gradient** — on y consacrera la moitié de la séance 2.

L'algorithme, en trois lignes

1. Initialiser θ au hasard.
2. Répéter :
$$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta)$$
3. S'arrêter quand ça ne bouge plus (ou après N epochs).

Pourquoi le signe « moins » ?

$\nabla_\theta J$ pointe vers la plus forte **montée**. Pour descendre, on va donc dans la direction **opposée**. (Démonstration visuelle en séance 2.)

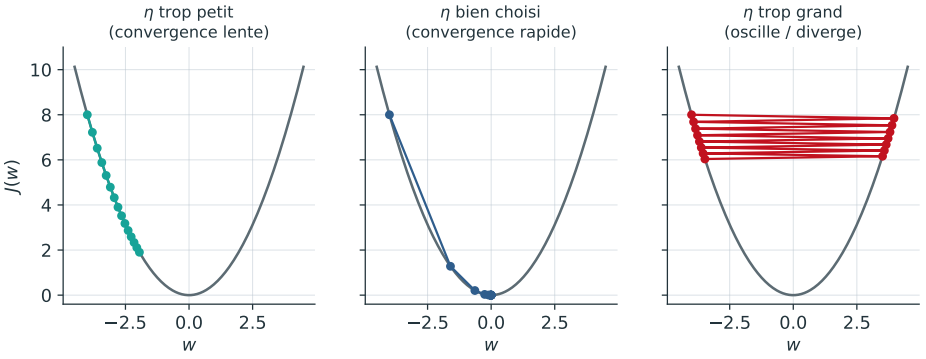
Exemple : régression linéaire

$$\frac{\partial J}{\partial w_j} = \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)\, x_{ij}$$

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

Le gradient est une moyenne pondérée des erreurs. Si le modèle sur-estime ($\hat{y} > y$) pour de grands x_j, alors w_j diminue. Logique.

Learning rate η



learning_rate (η)

Rôle : la *taille du pas* à chaque mise à jour des poids.

↓ Trop petit : convergence très lente, on peut rester coincé dans un plateau ou un minimum local. **↑ Trop grand :** la loss oscille, remonte, ou devient `NaN`.

✓ Valeur usuelle : 10^{-3} (Adam), 10^{-2} (SGD) -- puis on ajuste par facteurs de 10

Epoch, batch, iteration

Définition

- **batch** : le paquet d'exemples utilisé pour **une** mise à jour des poids.
- **iteration** (ou *step*) : **une** mise à jour des poids.
- **epoch** : un passage complet sur le jeu d'entraînement.

$$\text{iterations par epoch} = \left\lceil \frac{n}{\text{batch\_size}} \right\rceil$$

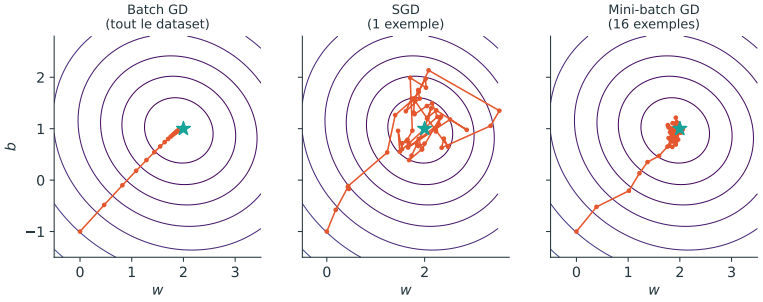
Calcul type

$n = 50\,000$ images, `batch_size = 64` :

$$\left\lceil \frac{50000}{64} \right\rceil = 782 \text{ itérations / epoch}$$

Sur 30 epochs : 23 460 mises à jour des poids.

Batch, SGD, mini-batch



Intuition

Le **bruit** du SGD n'est pas qu'un défaut : il permet parfois de s'échapper d'un mauvais minimum local. Mais il rend la trajectoire chaotique. Le **mini-batch** est le compromis universel.

`batch_size`

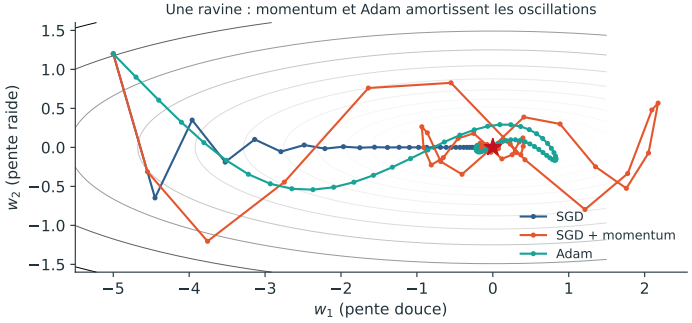
Rôle : combien d'exemples on moyenne pour estimer le gradient.

↓ **Trop petit :** gradient bruité, entraînement lent (mauvais usage du GPU), BatchNorm instable.

↑ **Trop grand :** gradient très lisse mais peu de mises à jour, saturation mémoire GPU.

✓ **Valeur usuelle :** `32, 64, 128 -- la plus grande valeur qui tient en mémoire`

Momentum et Adam



Momentum

$$\mathbf{v} \leftarrow \beta \mathbf{v} + \nabla J, \qquad \theta \leftarrow \theta - \eta \mathbf{v}$$

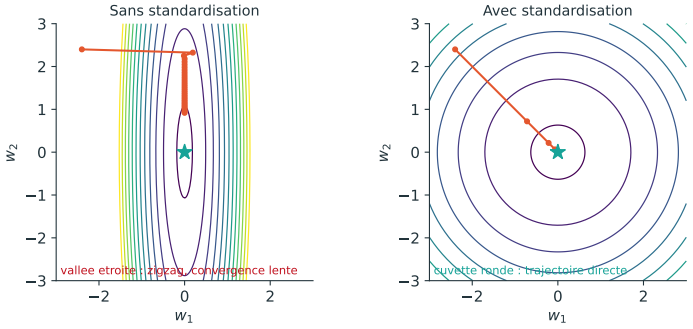
On garde de l'**inertie** : une bille qui roule traverse les petites bosses et n'oscille plus dans les vallées étroites. ($\beta = 0.9$)

Adam

Momentum **+** un pas **adapté à chaque paramètre** : les directions où le gradient est habituellement grand reçoivent un pas plus petit.

$\beta_1 = 0.9$, $\beta_2 = 0.999$: ne les touchez pas. C'est le choix par défaut raisonnable dans 90 % des cas.

Standardiser les features



Standardisation (*z-score*) -> centrer et réduire

$$x'_j = \frac{x_j - \mu_j}{\sigma_j} \quad \Rightarrow \quad \text{moyenne 0, écart-type 1}$$

Normalisation *min-max* -> mettre à l'échelle

$$x'_j = \frac{x_j - \min_j}{\max_j - \min_j} \in [0, 1]$$

C'est celle qu'on utilise pour les images : `img / 255`.

La règle absolue

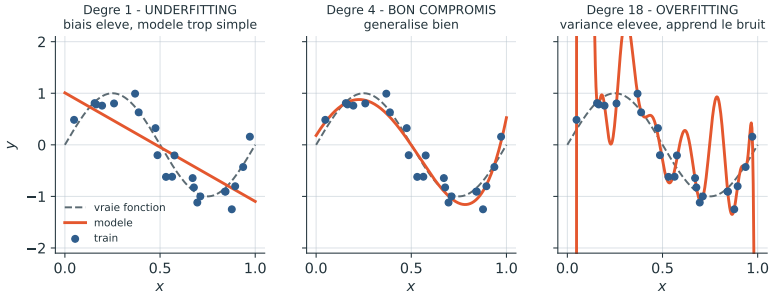
μ_j et σ_j se calculent **sur le train uniquement**, puis s'appliquent au val et au test.

```
sc = StandardScaler()
X_tr = sc.fit_transform(X_train)  # fit + apply
X_te = sc.transform(X_test)       # apply seul
```

Généralisation

Généraliser

Bien prédire sur des données **jamais vues**. Ce n'est *pas* la même chose que bien prédire sur le train — un modèle qui apprend le train par cœur y obtient 100 % et ne sert à rien.



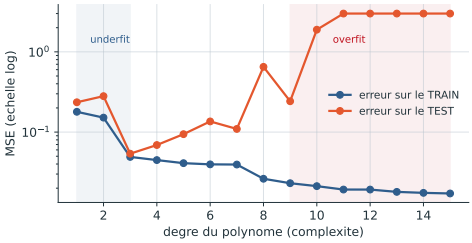
Underfitting

Le modèle est **trop simple** : il se trompe même sur le train. *Erreur train élevée, erreur test élevée.*

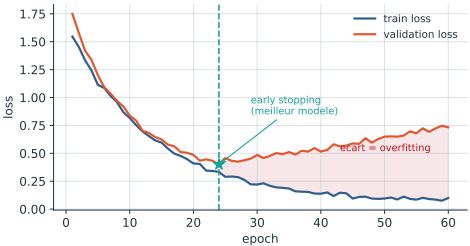
Overfitting

Le modèle apprend le **bruit** : il est parfait sur le train, mauvais ailleurs. *Erreur train faible, erreur test élevée.*

Courbes d'apprentissage



Erreur **en fonction de la complexité** du modèle.

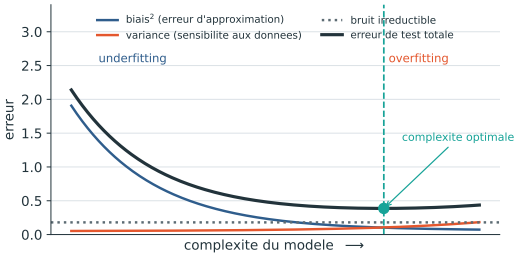


Erreur **en fonction du temps d'entraînement** (*learning curves*).

Le réflexe de diagnostic

Train	Validation	Diagnostic et remède
élevée	élevée	**underfitting** → modèle plus grand, entraîner plus longtemps, moins de régularisation
faible	élevée	**overfitting** → plus de données, augmentation, régularisation, early stopping
faible	faible	**c'est bon** → passez au test
élevée	faible	bug ou fuite : vérifiez votre split

Biais et variance



La décomposition

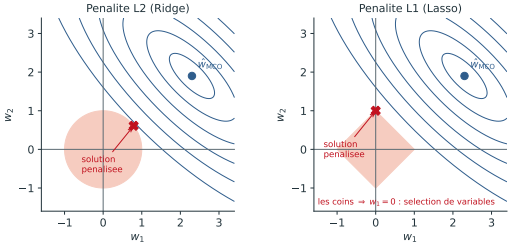
$$\underbrace{\mathbb{E}\left[(y - \hat{y})^2\right]}_{\text{erreur de test}} = \text{Biais}^2 + \text{Variance} + \sigma^2$$

- **Biais²** : le modèle est trop rigide
- **Variance** : trop sensible aux données
- σ^2 : le bruit, irréductible

L'analogie du tir à l'arc

Biais = toutes vos flèches groupées, mais à côté de la cible.
Variance = vos flèches éparpillées autour de la cible.
On ne peut pas réduire les deux à la fois sans plus de données.

Régularisation



Définition

On ajoute une **pénalité** sur la taille des poids :

$$J_{\text{reg}}(\mathbf{w}) = J(\mathbf{w}) + \lambda\,\Omega(\mathbf{w})$$

- **Ridge** (L2) : $\Omega = \|\mathbf{w}\|_2^2 = \sum_j w_j^2$
  → tous les poids rétrécissent
- **Lasso** (L1) : $\Omega = \|\mathbf{w}\|_1 = \sum_j |w_j|$
  → certains poids deviennent exactement 0

Intuition

Des poids plus petits ⇒ une fonction plus lisse ⇒ moins de capacité à épouser le bruit.

lambda (λ) -- en deep learning : weight_decay

Rôle : doser la pénalité : le curseur entre « coller aux données » et « rester simple ».
↓ Trop petit : aucune régularisation, overfitting possible. **↑ Trop grand :** underfitting : tous les poids tendent vers 0, le modèle prédit une constante.
✓ Valeur usuelle : 10^{-4} à 10^{-2}, à chercher sur une grille logarithmique

Remèdes à l'overfitting

Technique	Principe	Quand
Plus de données	la seule solution qui n'a aucun inconvénient	toujours en premier
Data augmentation	créer des variantes plausibles (rotation, flip…)	images, audio — *séance 3*
L2 / weight decay	pénaliser les gros poids	partout, par défaut
L1 / Lasso	pénaliser + mettre des poids à zéro	si on veut sélectionner des features
Dropout	éteindre p % des neurones au hasard à chaque batch	couches denses des réseaux
Early stopping	arrêter quand la val cesse de progresser	gratuit, à faire systématiquement
Réduire la capacité	moins de couches, moins de neurones	si le modèle est manifestement sur-dimensionné
Batch Normalization	stabilise, régularise un peu au passage	CNN — *séance 3*

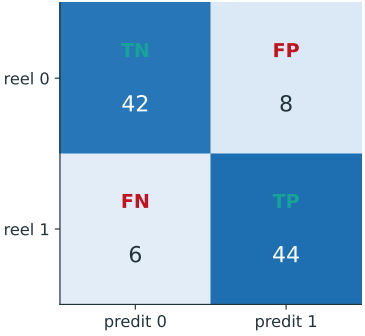
`dropout_rate` (p)

Rôle : proportion de neurones désactivés aléatoirement *pendant l'entraînement* (jamais à l'inférence).

↓ **Trop petit :** effet régularisant négligeable. ↑ **Trop grand :** le réseau n'apprend plus, la loss d'entraînement stagne.

✓ **Valeur usuelle :** `0.2 à 0.5 sur les couches denses ; 0 à 0.1 sur les couches convolutives`

Matrice de confusion



Définition

- **TP** (*true positive*) : positif, bien détecté
- **TN** : négatif, bien rejeté
- **FP** (*false positive*) : fausse alerte — *erreur de type I*
- **FN** : détection manquée — *erreur de type II*

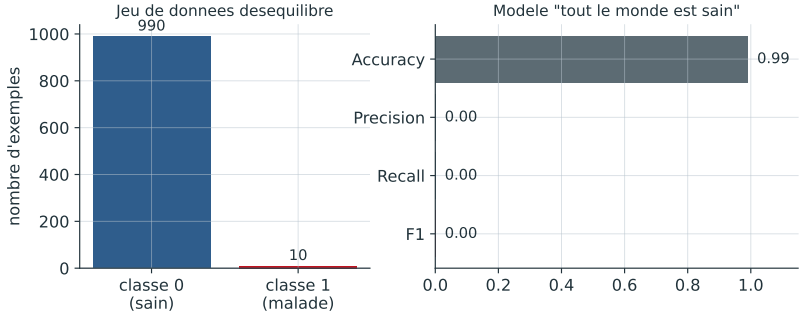
$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \qquad \text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN} \qquad F_1 = 2 \cdot \frac{\text{Prec} \cdot \text{Rec}}{\text{Prec} + \text{Rec}}$$

Les retenir en une phrase

Precision : « quand je crie au loup, ai-je raison ? » **Recall** : « ai-je repéré tous les loups ? » **F1** : leur moyenne harmonique, qui punit celui qui néglige l'un des deux.

Piège de l'accuracy



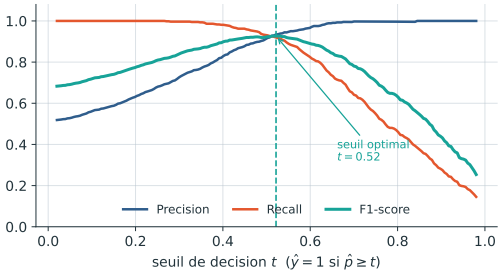
⚠ Le modèle « constant »

Sur un jeu où 99 % des patients sont sains, le modèle qui répond *toujours* « sain » atteint 99 % d'accuracy sans rien avoir appris — et ne sauve personne.

En pratique

Dès que les classes sont déséquilibrées : regardez le **recall** de la classe rare, le **F1**, la **balanced accuracy** ou l'**AUC-PR**. Et affichez toujours la matrice de confusion.

Choix du seuil



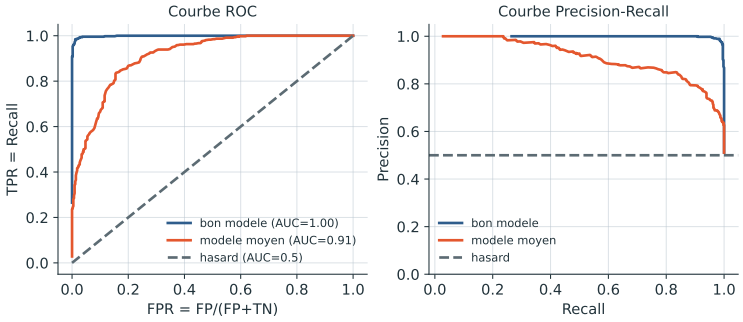
Intuition

Le modèle sort une **probabilité** $\hat{p}$. C'est *vous* qui décidez du seuil t au-delà duquel on déclare « positif ».

Le seuil dépend du coût des erreurs

- Dépistage d'un cancer : un **FN** est catastrophique → seuil **bas**, on privilégie le recall.
- Filtre anti-spam : un **FP** (mail important en spam) est très gênant → seuil **haut**, on privilégie la precision.

ROC et Precision-Recall



AUC

L'aire sous la courbe ROC. C'est la probabilité que le modèle donne un score plus élevé à un positif tiré au hasard qu'à un négatif tiré au hasard. 0.5 = hasard, 1.0 = parfait.

Piège classique

La courbe ROC est **trompeuse** quand les classes sont très déséquilibrées : elle reste belle même quand la precision est catastrophique. Dans ce cas, préférez la courbe **Precision-Recall**.

Avantage des deux courbes : elles évaluent le modèle *à tous les seuils*, donc indépendamment de ce choix.

Métriques de régression

Métrique	Formule	Interprétation
MSE	$\frac{1}{n}\sum(y_i - \hat{y}_i)^2$	pénalise fort les grosses erreurs ; unité au carré
RMSE	$\sqrt{\text{MSE}}$	**même unité que** y : lisible
MAE	$\frac{1}{n}\sum\lvert y_i - \hat{y}_i\rvert$	erreur moyenne « honnête », robuste
R^2	$1 - \frac{\sum(y_i-\hat{y}_i)^2}{\sum(y_i-\bar{y})^2}$	part de variance expliquée ; 1 = parfait, 0 = aussi bon que prédire la moyenne, **peut être négatif**
MAPE	$\frac{100}{n}\sum\left\lvert\frac{y_i-\hat{y}_i}{y_i}\right\rvert$	en %, mais explose si $y_i \approx 0$

En pratique

Donnez toujours une **référence** : un RMSE de 12 000 € est excellent pour des villas, catastrophique pour des studios. Comparez au modèle « je prédis toujours la moyenne ».

Question

Votre modèle de détection de fraude obtient : accuracy = 0.97, precision = 0.82, recall = 0.31.

Que se passe-t-il, et que faites-vous ?

Question

Votre modèle de détection de fraude obtient : accuracy = 0.97, precision = 0.82, recall = 0.31.

Que se passe-t-il, et que faites-vous ?

Réponse

Il rate **69 %** des fraudes. L'accuracy est élevée simplement parce que les fraudes sont rares.

Remèdes : **baisser le seuil** de décision, pondérer la classe rare dans la loss (`class_weight`), rééchantillonner — et suivre le **F1** ou l'**AUC-PR**, pas l'accuracy.

Paramètres et hyperparamètres

Paramètre θ

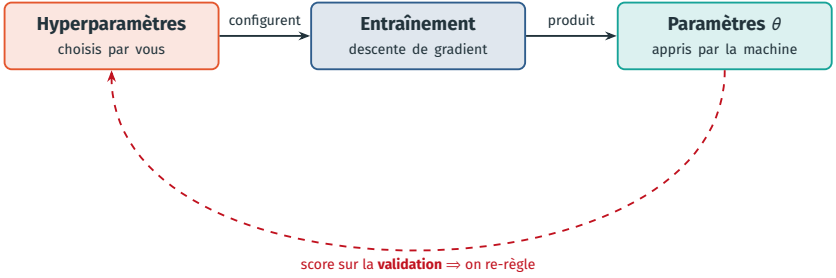
Appris **automatiquement** par la descente de gradient, à partir des données.

Les poids **w**, les biais b, les noyaux de convolution. Un ResNet-50 en a 25 millions. Vous n'en réglez aucun à la main.

Hyperparamètre

Choisi **par vous**, **avant** l'entraînement. Il ne s'apprend pas par gradient : on le règle en essayant, et en comparant **sur la validation**.

Learning rate, batch size, nombre de couches, λ, dropout, k de k-NN…



learning_rate

Rôle : taille du pas de la descente de gradient. *Le plus important de tous.*

↓ **Trop petit :** convergence lente, on plafonne trop tôt. ↑ **Trop grand :** oscillations, divergence, `NaN`.

✓ **Valeur usuelle :** `1e-3 avec Adam ; commencer par un` *lr range test*

batch_size

Rôle : nombre d'exemples par mise à jour ; contrôle le bruit du gradient et l'usage mémoire.

↓ **Trop petit :** gradient bruité, epochs très lentes, BatchNorm instable. ↑ **Trop grand :** mémoire GPU saturée, moins de mises à jour, généralisation parfois moins bonne.

✓ **Valeur usuelle :** `32 / 64 / 128`

n_epochs

Rôle : combien de fois on repasse sur tout le jeu de données.

↓ **Trop petit :** underfitting : le modèle n'a pas fini d'apprendre. ↑ **Trop grand :** overfitting (mais `EarlyStopping` vous protège).

✓ **Valeur usuelle :** `« beaucoup » + early stopping sur la validation`

optimizer

Rôle : la règle de mise à jour des poids (SGD, SGD+momentum, Adam, AdamW…).

↓ **Trop petit :** — ↑ **Trop grand :** —

✓ **Valeur usuelle :** `Adam / AdamW pour démarrer ; SGD+momentum pour les records en vision`

`nombre de couches / de neurones`

Rôle : la *capacité* du modèle : la complexité des fonctions qu'il peut représenter.

↓ **Trop petit :** underfitting : le modèle est trop pauvre pour le problème. ↑ **Trop grand :** overfitting, calcul coûteux, entraînement instable.

✓ **Valeur usuelle :** `partir petit, agrandir tant que la validation s'améliore`

`weight_decay` (λ)

Rôle : pénalité L2 sur les poids : pousse le modèle vers des solutions plus lisses.

↓ **Trop petit :** pas de régularisation. ↑ **Trop grand :** underfitting, les poids s'écrasent vers 0.

✓ **Valeur usuelle :** `1e-4 (vision) ; grille logarithmique 1e-6` → `1e-2`

`dropout_rate`

Rôle : proportion de neurones éteints au hasard pendant l'entraînement.

↓ **Trop petit :** effet négligeable. ↑ **Trop grand :** le réseau n'arrive plus à apprendre.

✓ **Valeur usuelle :** `0.2-0.5 sur les couches denses`

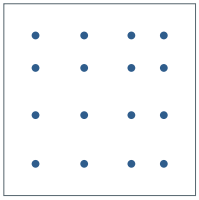
`k (k-NN) / max_depth (arbre) / C, gamma (SVM)`

Rôle : les hyperparamètres de capacité des modèles classiques.

↓ **Trop petit :** k petit, arbre profond, C grand ⇒ overfitting. ↑ **Trop grand :** k grand, arbre peu profond, C petit ⇒ underfitting.

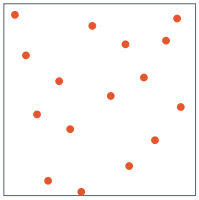
✓ **Valeur usuelle :** `GridSearchCV sur une grille logarithmique`

Stratégies de réglage



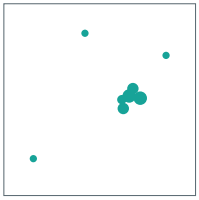
Grid search

exhaustif, mais gaspille des essais sur le paramètre inutile



Random search

explore plus de valeurs distinctes par paramètre — souvent *meilleur* à budget égal



Bayesian / Optuna

apprend des essais passés et se concentre sur la zone prometteuse

La méthode qui marche vraiment

1. Réglez d'abord le **learning rate** seul — il domine tout le reste.
2. Cherchez sur une **échelle logarithmique** ($10^{-5}, 10^{-4}, \ldots$), jamais linéaire.
3. Changez **un paramètre à la fois** et notez le score de validation.
4. Ne comparez que des runs avec le **même seed** et le même split.

Pipeline scikit-learn

```python
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score

X, y = load_breast_cancer(return_X_y=True)

# 1. on isole le test AVANT tout le reste
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=.2,
                                          stratify=y, random_state=42)

# 2. pipeline : le scaler est ré-ajusté à chaque fold -> pas de fuite
pipe = make_pipeline(StandardScaler(), LogisticRegression(max_iter=5000))

# 3. réglage de l'hyperparamètre C par validation croisée (sur le TRAIN)
grid = GridSearchCV(pipe, {"logisticregression__C": np.logspace(-3, 3, 7)},
                    cv=5, scoring="f1")
grid.fit(X_tr, y_tr)
print("meilleur C :", grid.best_params_, " F1 val :", round(grid.best_score_, 3))

# 4. évaluation finale : UNE SEULE FOIS, sur le test
y_pred = grid.predict(X_te)
print(confusion_matrix(y_te, y_pred))
print(classification_report(y_te, y_pred, digits=3))
print("AUC :", round(roc_auc_score(y_te, grid.predict_proba(X_te)[:, 1]), 3))
```

Boucle PyTorch

```python
import torch, torch.nn as nn

model = nn.Sequential(nn.Linear(d, 64), nn.ReLU(),
                      nn.Dropout(0.3),
                      nn.Linear(64, n_classes))
opt   = torch.optim.Adam(model.parameters(), lr=1e-3,
                         weight_decay=1e-4)
lossf = nn.CrossEntropyLoss()

for epoch in range(n_epochs):
    model.train()                          # active dropout / BN
    for xb, yb in train_loader:            # 1 batch
        opt.zero_grad()                    # 1. remettre à zéro
        out  = model(xb)                   # 2. forward
        loss = lossf(out, yb)              # 3. coût
        loss.backward()                    # 4. gradients
        opt.step()                         # 5. mise à jour

    model.eval()                           # désactive dropout / BN
    with torch.no_grad():                  # pas de gradients
        val_loss = evaluate(model, val_loader)
```

Les 5 lignes du cœur

1. `zero_grad()` — PyTorch **accumule** les gradients, il faut les effacer.
2. `forward` — calcul de la prédiction.
3. `loss` — mesure de l'erreur.
4. `backward()` — calcul de $\nabla_\theta \mathcal{L}$ par **rétropropagation**.
5. `step()` — $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$.

Piège classique

Oublier `model.eval()` est LE bug classique : le dropout reste actif et vos scores de validation sont faux (et instables d'un run à l'autre).

Récapitulatif — séance 1

Les idées

1. Le ML apprend des **règles** à partir d'**exemples**, il ne les reçoit pas.
2. Tout modèle = **une famille de fonctions** + **une loss** + **un optimiseur**.
3. On sépare **train / val / test** pour ne pas biaiser le modèle.
4. Le vrai ennemi est l'**overfitting** ; il se diagnostique avec deux courbes.
5. Les **hyperparamètres** se règlent sur la validation, jamais sur le test.
6. La **métrique** doit refléter le problème métier, pas la commodité.

Ce qui sert directement en séance 3 et 4

- la boucle d'entraînement PyTorch (identique pour un CNN) ;
- learning rate, batch size, epochs : *mêmes* hyperparamètres ;
- overfitting et régularisation : *mêmes* remèdes ;
- la cross-entropy : *même* loss ;
- le split : *même* discipline.

Séance 2 — Maths et gradient

Objectifs

À la fin de cette séance, vous saurez

- manipuler **vecteurs, matrices, tenseurs** et lire une *shape* ;
- **calculer** un gradient, savoir ce qu'il représente et dans quel sens on le suit ;
- dérouler une **rétropropagation** à la main sur un petit réseau ;
- expliquer pourquoi un gradient **s'évanouit** ou **explose**, et quoi y faire ;
- définir la **convolution**, lire une **gaussienne** et une **divergence KL**.

Plan

Pas de maths innutiles

Chaque notion sert **directement** en séance 3 ou 4. Et une notion sert *partout* : le **gradient**. On y consacre la moitié de la séance.

Notion		Où elle sert
Tenseur, *shape*	→	lire et débugger n'importe quel code de CNN
Produit scalaire, normes	→	ce que calcule un filtre ; les losses ; la régularisation
Matrice = transformation	→	couche dense, rotation d'image, augmentation
Rang, SVD, PCA	→	compression ⇒ espace latent (séance 4)
Gradient, règle de la chaîne	→	**comment tout réseau apprend, sans exception**
Convolution	→	toute la séance 3
Gaussienne, KL	→	l'espace latent régularisé (séance 4)

Structuré vs. non structuré

Données structurées

Un tableau. Chaque colonne a un **sens propre** : âge, salaire, code postal.

⇒ On interprète w_j : « +1 an d'âge fait +230 € ».

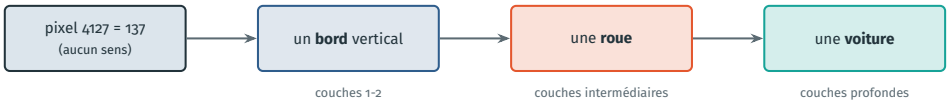
⇒ Arbres et régressions y sont excellents.

Données non structurées

Image, son, texte. Le pixel 4127 ne **signifie rien** tout seul ; c'est l'**organisation** qui porte le sens.

⇒ Il faut *extraire des features*.

⇒ C'est le métier du CNN : il les apprend.



La malédiction de la dimension

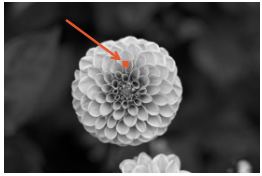
Une photo $224 \times 224 \times 3$ vit dans $\mathbb{R}^{150\,528}$. En si grande dimension, tous les points sont à peu près à la **même distance** : les méthodes fondées sur les distances s'effondrent.

Alors pourquoi ça marche ?

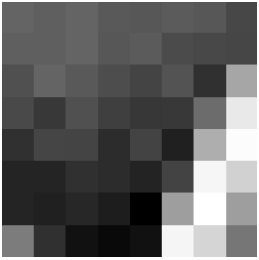
Parce que les **vraies** images n'occupent qu'une minuscule partie de cet espace : du bruit aléatoire n'est jamais une photo. Elles vivent sur une **variété** (*manifold*) de dimension bien plus faible — ce que l'espace latent de la séance 4 cherche justement à capturer.

L'image comme matrice

Image en niveaux de gris
427×640 pixels



Zoom : 8×8 pixels



... c'est juste une matrice
de nombres dans $[0, 255]$

147	146	147	145	144	145	144	140
146	146	147	144	145	141	140	140
142	147	144	141	139	143	134	164
140	136	142	138	136	136	149	181
134	139	139	133	139	130	166	186
131	131	134	133	131	139	184	175
131	130	132	129	122	162	187	162
153	134	126	124	126	184	176	152

Définition

Niveaux de gris : $I \in [0, 1]^{H \times W}$.
Couleur : un tenseur d'ordre 3, $I \in \mathbb{R}^{H \times W \times 3}$.

Indices

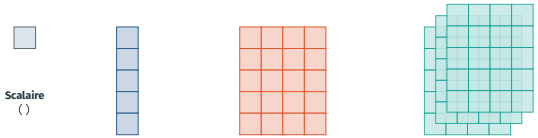
On indexe **(ligne, colonne)** = **(y, x)**, origine en **haut à gauche** — l'inverse d'un repère mathématique.

Deux conventions

`numpy/PIL` : **(H, W, C)**
`PyTorch` : **(C, H, W)**, par lot **(N, C, H, W)**.

D'où les `.permute(2,0,1)` partout.

Tenseurs et broadcasting



Broadcasting

NumPy et PyTorch **étirent** automatiquement les dimensions de taille 1 :

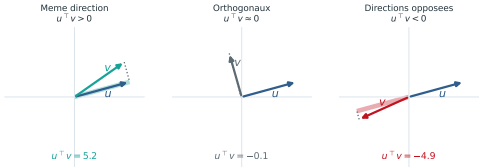
`(32,3,64,64) + (3,1,1)` → OK
`(32,64) + (64,)` → OK
`(32,64) + (32,)` → erreur

Piège classique

Le broadcasting *silencieux* est un piège : `(n,1) + (1,n)` produit une matrice `(n,n)` au lieu d'une erreur. Réflexe : `print(x.shape)`.

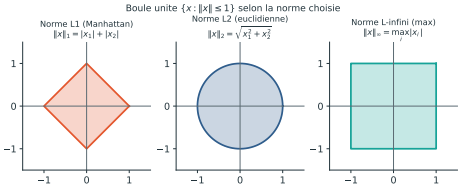
```python
import torch
x = torch.randn(32, 3, 64, 64)        # un batch
mean = torch.tensor([0.485, 0.456, 0.406])
std  = torch.tensor([0.229, 0.224, 0.225])

# normalisation par canal, sans aucune boucle :
x = (x - mean.view(1,3,1,1)) / std.view(1,3,1,1)

print(x.shape)                        # (32, 3, 64, 64)
print(x.mean(dim=(0,2,3)))            # ~ 0 par canal
```

Produit scalaire et normes



Définition

$$\mathbf{u}^\top \mathbf{v} = \sum_{j=1}^{d} u_j v_j = \|\mathbf{u}\|\,\|\mathbf{v}\| \cos\theta$$



Norme	Formule	Où on la croise
L1	$\sum_j \lvert x_j \rvert$	Lasso, MAE
L2	$\sqrt{\sum_j x_j^2}$	Ridge, MSE
L∞	$\max_j \lvert x_j \rvert$	attaques adverses

Pourquoi c'est LA formule du cours

Un **neurone** calcule $\mathbf{w}^\top \mathbf{x}$. Un **filtre de convolution** calcule $\mathbf{k}^\top \mathbf{x}_{\text{patch}}$. Même question à chaque fois : « *est-ce que ce que je vois ressemble à ce que je cherche ?* »

Une loss est une norme de l'erreur

MSE $= \frac{1}{n}\|\mathbf{y} - \hat{\mathbf{y}}\|_2^2$ et MAE $= \frac{1}{n}\|\mathbf{y} - \hat{\mathbf{y}}\|_1$.

Et les **coins** de la boule L1 sont exactement la raison pour laquelle le Lasso met des coefficients à zéro.

Vue « ligne × colonne »

Pour $A \in \mathbb{R}^{m \times k}$ et $B \in \mathbb{R}^{k \times n}$:

$$(AB)_{ij} = \sum_{\ell=1}^{k} A_{i\ell} B_{\ell j}$$

Le produit scalaire de la ligne i de A avec la colonne j de B.

La règle des dimensions

$$\underbrace{(m \times k)}_{A} \cdot \underbrace{(k \times n)}_{B} = (m \times n)$$

Les dimensions **intérieures** doivent coïncider ; les **extérieures** donnent le résultat.

Vue « transformation »

$A\mathbf{x}$ est une **combinaison linéaire des colonnes** de A :

$$A\mathbf{x} = x_1 \mathbf{a}_1 + \cdots + x_k \mathbf{a}_k$$

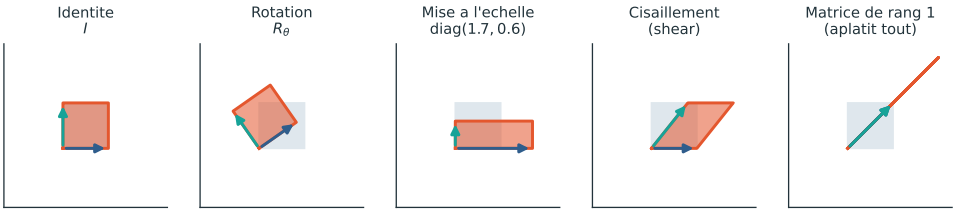
$\Rightarrow$ *A déplace* et *déforme* l'espace.

Une couche dense, c'est ça

$$\mathbf{H} = \underbrace{\mathbf{X}}_{(n \times d)} \underbrace{W}_{(d \times h)} + \mathbf{b} \quad \Rightarrow \quad (n \times h)$$

Tout un batch en *une* multiplication : c'est pour cela que les GPU sont si efficaces.

Transformations linéaires



Intuition

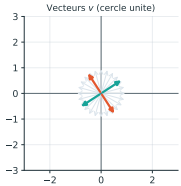
Pour savoir ce que fait une matrice, regardez où elle envoie les vecteurs de base $\mathbf{e}_1$ (bleu) et $\mathbf{e}_2$ (vert) : ce sont **ses colonnes**. Le reste suit par linéarité.

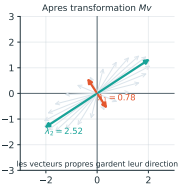
Rang

Le nombre de directions indépendantes qui survivent. La dernière matrice est de rang 1 : elle écrase le plan sur une droite. **L'information est perdue**, on ne peut pas revenir en arrière.

Ce sont exactement les transformations de la **data augmentation** (rotation, échelle, cisaillement) — séance 3.

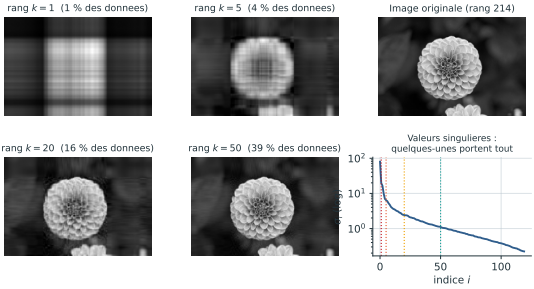
Vecteurs propres et SVD





Définition

$M\mathbf{v} = \lambda\mathbf{v}$: la direction est préservée, seule la **longueur** change. La plus grande valeur propre indique la direction où la matrice **étire le plus**.



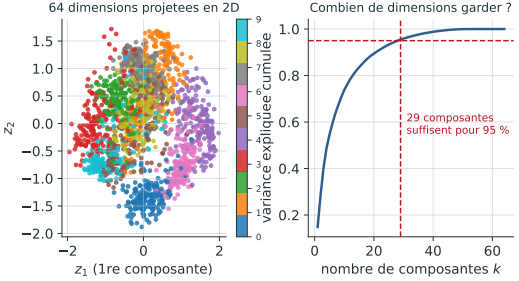
Décomposition en valeurs singulières

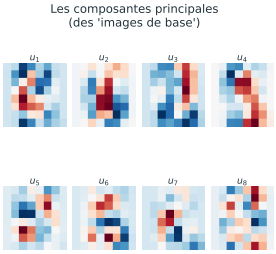
$$A = U\Sigma V^\top = \sum_{i=1}^{r} \sigma_i \, \mathbf{u}_i \mathbf{v}_i^\top, \quad \sigma_1 \geq \sigma_2 \geq \dots$$

Garder les k premiers termes donne la **meilleure** approximation de rang k.

Le message pour la séance 4

Une image réelle a un **spectre qui décroît vite** : quelques dizaines de composantes suffisent. C'est la preuve qu'il existe une **représentation compacte** — et c'est exactement ce qu'un espace latent cherche, mais de façon **non linéaire**.





Principal Component Analysis

On cherche les k directions orthogonales qui capturent le plus de **variance** (les vecteurs propres de la covariance). Projeter dessus donne un code $\mathbf{z} \in \mathbb{R}^k$ avec $k \ll d$.

À retenir

La PCA est un **auto-encodeur linéaire** :

$$\underbrace{\mathbf{z} = W^\top \mathbf{x}}_{\text{encodeur}}, \qquad \underbrace{\hat{\mathbf{x}} = W\mathbf{z}}_{\text{décodeur}}$$

On le vérifiera expérimentalement en séance 4.

Régression, CNN, U-Net, VAE : **tous** apprennent de la même façon.

$$\theta \quad \leftarrow \quad \theta \quad - \quad \eta \nabla_\theta \mathcal{L}$$

Si vous comprenez ce que vaut $\nabla_\theta \mathcal{L}$ et comment il se calcule, vous comprenez l'apprentissage profond. Le reste n'est que de l'architecture.

Les 3 questions

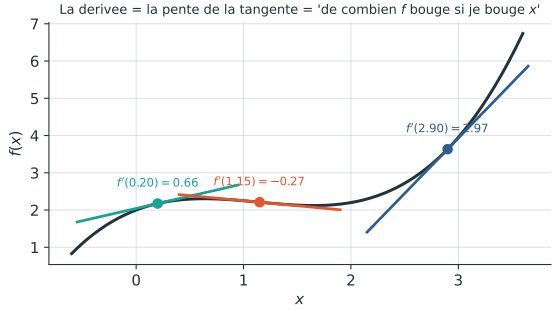
1. Que *représente* un gradient ?
2. Comment le *calcule*-t-on dans un réseau ?
3. Pourquoi *échoue*-t-il parfois ?

Le plan

dérivée → gradient → règle de la chaîne → rétropropagation → forme matricielle → pathologies.

Ce qu'on fera

Une rétropropagation complète **à la main**, avec les chiffres, sur un réseau à deux couches. C'est faisable, et ça démystifie tout.



Définition

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

La seule lecture à retenir

$f'(x)$ répond à : *« si j'augmente x d'un poil, de combien change f, et dans quel sens ? »*

$f' > 0$ ça monte $f' < 0$ ça descend $f' = 0$ c'est plat.

$$\frac{d}{dx}x^n = nx^{n-1} \quad \frac{d}{dx}e^x = e^x \quad \frac{d}{dx}\log x = \frac{1}{x} \quad \sigma'(x) = \sigma(x)(1-\sigma(x)) \quad \frac{d}{dx}\max(0,x) = \mathbb{1}_{x>0}$$

Définition

Pour $f : \mathbb{R}^d \to \mathbb{R}$, on dérive **une variable à la fois**, les autres étant gelées. Le gradient rassemble ces dérivées partielles :

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_d} \right)^\top \in \mathbb{R}^d$$

Piège classique

Le gradient est un **vecteur**, de la *même forme* que $\mathbf{x}$. Pour un réseau à 25 millions de paramètres, $\nabla_\theta \mathcal{L}$ a 25 millions de composantes — une par poids.

Un exemple, calculé entièrement

Soit $f(x, y) = 3x^2 + xy - 2y$.

$\frac{\partial f}{\partial x} = 6x + y$ (y est une constante)

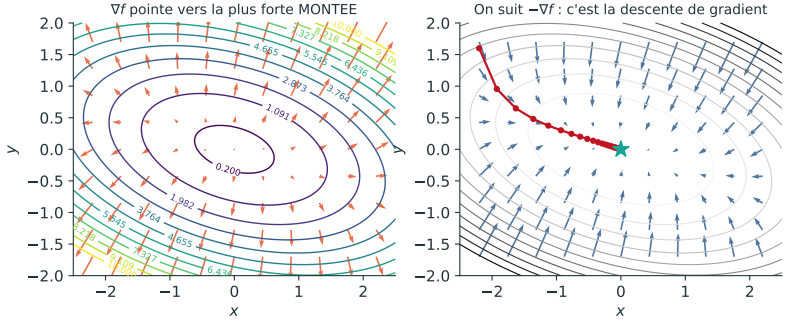
$\frac{\partial f}{\partial y} = x - 2$ (x est une constante)

Au point $(1, 2)$:

$$\nabla f(1, 2) = (6 \cdot 1 + 2,\ 1 - 2) = (8,\ -1)$$

Bouger x fait beaucoup varier f ; bouger y un peu, et en sens inverse.

3. Lire un gradient



Les deux propriétés

1. ∇f pointe dans la direction de plus forte **augmentation**.
2. ∇f est **orthogonal** aux lignes de niveau.

$\Rightarrow$ pour *minimiser*, on avance dans $-\nabla f$.

L'analogie de la carte IGN

Les lignes de niveau sont les courbes d'altitude. Le gradient est la direction « droit dans la pente ». Sa **norme** $\|\nabla f\|$ est la **raideur** : lignes serrées = grand gradient = ça descend fort.

La mise à jour

$$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta)$$

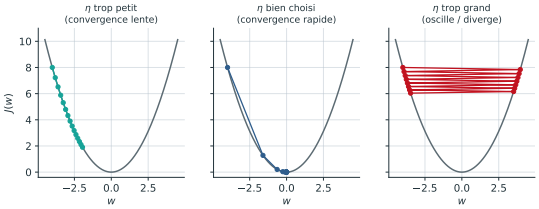
$\nabla_\theta J$ donne la **direction** ;

η donne la **longueur** du pas ;

le signe $-$ fait qu'on **descend**.

Piège classique

Le gradient est une information **locale** : il ne dit rien sur ce qui se passe plus loin. C'est pour cela qu'un pas trop grand peut sauter par-dessus le minimum, et qu'aucune garantie globale n'existe.



Pourquoi $-\nabla f$ est la meilleure direction

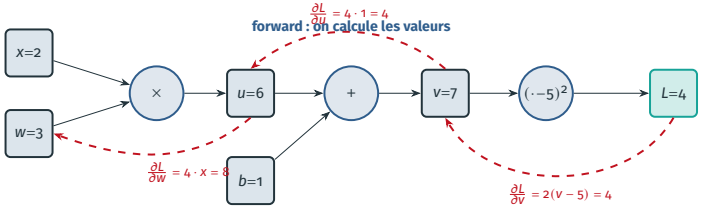
Pour un petit pas $\mathbf{h}$, $f(\theta + \mathbf{h}) \approx f(\theta) + \nabla f^\top \mathbf{h}$.

Ce produit scalaire est **le plus négatif possible** quand $\mathbf{h}$ pointe exactement à l'opposé de ∇f (séance 2, produit scalaire !). D'où la formule.

Chain rule

Si $y = f(u)$ et $u = g(x)$, alors $\dfrac{dy}{dx} = \dfrac{dy}{du} \cdot \dfrac{du}{dx}$

On **multiplie** les sensibilités le long de la chaîne. Avec L couches, on multiplie L termes — retenez ce mot, *multiplier* : il explique tout ce qui suit.



À retenir

La **rétropropagation** n'est *que* la règle de la chaîne appliquée systématiquement, de la sortie vers l'entrée, en réutilisant les calculs déjà faits. Rien de plus.

6. Rétropropagation à la main

Backward — on remonte, terme par terme

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - y) = 2(0.7 - 1) = \mathbf{-0.6}$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot a_1 = -0.6 \times 0.5 = \mathbf{-0.30}$$

$$\frac{\partial \mathcal{L}}{\partial b_2} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot 1 = \mathbf{-0.60}$$

$$\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot w_2 = -0.6 \times 2 = \mathbf{-1.20}$$

$$\frac{\partial \mathcal{L}}{\partial z_1} = \frac{\partial \mathcal{L}}{\partial a_1} \cdot \mathbb{1}_{z_1 > 0} = \mathbf{-1.20}$$

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial z_1} \cdot x = \mathbf{-1.20}$$

La mise à jour, avec $\eta = 0.1$

$$\begin{aligned} w_1 &\leftarrow 0.5 - 0.1 \times (-1.20) = \mathbf{0.62} \\ b_1 &\leftarrow 0 - 0.1 \times (-1.20) = \mathbf{0.12} \\ w_2 &\leftarrow 2 - 0.1 \times (-0.30) = \mathbf{2.03} \\ b_2 &\leftarrow -0.3 - 0.1 \times (-0.60) = \mathbf{-0.24} \end{aligned}$$

Ça marche ?

Nouveau passage avant : $z_1 = 0.74$, $\hat{y} = 1.262$.

$$\mathcal{L} : 0.090 \longrightarrow 0.069$$

La loss a bien **baissé**. Répétez ce calcul quelques milliers de fois : c'est, littéralement, tout l'entraînement d'un réseau de neurones.

Une couche dense

Passage avant : $\mathbf{z} = W\mathbf{a} + \mathbf{b}$, puis $\mathbf{a}' = g(\mathbf{z})$.

On note $\delta = \dfrac{\partial \mathcal{L}}{\partial \mathbf{z}}$ (le gradient *à la sortie* de la couche, avant l'activation).

Les trois formules

$$\frac{\partial \mathcal{L}}{\partial W} = \delta\, \mathbf{a}^\top \qquad (h \times 1)(1 \times d) = (h \times d)\ \checkmark$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \delta \qquad (h \times 1)\ \checkmark$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}} = W^\top \delta \qquad (d \times h)(h \times 1) = (d \times 1)\ \checkmark$$

La récurrence de la rétropropagation

$$\delta^{(\ell-1)} = \left(W^{(\ell)\top} \delta^{(\ell)}\right) \odot g'\left(\mathbf{z}^{(\ell-1)}\right)$$

On remonte couche par couche : une **transposée** pour renvoyer le gradient en arrière, un **produit terme à terme** par la dérivée de l'activation.

Le réflexe qui sauve

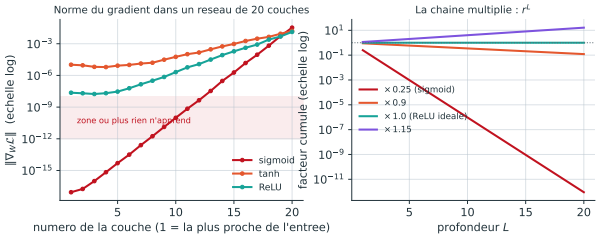
Le gradient d'un objet a **toujours la même shape que l'objet**.

W est $(h \times d) \Rightarrow$ `W.grad` est $(h \times d)$. Si vos shapes ne collent pas, c'est là qu'il faut chercher — pas dans les maths.

Intuition

La transposée $W^\top$ n'a rien de mystique : c'est la même matrice, lue « à l'envers », pour renvoyer l'information de la sortie vers l'entrée.

8. Vanishing / exploding



Le mécanisme

La règle de la chaîne **multiplie** L facteurs. Si chacun vaut en moyenne r, le gradient de la première couche est multiplié par r^L :

$r < 1 \Rightarrow$ **vanishing** : les premières couches n'apprennent plus.

$r > 1 \Rightarrow$ **exploding** : la loss devient `NaN`.

Ce que montre la mesure

Un vrai réseau de 20 couches, un seul `backward`. Avec la **sigmoid** (dérivée ≤ 0.25), le gradient de la couche 1 vaut $9 \cdot 10^{-18}$ contre $3 \cdot 10^{-2}$ à la couche 20 : un rapport de $\mathbf{4 \cdot 10^{15}}$, la première couche est gelée. Avec **ReLU**, le rapport tombe à $\sim 10^5$ — **dix ordres de grandeur** de mieux.

Les remèdes, tous vus dans ce cours

ReLU plutôt que sigmoid • **BatchNorm** (séance 3) • **connexions résiduelles** : le $+\mathbf{x}$ ajoute un $+1$ à la dérivée et crée une autoroute pour le gradient (séance 4) • bonne **initialisation** (He, Xavier) • **gradient clipping** contre l'explosion.

```python
import torch

x = torch.tensor(1.0)
w1 = torch.tensor(0.5, requires_grad=True)
b1 = torch.tensor(0.0, requires_grad=True)
w2 = torch.tensor(2.0, requires_grad=True)
b2 = torch.tensor(-0.3, requires_grad=True)

z1 = w1 * x + b1            # 0.5
a1 = torch.relu(z1)         # 0.5
yh = w2 * a1 + b2           # 0.7
loss = (yh - 1.0) ** 2      # 0.09

loss.backward()             # remonte tout le graphe

print(w1.grad, b1.grad)     # -1.2000  -1.2000
print(w2.grad, b2.grad)     # -0.3000  -0.6000
```

✔ Exactement les valeurs calculées à la main deux diapositives plus tôt.

Intuition

PyTorch construit le **graphe de calcul** pendant le passage avant, puis le parcourt à l'envers au `.backward()`. Vous n'écrivez *jamais* de dérivée à la main — mais vous devez savoir ce qu'il fait pour débugger.

⚠ Trois conséquences pratiques

- Les gradients s'**accumulent** ⇒ `opt.zero_grad()` à chaque itération, sinon vous additionnez les batches.
- Le graphe consomme de la mémoire ⇒ `with torch.no_grad():` en évaluation.
- `requires_grad=False` gèle un paramètre : c'est tout le mécanisme du **transfer learning** (séance 4).

⚠ Le gradient est nul ou minuscule

- **Saturation** : sigmoid/tanh loin de 0, dérivée ≈ 0.
- **Dead ReLU** : le neurone sort toujours 0, donc $g' = 0$, donc plus aucun gradient ne passe — définitivement.
- Trop de couches sans BatchNorm ni skip.
- `requires_grad=False` oublié quelque part.
- Loss détachée du graphe (`.item()`, `.detach()`, conversion en `numpy`).

⚠ Le gradient explose

- Learning rate trop grand.
- Entrées non normalisées (pixels 0–255).
- Mauvaise initialisation (poids trop grands).
- Division par un nombre proche de 0 dans la loss, ou `log(0)`.

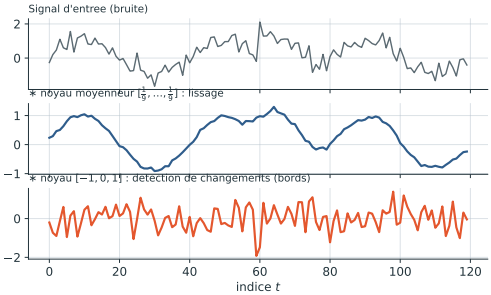
Remède immédiat : `clip_grad_norm_(params, 1.0)` et $\eta/10$.

🛠 Le diagnostic en deux lignes

```
for n, p in model.named_parameters():
    if p.grad is not None:
        print(n, p.grad.norm().item())
```

Des normes à 10^{-9} ou à `nan` : vous avez trouvé le coupable.

La convolution



En 1D — ce que fait un CNN

$$(x \star k)[t] = \sum_{m=-M}^{M} k[m]\, x[t+m]$$

On **glisse** le noyau et, à chaque position, on calcule un **produit scalaire**.

En 2D

$$S[i,j] = \sum_{m} \sum_{n} K[m,n]\; I[i+m,\; j+n]$$

Noyau 3×3 : 9 multiplications par pixel de sortie.

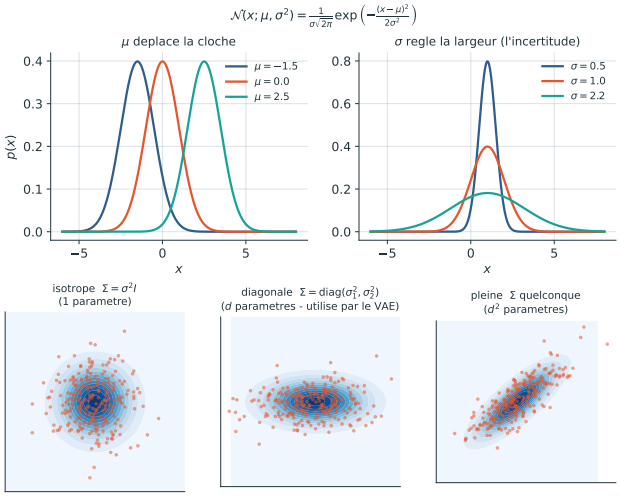
Les deux propriétés qui font tout

1. **Linéarité** $\Rightarrow$ dérivable, donc entraînable par gradient.
2. **Invariance par translation** : si l'objet se déplace, la réponse se déplace pareil. Le filtre le reconnaît *où qu'il soit*.

Le petit mensonge de vocabulaire

La vraie convolution retourne le noyau. Les bibliothèques implémentent la **corrélation croisée** et l'appellent « convolution » : comme le noyau est *appris*, le retournement n'a aucune importance.

Loi normale



Définition

Deux paramètres suffisent : μ (**où**) et σ^2 (**à quel point c'est flou**). En dimension k : $\mathbf{z} \sim \mathcal{N}(\mu, \Sigma)$.

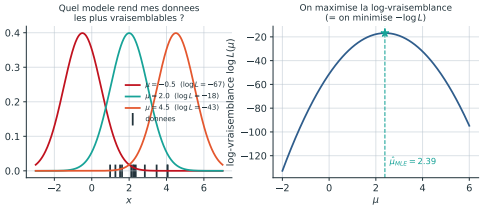
En pratique

L'espace latent de la séance 4 utilise toujours une covariance **diagonale** : k nombres au lieu de k^2, et des dimensions indépendantes donc interprétables une à une.

Intuition

Retenez l'idée : un encodeur peut sortir non pas un *point*, mais une **zone d'incertitude** — un μ et un σ.

Vraisemblance et KL

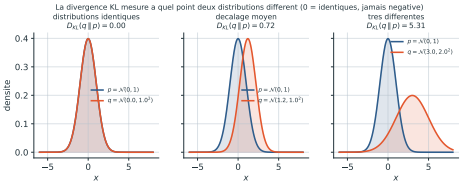


Maximum de vraisemblance

$$\hat{\theta} = \arg\min_{\theta} \; -\sum_{i=1}^{n} \log p_\theta(x_i)$$

La révélation

Bruit gaussien $\Rightarrow$ $-\log p$ donne la **MSE**. Sortie binaire $\Rightarrow$ $-\log p$ donne la **cross-entropy**. Les pertes de la séance 1 n'étaient pas arbitraires.



Définition

$$D_{\mathrm{KL}}(q \,\|\, p) = \int q(z) \log \frac{q(z)}{p(z)} \, dz \;\geq\; 0$$

Vaut 0 **si et seulement si** $q = p$. Ce n'est **pas** une distance : $D_{\mathrm{KL}}(q\|p) \neq D_{\mathrm{KL}}(p\|q)$.

Le cas qui servira en séance 4

Entre $q = \mathcal{N}(\mu, \sigma^2)$ et $p = \mathcal{N}(0, 1)$, pas d'intégrale à calculer :

$$D_{\mathrm{KL}} = \tfrac{1}{2}\left(\mu^2 + \sigma^2 - \log \sigma^2 - 1\right)$$

Le gradient — l'essentiel

1. ∇f = vecteur des dérivées partielles ; il pointe vers la **montée**, on va à l'opposé.
2. Le gradient a **toujours la shape** de l'objet qu'il dérive.
3. La **règle de la chaîne** multiplie les sensibilités : c'est la rétropropagation.
4. Une couche dense : $\partial \mathcal{L} / \partial W = \delta \, \mathbf{a}^\top$, et on remonte avec $W^\top \delta$.
5. Multiplier L termes $\Rightarrow$ **vanishing / exploding** ; remèdes : ReLU, BatchNorm, skips, init, clipping.

Le reste de la boîte à outils

- Une image = un **tenseur** (C, H, W) ; vérifiez la *shape*.
- Le **produit scalaire** mesure une ressemblance : neurone et filtre.
- Une **matrice** transforme l'espace ; son **rang** dit ce qui survit.
- **SVD / PCA** : une représentation compacte existe.
- La **convolution** = glisser un noyau.
- La **KL** mesure l'écart entre deux lois, et vaut 0 si elles sont égales.

Intuition

Vous avez maintenant *tout* l'outillage. La suite n'est que de l'assemblage.

Séance 3 — Les CNN

Objectifs

À la fin de cette séance, vous saurez

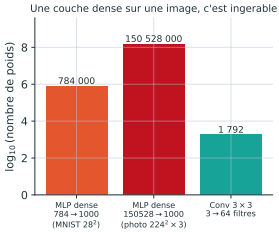
- expliquer **pourquoi** un MLP échoue sur des images ;
- calculer la **taille de sortie** et le **nombre de paramètres** d'une couche de convolution ;
- dire à quoi servent `kernel_size`, `stride`, `padding`, `out_channels`, le **pooling** et la **BatchNorm** ;
- lire une **architecture** de CNN et comprendre ce que chaque couche extrait ;
- entraîner un CNN et **diagnostiquer** ce qui ne va pas.

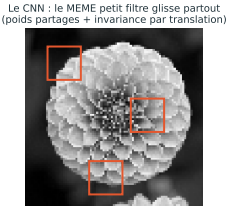
Plan

1. Pourquoi pas un MLP ?
2. Manipuler des images
3. La convolution en pratique
4. Les briques d'un CNN
5. L'architecture type
6. Ce que le réseau apprend
7. Entraîner et diagnostiquer

Les architectures avancées (ResNet, transfer learning) ouvrent la séance 4.

Explosion des paramètres





⚠ 1. Le nombre de paramètres

Une image $224 \times 224 \times 3$ vers une couche de 1000 neurones :

$$150\,528 \times 1000 = \textbf{150 millions} \text{ de poids}$$

pour une seule couche. Impossible à entraîner, mémoire explosée, overfitting garanti.

⚠ 2. On détruit la structure

Le `flatten` met les pixels $(0,0)$ et $(0,1)$ côte à côte, mais $(0,0)$ et $(1,0)$ à 224 cases d'écart — alors qu'ils sont **voisins** dans l'image. Le MLP doit *réapprendre* la notion de voisinage.

⚠ 3. Aucune invariance par translation

Un MLP qui a appris à reconnaître un chat en haut à gauche ne le reconnaît pas en bas à droite : ce sont d'autres poids.

Trois idées

1. Connectivité locale

Un neurone ne regarde qu'une petite **fenêtre** (3×3, 5×5), pas toute l'image.

$\Rightarrow$ 9 poids au lieu de 150 000.

2. Partage de poids

Le **même** filtre est appliqué à **toutes** les positions de l'image.

$\Rightarrow$ invariance par translation *gratuite*, et encore moins de paramètres.

3. Hiérarchie

On empile : bords $\rightarrow$ textures $\rightarrow$ motifs $\rightarrow$ objets.

$\Rightarrow$ des features de plus en plus **abstraites** et un **champ réceptif** de plus en plus large.

MLP dense
chaque neurone voit *tout*
$\Rightarrow$ 150 528 $\times$ 1000 poids

$\div$ 84 000 $\rightarrow$

Couche convolutive
64 filtres 3×3 sur 3 canaux
$\Rightarrow 3 \cdot 3 \cdot 3 \cdot 64 + 64 = \mathbf{1792}$ poids

Opérations de base

originale 340 × 340 | crop (recadrage) | resize ÷4 (85×85) | flip horizontal | rotation 22 ° | luminosite ×1.5

Préparer une image pour un réseau

1. **Redimensionner** à une taille fixe (le réseau attend une entrée fixe).
2. Convertir en `float32`.
3. **Normaliser** : `img/255` $\longrightarrow [0,1]$, puis souvent $(x-\mu)/\sigma$ par canal.
4. Réordonner en `(C,H,W)`.

Trois pièges d'images

- Oublier `/255` : les activations explosent dès la première couche.
- Normaliser le **train** et pas le **test** (ou avec d'autres μ, σ).
- Inverser **BGR** (OpenCV) et **RGB** (PIL) : le modèle voit des couleurs fausses et perd plusieurs points d'accuracy sans erreur visible.

Data augmentation

Data augmentation : 1 image annotee → des dizaines d'exemples (le label ne change pas)

ORIGINALE

Intuition

On apprend au réseau que « un chat tourné de 15°, plus sombre et décalé, est toujours un chat ». C'est la régularisation **la plus efficace** en vision.

L'augmentation doit préserver le label

- Flip horizontal sur des chats : ✓
- Flip horizontal sur des **chiffres** : ✗ (un 2 miroir n'est pas un 2)
- Flip **vertical** sur des photos de rue : ✗
- Rotation de 90° sur des images satellite ou histologiques : ✓

Et surtout : on augmente **uniquement le train**, jamais la validation ni le test.

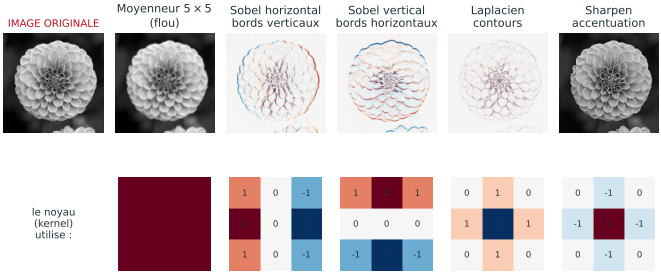
En pratique : `torchvision`

```python
from torchvision import transforms as T

train_tf = T.Compose([
    T.RandomResizedCrop(224, scale=(0.7, 1.0)),
    T.RandomHorizontalFlip(p=0.5),
    T.RandomRotation(15),
    T.ColorJitter(brightness=0.25, contrast=0.25),
    T.ToTensor(),                          # -> [0,1], (C,H,W)
    T.Normalize(mean=[0.485, 0.456, 0.406],
                std =[0.229, 0.224, 0.225]),
])

# PAS d'aléatoire en validation / test !
eval_tf = T.Compose([
    T.Resize(256), T.CenterCrop(224),
    T.ToTensor(),
    T.Normalize(mean=[0.485, 0.456, 0.406],
                std =[0.229, 0.224, 0.225]),
])
```

À savoir

- L'augmentation est appliquée **à la volée** : chaque epoch voit une version différente de la même image.
- Les valeurs `mean`/`std` ci-contre sont celles d'**ImageNet** : on les garde si on part d'un modèle pré-entraîné.
- Pour la **segmentation**, l'augmentation géométrique doit être appliquée **à l'identique** à l'image *et* au masque (cf. séance 4).

Filtres appris



Avant

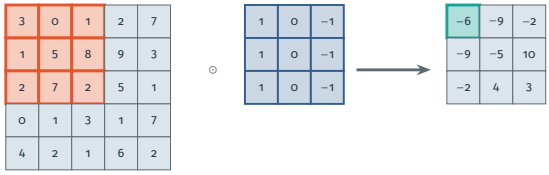
Pendant 40 ans, la vision par ordinateur a consisté à **concevoir** ces noyaux (Sobel, Gabor, SIFT, HOG…) puis à mettre un classifieur au bout. Cela marchait… jusqu'à un certain point.

Le basculement conceptuel de toute la séance

Les 9 nombres d'un noyau 3×3 ne sont plus décidés par un humain : ce sont des **paramètres** θ, initialisés au hasard et ajustés par **descente de gradient** — exactement comme les poids $\mathbf{w}$ d'une régression linéaire, et avec la mécanique de la séance 2. Le réseau ne « voit » pas les images : il ajuste des nombres pour faire baisser une loss. Que ces nombres finissent par ressembler à des détecteurs de bords est un *résultat*, pas une consigne.

Convolution pas à pas

Entrée 5×5 **Noyau** 3×3 **Sortie** 3×3

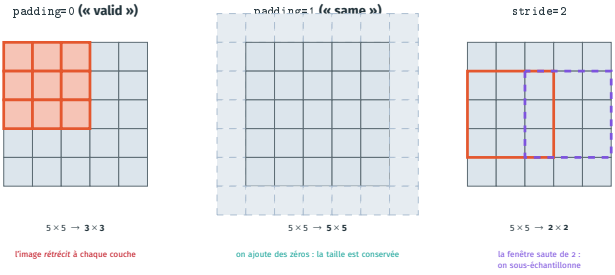


$$(3{\cdot}1 + 0{\cdot}0 + 1{\cdot}(-1)) + (1{\cdot}1 + 5{\cdot}0 + 8{\cdot}(-1)) + (2{\cdot}1 + 7{\cdot}0 + 2{\cdot}(-1)) = 2 - 7 - 0 \Rightarrow \mathbf{-6}$$

Intuition

On glisse la fenêtre orange sur toutes les positions possibles. À chaque position : multiplication terme à terme, puis somme. **Un produit scalaire**, comme en séance 2.

Padding et stride



La formule à connaître par cœur

$$O = \left\lfloor \frac{I + 2P - K}{S} \right\rfloor + 1$$

I = taille d'entrée, K = `kernel_size`, P = `padding`, S = `stride`, O = taille de sortie.

Cas particulier utile : K impair, $P = (K-1)/2$, $S = 1 \Rightarrow O = I$ (« same »).

Question

Entrée 224×224, `Conv2d(3, 64, kernel_size=7, stride=2, padding=3)`.

a) Quelle est la taille de sortie ?
b) Combien de paramètres a cette couche ?

Question

Entrée `224 × 224`, `Conv2d(3, 64, kernel_size=7, stride=2, padding=3)`.

a) Quelle est la taille de sortie ?
b) Combien de paramètres a cette couche ?

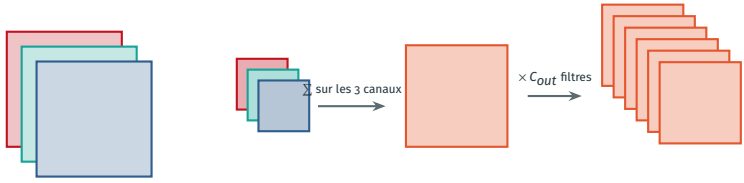
Réponse

a) $O = \left\lfloor \frac{224+6-7}{2} \right\rfloor + 1 = \lfloor 111.5 \rfloor + 1 = \mathbf{112}$, soit $\mathbf{112} \times \mathbf{112} \times \mathbf{64}$.

b) $\underbrace{7 \times 7 \times 3}_{\text{un filtre}} \times \underbrace{64}_{\text{filtres}} + \underbrace{64}_{\text{biais}} = \mathbf{9\,472}$.

(C'est exactement la première couche de ResNet-50.)

Plusieurs canaux



Le compte des paramètres

$$\#\theta = \underbrace{K \times K \times C_{in} \times C_{out}}_{\text{poids}} + \underbrace{C_{out}}_{\text{biais}}$$

Indépendant de H et W ! C'est le partage de poids : la même couche marche sur du 64×64 ou du 512×512.

L'erreur de vocabulaire à éviter

Un « filtre 3×3 » sur une entrée à 64 canaux est en fait un tenseur $3 \times 3 \times 64$, soit 576 poids — pas 9.

Fiches : convolution

kernel_size (K)

Rôle : la taille de la fenêtre : combien de pixels voisins le filtre regarde d'un coup.

↓ Trop petit : champ réceptif minuscule, il faut beaucoup de couches. **↑ Trop grand :** beaucoup de paramètres et de calcul ; détails fins perdus.

✓ Valeur usuelle : `3 presque partout ; 7 ou 5 seulement en toute première couche`

stride (S)

Rôle : le pas de déplacement de la fenêtre ; $S > 1$ *réduit* la résolution.

↓ Trop petit : $S = 1$: résolution conservée, calcul plus lourd. **↑ Trop grand :** on saute des pixels, perte d'information spatiale.

✓ Valeur usuelle : `1 (avec du pooling pour réduire) ; 2 pour remplacer le pooling`

padding (P)

Rôle : ajoute des zéros au bord pour éviter que l'image rétrécisse et pour ne pas sous-exploiter les pixels du bord.

↓ Trop petit : $P = 0$: l'image rétrécit à chaque couche, effets de bord. **↑ Trop grand :** beaucoup de zéros artificiels aux bords.

✓ Valeur usuelle : $P = (K-1)/2$ `pour garder la taille (padding='same')`

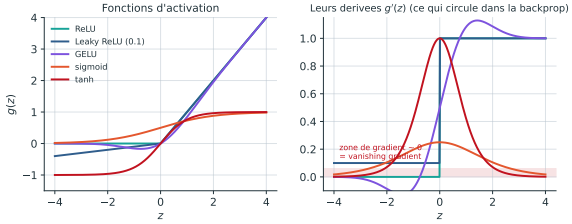
out_channels (C_{out})

Rôle : le nombre de filtres, donc le nombre de motifs *différents* détectés par cette couche. C'est la *largeur* du réseau.

↓ Trop petit : le réseau ne peut détecter que peu de motifs : underfitting. **↑ Trop grand :** coût mémoire et calcul, overfitting.

✓ Valeur usuelle : `32 → 64 → 128 → 256 : on double à chaque réduction de résolution`

Fonctions d'activation



Pourquoi elles sont indispensables

Sans activation, empiler deux couches donne $W_2(W_1\mathbf{x}) = (W_2W_1)\mathbf{x}$: **une seule** couche linéaire. Toute la profondeur serait perdue.

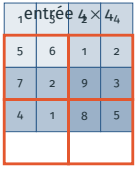
Que choisir

- **ReLU** : le défaut, partout dans les couches cachées. Simple, rapide.
- **Leaky ReLU / GELU** : si des neurones « meurent » (bloqués à 0).
- **sigmoid** : *uniquement* en sortie binaire.
- **softmax** : *uniquement* en sortie multi-classes.
- **tanh** : rare aujourd'hui (sauf sorties dans $[-1, 1]$).

Vanishing gradient — rappel

La dérivée de la sigmoïde ne dépasse jamais 0.25 : sur L couches le gradient est multiplié par $\leq 0.25^L$. C'est la courbe mesurée en séance 2. La ReLU (dérivée = 1) a rendu les réseaux profonds entraînables.

Pooling



Max pool

6	4
7	9

Avg pool

3.75	2.25
3.5	6.25

Max-pooling 2×2, stride 2

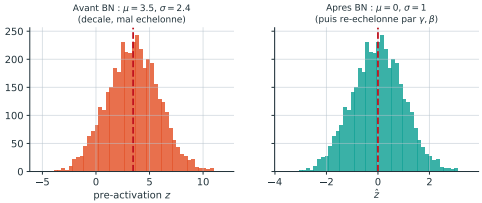
On garde le **maximum** de chaque bloc 2×2. La taille est **divisée par 2** en hauteur et en largeur (donc par 4 en pixels). **Aucun paramètre à apprendre.**

À quoi ça sert

1. Réduire le calcul et la mémoire.
2. Élargir le **champ réceptif** plus vite.
3. Apporter une **invariance locale** : peu importe où exactement, dans le bloc 2×2, le motif a été détecté.

Max-pooling 2 × 2 repete : on perd la resolution, on garde l'information saillante

Batch Normalization



Définition

Pour chaque canal, sur le batch :

$$\hat{z} = \frac{z - \mu_{\text{batch}}}{\sqrt{\sigma^2_{\text{batch}} + \epsilon}}, \qquad y = \gamma \hat{z} + \beta$$

γ et β sont **appris** : le réseau garde le droit de « dénormaliser » s'il en a besoin.

Ce que ça apporte

entraînement plus **rapide** et plus **stable** • permet des learning rates plus grands • réduit la sensibilité à l'initialisation • effet régularisant léger.

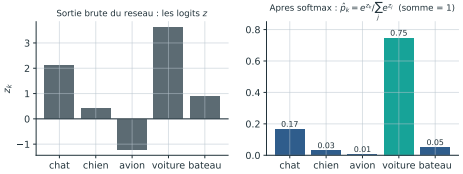
Train ≠ eval

En **entraînement**, BN utilise les statistiques *du batch courant*. En **évaluation**, elle utilise une *moyenne mobile* accumulée pendant l'entraînement. D'où l'obligation d'appeler `model.eval()` — et le fait qu'un `batch_size` de 2 ou 4 rend la BN instable (préférez alors `GroupNorm`).

Tête de classification

Deux façons de « terminer » un CNN

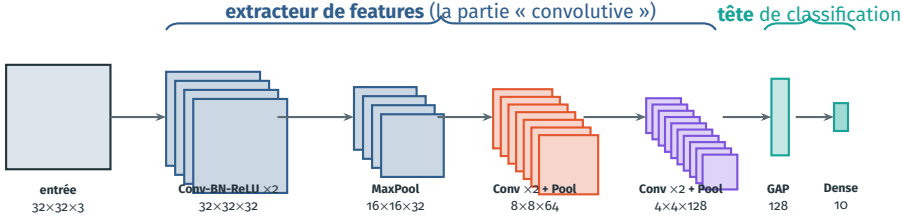
- **Flatten + Dense** : on aplatit le volume (C, H, W) en un vecteur de $C \cdot H \cdot W$ valeurs, puis couches denses.
  → beaucoup de paramètres, taille d'entrée figée.
- **Global Average Pooling** : on moyenne chaque canal sur toute sa carte, donnant C valeurs.
  → presque aucun paramètre, accepte n'importe quelle taille d'entrée.



Piège classique

En PyTorch, `nn.CrossEntropyLoss` applique **déjà** le `log_softmax`. Donnez-lui les **logits** bruts. Ajouter un `Softmax` avant est une erreur fréquente qui dégrade silencieusement l'apprentissage.

Architecture type



Le motif universel

$$\underbrace{\left[\text{Conv} \to \text{BN} \to \text{ReLU}\right]^{\times n}}_{\text{bloc}} \to \text{Pool}$$

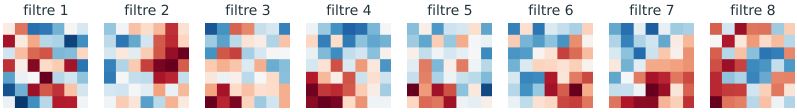
répété 3 à 5 fois, en **doublant les canaux** à chaque fois qu'on **divise la résolution par 2**.

Intuition

La résolution **diminue** (moins de « où ») pendant que le nombre de canaux **augmente** (plus de « quoi »). Le réseau troque progressivement de l'**information spatiale** contre de l'**information sémantique**.

La première couche

Les 8 filtres 7 × 7 APPRIS par la 1re couche (personne ne les a concus : ils sortent de la descente de gradient)



D'où viennent ces images

Un petit CNN (3 blocs convolutifs) entraîné sur un jeu de formes géométriques bruitées — cercles, carrés, triangles. Après 25 epochs, il atteint **100 % d'accuracy**. Ci-dessus : les 8 noyaux 7 × 7 de sa première couche, tels qu'ils sont après entraînement.

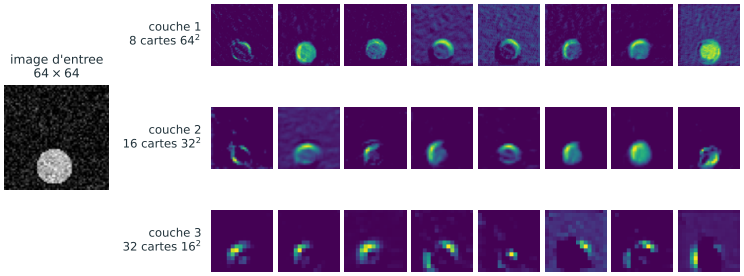
Ce qu'on y voit — et ce qu'on n'y voit pas

Plusieurs filtres montrent une **orientation** nette — rouge d'un côté, bleu de l'autre : ce sont des **détecteurs de bords**, de la même famille que le Sobel écrit à la main. Personne ne les a programmés : la descente de gradient les a **trouvés seule**.

Mais soyons honnêtes : ils restent **bruités**, loin des jolis noyaux de manuel. La tâche est facile et le réseau minuscule — il atteint 100 % bien avant d'avoir besoin de filtres propres. Les filtres nets « gabor-like » des articles viennent de réseaux profonds entraînés sur des millions d'images.

Feature maps

Feature maps : chaque filtre produit une carte d'activation. Plus on descend, plus c'est petit et abstrait.



Feature map

La sortie d'*un* filtre : une carte 2D qui indique **où**, dans l'image, le motif recherché par ce filtre est présent (et avec quelle intensité).

La hiérarchie

couche 1 : bords, contrastes → couche 2 : coins, textures → couches profondes : parties d'objets → dernière couche : concepts.
La résolution baisse ($64^2 \to 32^2 \to 16^2$), le nombre de cartes monte ($8 \to 16 \to 32$).

CNN en PyTorch

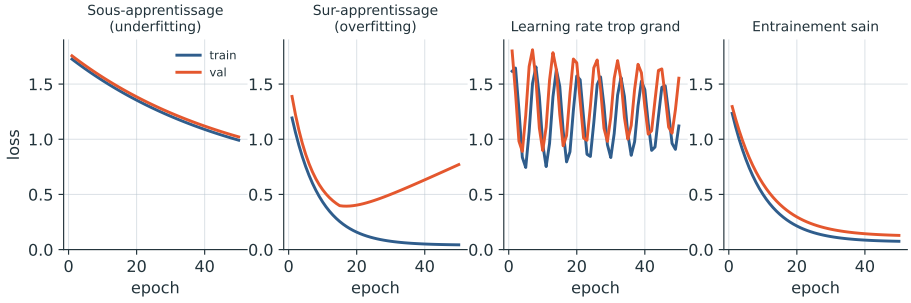
```python
import torch.nn as nn

def bloc(c_in, c_out):                  # Conv-BN-ReLU x2, puis Pool
    return nn.Sequential(
        nn.Conv2d(c_in, c_out, 3, padding=1),
        nn.BatchNorm2d(c_out),
            nn.ReLU(inplace=True),
        nn.Conv2d(c_out, c_out, 3, padding=1),
        nn.BatchNorm2d(c_out),
            nn.ReLU(inplace=True),
        nn.MaxPool2d(2))                # H, W divisés par 2

model = nn.Sequential(
    bloc(3,   32),                      # 32x32 -> 16x16
    bloc(32,  64),                      # 16x16 ->  8x8
    bloc(64, 128),                      #  8x8  ->  4x4
    nn.AdaptiveAvgPool2d(1),            # -> (128, 1, 1)
    nn.Flatten(), nn.Dropout(0.3),
    nn.Linear(128, 10))                 # 10 classes (logits)
# ~ 300 000 paramètres
```

```python
opt   =
    torch.optim.AdamW(model.parameters(),
                      lr=1e-3,
                            weight_decay=1e-4)
sched =
    torch.optim.lr_scheduler.CosineAnnealingLR(
                      opt,
                            T_max=n_epochs)
lossf = nn.CrossEntropyLoss()   # attend des LOGITS

for epoch in range(n_epochs):
    model.train()
    for xb, yb in train_loader:
        xb, yb = xb.to(dev), yb.to(dev)
        opt.zero_grad()
        loss = lossf(model(xb), yb)
        loss.backward()
        opt.step()
    sched.step()                        # 1 fois par epoch

    model.eval()
    with torch.no_grad():
        acc = accuracy(model, val_loader)
    print(epoch, round(acc, 4))
```

Deux remarques

- C'est **exactement** la boucle de la séance 1 ; seul le modèle a changé.
- Le *scheduler* `CosineAnnealingLR` fait décroître η : **grands pas** au début, **petits pas** à la fin. Un `sched.step()` par epoch, quelques points gagnés gratuitement.

Lire les courbes



🔑 Toujours tracer train ET validation, à chaque epoch

Une seule courbe ne dit rien. C'est l'**écart** entre les deux qui vous renseigne, et sa **tendance** qui vous dit quand arrêter.

⚠ La loss ne bouge pas du tout

- Learning rate trop petit (ou trop grand !).
- Oubli de `opt.zero_grad()` ou de `loss.backward()`.
- Images non normalisées (valeurs 0–255).
- Un `Softmax` en trop avant `CrossEntropyLoss`.
- Labels décalés (1..10 au lieu de 0..9).

⚠ La loss devient `NaN`

learning rate trop grand • division par zéro dans la loss • `log(0)` • données contenant des `NaN`. *Remède immédiat :* diviser η par 10, ajouter du `gradient clipping`.

🛠 Le protocole de débogage, dans l'ordre

1. **Affichez un batch** : images et labels. Sont-ils corrects ? À l'endroit ?
2. **Surapprenez 10 images** sans augmentation ni régularisation. Vous devez atteindre 100 %. Sinon : bug.
3. Vérifiez la loss initiale : pour K classes équilibrées, elle doit valoir $\approx \log K$ (≈ 2.30 pour 10 classes). Sinon : problème d'initialisation ou de labels.
4. Faites tourner 1 epoch sur 1 % des données pour vérifier la mécanique.
5. *Ensuite* seulement, lancez le vrai entraînement.

Récapitulatif — séance 3

Les mécanismes

1. Un CNN = **connectivité locale** + **partage de poids** + **hiérarchie**.
2. Une convolution glisse un noyau et calcule des **produits scalaires**.
3. $O = \lfloor (I + 2P - K)/S \rfloor + 1$ et $\#\theta = K^2 C_{in} C_{out} + C_{out}$.
4. La résolution baisse, le nombre de canaux monte.
5. Les filtres sont **appris**, pas conçus — et ils redécouvrent les détecteurs de bords.

La méthode, inchangée depuis la séance 1

- Même **split**, mêmes **métriques**, même discipline.
- Mêmes **hyperparamètres** d'optimisation, plus ceux de la conv.
- Même **overfitting**, avec une arme en plus : la **data augmentation**.
- Même **rétropropagation** qu'en séance 2 : un noyau de convolution est un paramètre comme un autre.

La suite

La séance 4 part de là et va dans deux directions : **plus profond** (ResNet, transfer learning), puis **plus loin que la classification** — on remplace la tête par un **décodeur** qui reconstruit une image.

Séance 4 — U-Net et espace latent

Objectifs

À la fin de cette séance, vous saurez

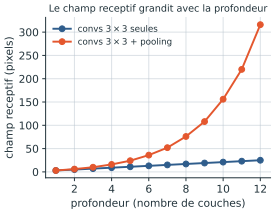
- expliquer ce qu'un neurone profond « voit » (**champ réceptif**) et pourquoi les **connexions résiduelles** rendent la profondeur possible ;
- réutiliser un modèle pré-entraîné (**transfer learning**) et choisir quoi geler ;
- lire l'architecture d'un **U-Net** et dire pourquoi ses **skip connections** sont indispensables ;
- comprendre ce qu'est un **espace latent**, comment on le construit et comment on le **visualise**.

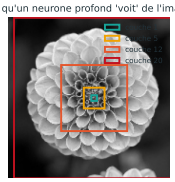
Plan

1. **CNN avancé**
   champ réceptif, ResNet, transfer learning
2. **U-Net et segmentation**
   remonter en résolution, skips, Dice
3. **L'espace latent**
   auto-encodeur, goulot, distribution latente

Une seule idée relie les trois : on garde l'extracteur convolutif, on change ce qu'on met au bout.

Champ réceptif





Définition

Le **champ réceptif** d'un neurone est la région de l'**image d'entrée** qui influence sa valeur.

Intuition

Deux convolutions 3×3 empilées « voient » 5×5, mais avec $2 \times 9 = 18$ poids au lieu de 25 — et une non-linéarité en plus. C'est pour cela que VGG a remplacé les gros noyaux par des empilements de 3×3.

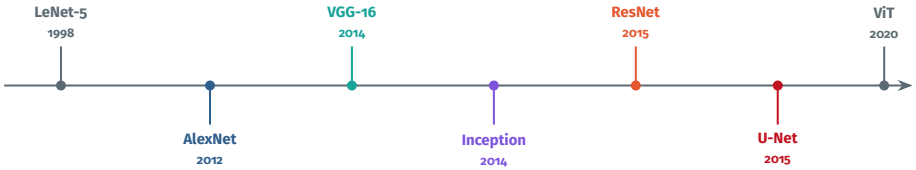
Les trois leviers pour l'agrandir

profondeur (empiler des couches) • **pooling / stride** (chaque division par 2 double le champ) • **convolution dilatée** (`dilation=2` : le noyau saute un pixel sur deux, champ élargi à coût constant).

Piège classique

Pour classifier un objet, il faut que le champ réceptif des dernières couches **couvre l'objet entier**. Un réseau trop peu profond sur de grandes images ne « voit » jamais l'objet dans son ensemble.

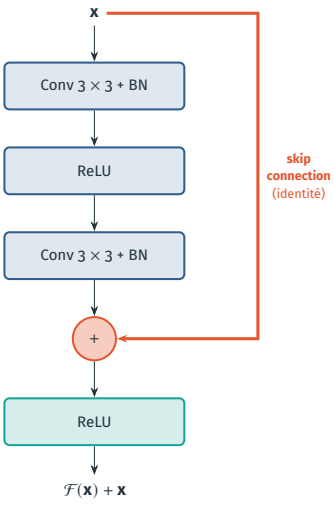
Architectures marquantes



Modèle	Couches	Params	L'idée apportée
LeNet-5	7	60 k	la formule conv → pool → dense, sur des chiffres manuscrits
AlexNet	8	60 M	ReLU + dropout + GPU : le déclic d'ImageNet 2012
VGG-16	16	138 M	*uniquement* des 3×3 empilés : la profondeur paie
Inception	22	5 M	plusieurs tailles de noyaux *en parallèle*
ResNet-50	50	25 M	les **connexions résiduelles** : on peut aller à 150 couches
U-Net	23	31 M	**skip connections** pour la segmentation — *dans 30 minutes*

Intuition

Notez la tendance : le nombre de paramètres *baisse* après VGG. On n'a pas gagné en empilant plus de poids, mais en trouvant de **meilleures façons de les connecter**.



Bloc résiduel

$$\text{sortie} = \mathcal{F}(\mathbf{x}) + \mathbf{x}$$

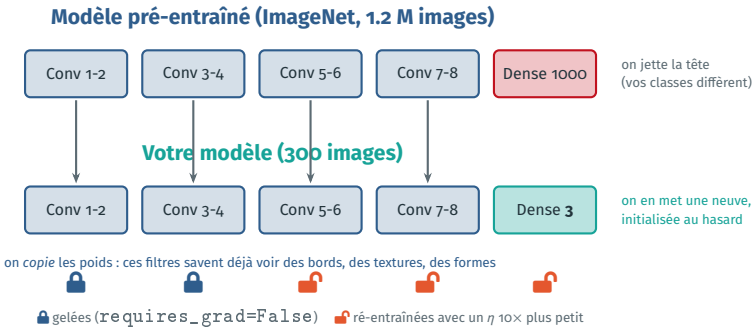
La couche n'apprend plus la transformation complète, mais seulement la **correction** $\mathcal{F}(\mathbf{x})$ à apporter à son entrée.

Pourquoi ça marche

1. Si une couche est inutile, il lui suffit d'apprendre $\mathcal{F} = 0$: ajouter des couches ne peut plus *dégrader* le réseau.
2. Le gradient a une **autoroute** : $\partial(\mathcal{F}(\mathbf{x}) + \mathbf{x})/\partial\mathbf{x} = \mathcal{F}'(\mathbf{x}) + 1$.

Le lien direct avec la séance 2

Le **vanishing gradient** vient de la *multiplication* de L facteurs < 1. Le $+1$ du bloc résiduel garantit qu'un terme au moins vaut 1 : la chaîne ne s'écrase plus. C'est *exactement* le remède annoncé sur la courbe des gradients.



Intuition

Les premières couches apprennent des features **génériques** (bords, textures) qui servent pour *n'importe quelle* image — chats, cellules, radiographies ou satellites. Seules les dernières sont **spécifiques** à la tâche d'origine. Et `requires_grad=False`, c'est simplement dire à `autograd` de ne pas calculer ce gradient-là (séance 2).

Stratégies de fine-tuning

Vos données	Proches d'ImageNet ?	Stratégie	Learning rate
Peu (< 1 000)	oui	geler tout, ré-entraîner *la tête seule*	10^{-3} sur la tête
Peu (< 1 000)	non (IRM, satellite)	geler les premières couches, fine-tuner les dernières	10^{-4}
Beaucoup (> 10 k)	oui	fine-tuner *tout* le réseau	10^{-4}, cosine
Beaucoup (> 100 k)	non	entraîner de zéro devient envisageable	10^{-3}

```python
from torchvision import models

model = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)
for p in model.parameters():
    p.requires_grad = False                # on gèle tout
model.fc = nn.Linear(model.fc.in_features, 3)   # tête neuve : dégelée par défaut

opt = torch.optim.AdamW(model.fc.parameters(), lr=1e-3)   # on n'optimise QUE la tête
```

⚠ Piège classique

Utilisez toujours un learning rate **plus petit** qu'un entraînement de zéro : sinon les premiers gradients, énormes et aléatoires (à cause de la tête neuve), détruisent les features pré-entraînées. Astuce : geler 1–2 epochs le temps que la tête se stabilise, puis tout dégeler.

Métriques par tâche

Classification
« c'est un chat »
Accuracy, F1, top-1 / top-5
séance 3

Détection
« un chat, ici »
IoU, mAP
(hors programme)

Segmentation
« ces pixels-là »
IoU, Dice
maintenant

Reconstruction
« refais-la »
MSE, inspection visuelle
en fin de séance

IoU = 0.91
Dice = 0.95

IoU = 0.49
Dice = 0.66

IoU = 0.24
Dice = 0.39

IoU = 0.04
Dice = 0.08

bleu = verite terrain Y orange = prediction $\hat{Y}$ vert = intersection

IoU — *Intersection over Union*

$$\mathrm{IoU} = \frac{|A \cap B|}{|A \cup B|} \in [0, 1]$$

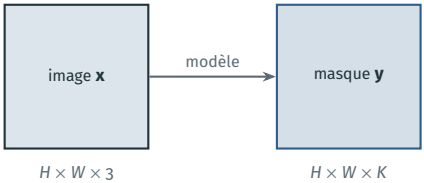
Seuil de 0.5 : la convention usuelle pour « détection correcte ».

Dice (le F_1 spatial)

$$\mathrm{Dice} = \frac{2|A \cap B|}{|A| + |B|}$$

Toujours $\geq$ IoU. La métrique standard en imagerie médicale.

Segmentation sémantique



Segmentation sémantique

sortie de *même taille* que l'entrée : une classe par pixel

C'est une classification… par pixel

Pour 512×512, c'est 262 144 classifications simultanées — mais qui doivent être **cohérentes entre elles**.

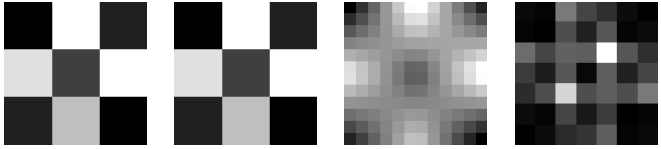
Où on en a besoin

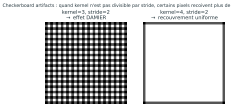
imagerie médicale (tumeurs, organes) • voiture autonome (route, piétons) • télédétection (cultures, bâtiments) • microscopie (cellules) • retouche photo (détourage).

Le dilemme

Pour dire **ce que c'est**, il faut du **contexte** ⇒ descendre en résolution (grand champ réceptif). Pour dire **où exactement**, il faut de la **précision spatiale** ⇒ ne pas descendre.

Upsampling





Deux façons de faire

Upsample + Conv : on agrandit bêtement (plus proche voisin ou bilinéaire), puis une convolution normale « répare ». ✓ simple, pas d'artefacts — le choix par défaut.

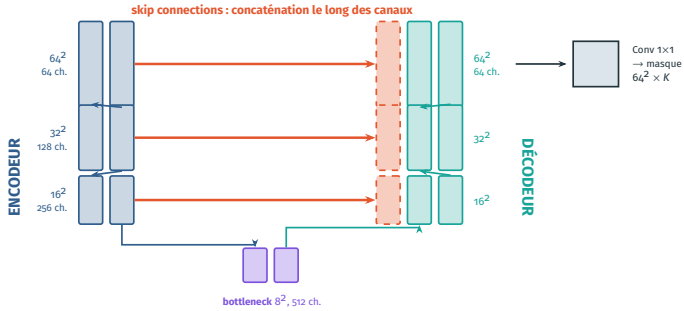
ConvTranspose2d : chaque pixel d'entrée « dépose » le noyau, pondéré, dans la sortie. L'agrandissement est *appris*. × produit des artefacts en damier.

⚠ L'artefact en damier (*checkerboard*)

Quand `kernel_size` n'est pas divisible par `stride`, les fenêtres de dépôt se **recouvrent inégalement** : certains pixels reçoivent 4 contributions, leurs voisins 2. D'où la grille visible.

Remèdes : K divisible par S (typiquement $K = 4$, $S = 2$), ou `Upsample + Conv2d`.

Architecture U-Net



La forme en U

Descente (*contracting path*) : conv + pooling, on gagne du contexte.
Remontée (*expanding path*) : upsampling + conv, on regagne la résolution.
Skips : on **concatène** la carte de l'encodeur à celle du décodeur, au même niveau.

Pourquoi les skips sont indispensables

Le bottleneck sait **quoi** (« il y a une cellule »), mais il a perdu le **où** précis : un trait de 2 pixels ne survit pas à trois divisions par 2. Les skips **réinjectent** les cartes haute résolution du début, qui contiennent les contours exacts.

Skip connections

La différence avec ResNet

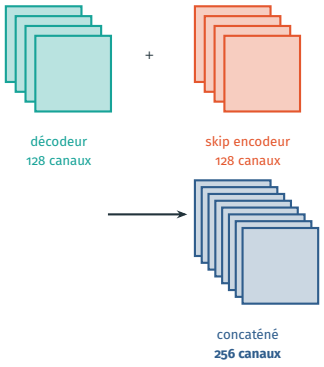
ResNet : sortie = $\mathcal{F}(\mathbf{x}) + \mathbf{x}$ — une **addition**, les canaux se mélangent, leur nombre ne change pas.

U-Net : sortie = $[\,\mathrm{up}(\mathbf{z})\ ;\ \mathrm{skip}\,]$ — une **concaténation** le long de l'axe des canaux. Les deux sources restent distinctes, et la convolution suivante **apprend comment les combiner**.

En pratique

`torch.cat([up, skip], dim=1)`

$(N, 128, 32, 32)$ et $(N, 128, 32, 32) \to (N, \mathbf{256}, 32, 32)$. D'où le `blk(4*c, 2*c)` dans le code : l'entrée du bloc décodeur a *deux fois* plus de canaux.



U-Net en PyTorch

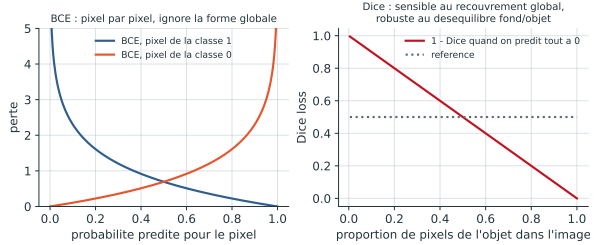
```python
# blk(i, o) = Conv-BN-ReLU x2   (voir l'auto-encodeur convolutif)
class UNet(nn.Module):
    def __init__(self, c=32, n_classes=1):
        super().__init__()
        self.e1, self.e2, self.e3 = blk(1, c), blk(c, 2*c), blk(2*c, 4*c)
        self.bott = blk(4*c, 8*c)
        self.u3, self.d3 = up(8*c, 4*c), blk(8*c, 4*c)  # 8*c = 4*c up + 4*c skip
        self.u2, self.d2 = up(4*c, 2*c), blk(4*c, 2*c)
        self.u1, self.d1 = up(2*c, c),   blk(2*c, c)
        self.out = nn.Conv2d(c, n_classes, 1)           # conv 1x1

    def forward(self, x):
        s1 = self.e1(x)                                 # 64x64
        s2 = self.e2(F.max_pool2d(s1, 2))               # 32x32
        s3 = self.e3(F.max_pool2d(s2, 2))               # 16x16
        b  = self.bott(F.max_pool2d(s3, 2))             # 8x8
        d3 = self.d3(torch.cat([self.u3(b),  s3], dim=1))
        d2 = self.d2(torch.cat([self.u2(d3), s2], dim=1))
        d1 = self.d1(torch.cat([self.u1(d2), s1], dim=1))
        return self.out(d1)                             # logits, 64x64
```

`up(i,o) = nn.ConvTranspose2d(i, o, 2, stride=2)`

⚠ Les quatre pièges

1. Taille d'entrée **divisible par $2^{\text{profondeur}}$** (ici $2^3 = 8$), sinon les shapes ne coïncident plus à la concaténation.
2. Les canaux d'entrée des blocs décodeurs sont **doublés** par la concaténation.
3. La sortie est une **conv** 1×1 : le « classifieur par pixel ».
4. On renvoie des **logits** : sigmoïde et softmax sont dans la loss.

Pertes de segmentation



Dice loss

$$\mathcal{L}_{\text{Dice}} = 1 - \frac{2\sum_i p_i y_i + \epsilon}{\sum_i p_i + \sum_i y_i + \epsilon}$$

p_i = probabilité prédite du pixel i, $y_i \in \{0, 1\}$ la vérité. Le ϵ évite la division par zéro quand le masque est vide.

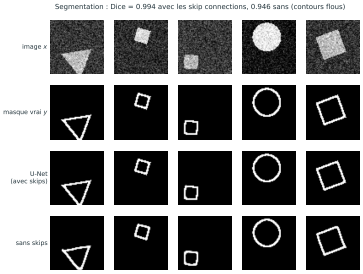
Ce qu'on utilise en pratique

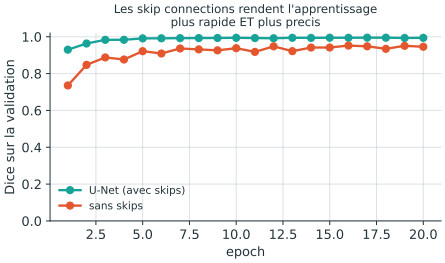
$$\mathcal{L} = \mathcal{L}_{\text{BCE}} + \mathcal{L}_{\text{Dice}}$$

La **BCE** donne des gradients stables partout ; le **Dice** optimise directement la métrique et corrige le déséquilibre fond/objet (une tumeur peut occuper 0.5 % des pixels).

⚠ **Le piège :** avec la BCE seule sur une image où l'objet occupe 1 % des pixels, le modèle « tout à zéro » obtient une loss déjà très basse et y reste coincé — **exactement** le piège de l'accuracy de la séance 1, transposé aux pixels.

Effet des skips





Le protocole

Deux réseaux **strictement identiques** (mêmes couches, mêmes paramètres, même entraînement), à une différence près : dans le second, les skip connections sont coupées. Tâche : retrouver le **contour** (2 pixels d'épaisseur) de formes bruitées.

Intuition

Sur des structures fines, le bottleneck 8×8 ne peut pas encoder la position exacte d'un trait. Sans skips, le décodeur *devine* ; avec, il *lit* la carte haute résolution de l'encodeur.

Préparer les données

- Taille d'entrée **divisible par** $2^{\text{prof.}}$ (256, 512…).
- Images trop grandes ? Découpez en **patches** avec recouvrement, puis recollez les prédictions.
- L'augmentation **géométrique** doit s'appliquer à l'image **et** au masque (même rotation, même flip). L'augmentation **colorimétrique**, à l'image seule.
- Masque en `long` pour le multi-classes, en `float` pour le binaire.

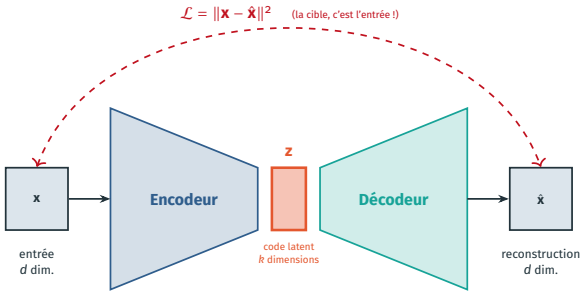
Entraîner et évaluer

- Loss : `BCEWithLogitsLoss + Dice` (binaire) ou `CrossEntropyLoss + Dice` (multi-classes).
- Métrique de suivi : **Dice** ou **IoU** sur la validation, jamais la loss seule.
- Optimiseur : AdamW, $\eta = 10^{-3}$, cosine.
- Batch size souvent petit (2–8) car les cartes sont volumineuses $\Rightarrow$ méfiance avec la BatchNorm, envisagez `GroupNorm`.
- **Regardez les prédictions**, pas seulement les chiffres.

À retenir

Le U-Net est aujourd'hui l'architecture par défaut dès qu'une sortie a la **même géométrie** que l'entrée : segmentation, débruitage, super-résolution, prédiction de profondeur — et le cœur des modèles de diffusion.

Auto-encodeur



⚠ Annoter coûte cher

Classer une image : 2 s. Segmenter **pixel par pixel** : plusieurs minutes — et par un radiologue, pour une IRM. Les images **non annotées**, elles, sont gratuites et illimitées.

Auto-encodeur

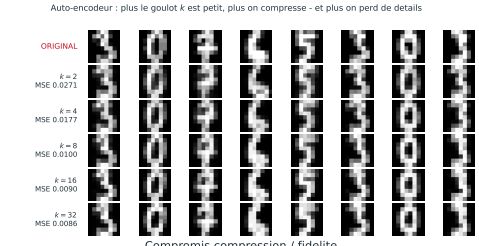
$\mathbf{z} = f_\phi(\mathbf{x}) \in \mathbb{R}^k, \quad \hat{\mathbf{x}} = g_\theta(\mathbf{z}) \in \mathbb{R}^d, \quad k \ll$

La cible, c'est l'entrée. **Aucune annotation.**

Pourquoi ça apprend

Copier l'entrée serait trivial — mais il faut passer par k **nombres seulement**. Le réseau est **obligé** de garder ce qui compte et de jeter le reste. $\mathbf{z}$ devient un **résumé**.

Dimension du goulot



latent_dim (k)

Rôle : la taille du code : combien de nombres résument une image.

↓ Trop petit : reconstruction floue, mais représentation très compacte. **↑ Trop grand :** il ne compresse plus rien ; à la limite $k = d$, il apprend l'identité.

✓ Valeur usuelle : 2 pour *visualiser* ; 16-256 pour *utiliser*

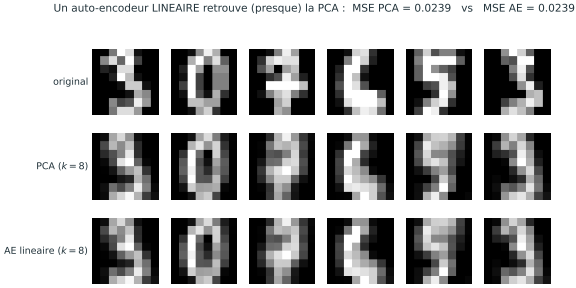
Intuition

De vraies reconstructions : cinq auto-encodeurs entraînés sur des chiffres 8×8, de $k = 2$ à $k = 32$. Avec $k = 2$ il ne reste que la forme générale ; avec $k = 32$ on ne voit presque plus la différence.

Deux usages immédiats

Débruitage : on donne $\tilde{\mathbf{x}}$ bruitée en entrée, on demande $\mathbf{x}$ propre en sortie. **Détection d'anomalies** : on entraîne sur du normal seulement ; ce que le modèle reconstruit mal est suspect.

AE linéaire = PCA



Le résultat

Si l'encodeur et le décodeur sont **linéaires** et que la loss est la MSE, le sous-espace optimal trouvé par l'auto-encodeur est exactement celui de la **PCA** (séance 2). Ci-dessus, les deux MSE sont identiques à 10^{-4} près.

Donc à quoi sert un AE ?

À faire la **même chose, en non linéaire**. Les couches cachées et les ReLU lui permettent de suivre des **variétés courbes** que la PCA, limitée aux sous-espaces plats, ne peut pas capturer.

L'auto-encodeur est une *PCA non linéaire*. C'est la meilleure façon de le retenir.

AE convolutif

```python
def down(i, o):    # stride 2 : divise H et W par 2
    return nn.Sequential(nn.Conv2d(i, o, 3, 2, 1), nn.ReLU())
def up(i, o):      # upsample puis conv : multiplie H et W par 2
    return
        nn.Sequential(nn.Upsample(scale_factor=2),
                      nn.Conv2d(i, o, 3,
                                padding=1),
                                nn.ReLU())

class ConvAE(nn.Module):
    def __init__(self, k=32):
        super().__init__()
        self.enc = nn.Sequential(
              # 1 x 64 x 64
            down(1, 16), down(16, 32), down(32, 64),   # -> 64 x 8 x 8
            nn.Flatten(), nn.Linear(64*8*8, k))
                          # -> k
        self.dec = nn.Sequential(
            nn.Linear(k, 64*8*8),
                nn.Unflatten(1, (64, 8, 8)),
            up(64, 32), up(32, 16),
                                    # -> 16 x 32 x 32
            nn.Upsample(scale_factor=2),
                                # -> 16 x 64 x 64
            nn.Conv2d(16, 1, 3, padding=1),
                nn.Sigmoid())

    def forward(self, x):
        z = self.enc(x)
        return self.dec(z), z
```

C'est un U-Net sans les skips

L'encodeur est le **CNN de la séance 3** auquel on a retiré la tête de classification ; le décodeur est son **miroir**.

Retirez les skips d'un U-Net et vous obtenez un auto-encodeur ; ajoutez-les à un auto-encodeur et vous obtenez un U-Net. **C'est la même famille**, avec deux objectifs différents : *comprimer* ou *localiser*.

Quelle loss ? pixels continus dans $[0, 1]$: **MSE** ; pixels quasi binaires : **BCE**, souvent plus nette car elle pénalise davantage les erreurs proches de 0 et 1.

Espace latent troué

⚠ On ne peut pas générer

Rien ne contraint *où* l'encodeur place les codes **z**. Il peut les éparpiller, laisser des **trous**, créer des **îlots**.

Résultat : si vous tirez un **z** au hasard et le décodez, vous tombez presque sûrement dans une zone **vide** ⇒ le décodeur produit une bouillie.

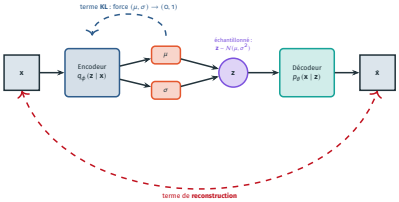
Ce qu'il nous faudrait

Un espace latent où :

- **tout point** décode en quelque chose de plausible ;
- les points **proches** décodent en images **proches** ;
- on **connaît** la distribution des **z**, pour pouvoir y échantillonner.

Un auto-encodeur *compresse*. Pour qu'il *génère*,
il faut **contraindre la forme** de l'espace latent.
C'est exactement ce que fait le VAE — et c'est là que servent la gaussienne et la KL de la séance 2.

Le VAE



La loss, telle qu'on la code

$$\mathcal{L}_{\text{VAE}} = \underbrace{\mathcal{L}_{\text{rec}}}_{(1)} + \beta \cdot \underbrace{\tfrac{1}{2} \textstyle\sum_{j=1}^{k} (\mu_j^2 + \sigma_j^2 - \log \sigma_j^2 - 1)}_{(2)}$$

(1) $\|\mathbf{x} - \hat{\mathbf{x}}\|^2$ ou $\mathrm{BCE}(\mathbf{x}, \hat{\mathbf{x}})$: « ça ressemble à l'entrée ».

(2) la **KL** de la séance 2 : « les codes restent groupés autour de 0 ».

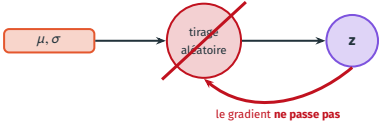
D'où vient-elle ?

On ne sait pas calculer $\log p_\theta(\mathbf{x})$: il faudrait intégrer sur *tous* les $\mathbf{z}$. On maximise donc une **borne inférieure**, l'**ELBO**, dont les deux termes sont exactement ceux-ci. Maximiser la borne suffit.

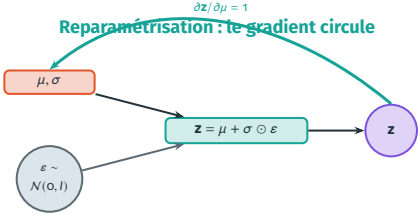
Une tension permanente

La reconstruction veut des codes **très séparés** (pour distinguer les images) ; la KL les veut **tous groupés** autour de 0. L'équilibre produit un espace latent à la fois **informatif** et **continu**.

Version naïve : on ne peut pas apprendre



Reparamétrisation : le gradient circule



le hasard est **sorti** du chemin :
c'est une *entrée*, pas une opération

Le problème

Un **tirage aléatoire** n'est pas dérivable : on ne peut pas calculer $\partial \mathbf{z} / \partial \mu$ à travers un `sample()`. Sans cela, l'encodeur ne reçoit **aucun gradient** et n'apprend jamais — reprenez la chaîne de la séance 2 : elle est coupée.

La solution

$$\mathbf{z} = \mu + \sigma \odot \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, I)$$

$\mathbf{z}$ suit toujours $\mathcal{N}(\mu, \sigma^2)$, mais c'est maintenant une **fonction dérivable** de μ et σ. Le hasard est confiné dans ε, qui ne dépend d'aucun paramètre.

VAE en PyTorch

```python
class VAE(nn.Module):
    def __init__(self, d=784, h=256, k=2):
        super().__init__()
        self.enc = nn.Sequential(nn.Linear(d, h),
                                 nn.ReLU())
        self.fc_mu     = nn.Linear(h, k)
        self.fc_logvar = nn.Linear(h, k)    # log(sigma^2)
        self.dec = nn.Sequential(nn.Linear(k, h),
                                 nn.ReLU(),
                                 nn.Linear(h, d),
                                 nn.Sigmoid())

    def forward(self, x):
        h  = self.enc(x)
        mu, logvar = self.fc_mu(h), self.fc_logvar(h)
        std = torch.exp(0.5 * logvar)           # sigma
        eps = torch.randn_like(std)             # bruit
        z   = mu + std * eps                    # LE TRICK
        return self.dec(z), mu, logvar

def vae_loss(x_hat, x, mu, logvar, beta=1.0):
    rec = F.binary_cross_entropy(x_hat, x,
          reduction="sum")
    kl  = -0.5 * torch.sum(1 + logvar - mu.pow(2) -
          logvar.exp())
    return (rec + beta * kl) / x.size(0)        # moyenne par exemple
```

⚠ Pourquoi `logvar` et pas σ ?

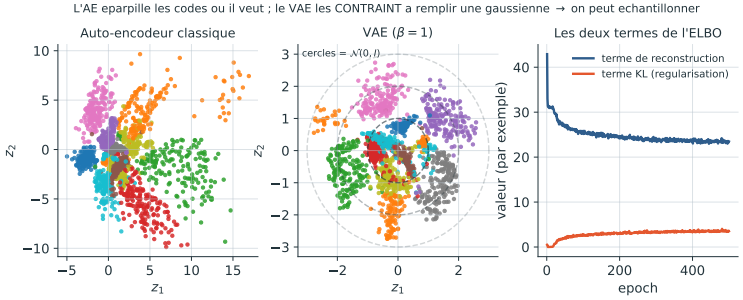
- Un réseau sort n'importe quel réel ; or σ doit être > 0. $\sigma = e^{\frac{1}{2}\log\sigma^2}$ l'est toujours.
- Numériquement bien plus stable : pas de `log(0)` dans la KL.

🛠 Générer, après entraînement

```python
model.eval()
with torch.no_grad():
    z = torch.randn(16, k)       # dans N(0, I)
    imgs = model.dec(z)          # 16 images neuves
```

L'encodeur ne sert plus : il n'était là que pour *organiser* l'espace latent pendant l'entraînement.

AE contre VAE



Intuition

À gauche, l'AE étale ses codes librement, avec des trous et des bras isolés. Au milieu, le VAE les a repliés dans une gaussienne compacte : les classes restent séparées, mais **sans espace vide entre elles**.

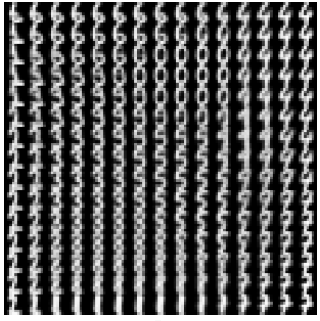
En pratique

À droite, les deux termes de la loss pendant l'entraînement. La reconstruction descend ; la KL monte d'abord (l'encodeur exploite l'espace) puis se stabilise : le modèle a trouvé son équilibre.

⚠ On ne peut faire ce genre de figure directement que pour $k = 2$. Au-delà, on projette avec une **t-SNE** ou une **UMAP**.

Générer et interpoler

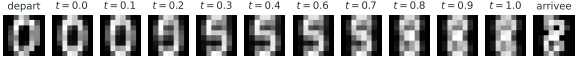
On decode une grille de z : le VAE GENERE des chiffres qui n'existent pas



Ce qu'il faut voir dans la grille

- **Aucune case vide** : tout point de l'espace latent décode en quelque chose de plausible. C'est le résultat direct du terme KL.
- Les transitions sont **continues** : les chiffres se déforment progressivement en leurs voisins.
- Aucune de ces images n'est dans le jeu d'entraînement : le modèle **génère**.

Interpolation dans l'espace latent : $z_t = (1-t)\,z_A + t\,z_B$ - les etapes sont toutes plausibles

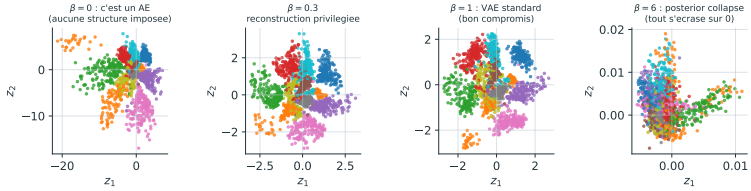


Le test de qualité d'un espace latent

On encode deux images en $\mathbf{z}_A$ et $\mathbf{z}_B$, puis on décode $\mathbf{z}_t = (1-t)\mathbf{z}_A + t\,\mathbf{z}_B$. Si toutes les étapes sont **plausibles**, l'espace est bien structuré. Comparez avec une moyenne *pixel à pixel* : elle donne toujours une image floue et double.

Le paramètre β

β dose la regularisation : reconstruction fidele ↔ espace latent bien organise



beta (β)

Rôle : le poids du terme KL : le curseur entre fidélité de reconstruction et organisation de l'espace latent.

↓ Trop petit : $\beta \to 0$: on retombe sur un auto-encodeur ordinaire, on ne peut plus générer.

↑ Trop grand : *posterior collapse* : l'encodeur renvoie $\mu = 0, \sigma = 1$ pour *toute* entrée, il ignore **x** et le décodeur produit une image moyenne.

✓ Valeur usuelle : `1 (VAE standard) ; 2-10 pour un latent plus « démêlé »`

⚠ Les images sont floues — pourquoi ?

La loss de reconstruction est une **moyenne pixel à pixel**. Face à l'incertitude (« ce bord est-il ici ou 2 pixels plus loin ? »), la solution qui minimise la MSE est la **moyenne** des possibilités — donc un bord flou. C'est une limite **structurelle**, pas un défaut de réglage.

En pratique

Remède courant au *posterior collapse* : le **KL annealing** — faire monter β de 0 à 1 progressivement sur les premières epochs, pour laisser le modèle apprendre à reconstruire avant de le contraindre.

Récapitulatif — séance 4

CNN avancé

- Le **champ réceptif** doit couvrir l'objet.
- **ResNet** : $\mathcal{F}(\mathbf{x}) + \mathbf{x}$; le +1 dans la dérivée est l'antidote au vanishing gradient.
- **Transfer learning** : geler les premières couches, η plus petit, ne jamais repartir de zéro.

U-Net

- Encodeur-décodeur **+ skip connections** par **concaténation**.
- Les skips restituent la **résolution spatiale** perdue au bottleneck (mesuré : Dice 0.99 contre 0.95).
- Loss **BCE + Dice** ; métrique **Dice / IoU**.
- Entrée divisible par $2^{\text{profondeur}}$.

Espace latent

- **AE** = U-Net sans les skips : il comprime au lieu de localiser.
- Version linéaire = **PCA**.
- Un AE seul laisse des **trous** : on ne peut pas générer.
- Encoder une **distribution** + **KL** $\Rightarrow$ latent continu, on peut échantillonner et interpoler.

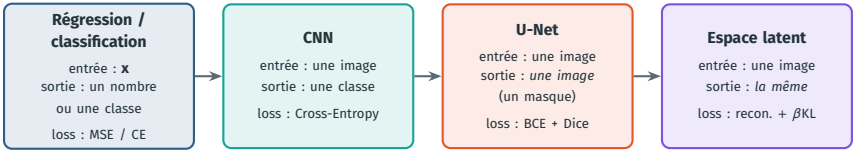
Où tout cela se retrouve : Stable Diffusion

un **auto-encodeur** compresse l'image 512×512 en un latent 64×64 $\rightarrow$
un **U-Net** y débruite pas à pas $\rightarrow$ le **décodeur** reconstruit l'image finale.

Les deux architectures de cette séance, littéralement, dans l'ordre.

Annexes

Vue d'ensemble



Dans les quatre cas : **un modèle paramétré f_θ + une fonction de coût $\mathcal{L}$ + une descente de gradient**.
Seuls changent la *forme* de f_θ et ce qu'on met en face dans $\mathcal{L}$.

Formulaire (1/2)

Modèles et pertes

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b$$

$$\hat{p} = \sigma(z) = 1/(1+e^{-z})$$

$$\hat{p}_k = e^{z_k} / \sum_j e^{z_j} \quad \text{(softmax)}$$

$$\text{MSE} = \frac{1}{n} \sum_i (\hat{y}_i - y_i)^2$$

$$\text{BCE} = -\frac{1}{n} \sum_i \left[y_i \log \hat{p}_i + (1-y_i) \log(1-\hat{p}_i) \right]$$

$$\text{CE} = -\frac{1}{n} \sum_i \sum_k y_{ik} \log \hat{p}_{ik}$$

Optimisation

$$\theta \leftarrow \theta - \eta \, \nabla_\theta J(\theta)$$

$$\text{iters/epoch} = \lceil n / \text{batch\_size} \rceil$$

$$J_{\text{reg}} = J + \lambda \|\mathbf{w}\|_2^2 \quad \text{(weight decay)}$$

Métriques de classification

$$\text{Acc} = \frac{TP+TN}{TP+TN+FP+FN} \qquad \text{Prec} = \frac{TP}{TP+FP}$$

$$\text{Rec} = \frac{TP}{TP+FN} \qquad F_1 = \frac{2 \, \text{Prec} \cdot \text{Rec}}{\text{Prec}+\text{Rec}}$$

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|} \qquad \text{Dice} = \frac{2|A \cap B|}{|A|+|B|}$$

Métriques de régression

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum (y_i - \hat{y}_i)^2}$$

$$\text{MAE} = \frac{1}{n} \sum |y_i - \hat{y}_i|$$

$$R^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}$$

Formulaire (2/2)

Convolution

$$S[i,j] = \sum_m \sum_n K[m,n]\, I[i+m, j+n]$$

$$O = \left\lfloor \frac{I + 2P - K}{S} \right\rfloor + 1$$

$$\#\theta = K \cdot K \cdot C_{in} \cdot C_{out} + C_{out}$$

Cas « same » : K impair, $P = (K-1)/2$, $S = 1 \Rightarrow O = I$.

Batch Normalization

$$\hat{z} = \frac{z - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \qquad y = \gamma \hat{z} + \beta$$

μ_B, σ_B : statistiques du batch en *train* ; moyennes mobiles en *eval*.

Auto-encodeur / VAE

$$\mathcal{L}_{\text{AE}} = \|\mathbf{x} - g_\theta(f_\phi(\mathbf{x}))\|^2$$

$$\mathbf{z} = \mu + \sigma \odot \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, I)$$

$$D_{\text{KL}} = \tfrac{1}{2} \sum_{j=1}^{k} (\mu_j^2 + \sigma_j^2 - \log \sigma_j^2 - 1)$$

$$\mathcal{L}_{\text{VAE}} = \mathcal{L}_{\text{rec}} + \beta\, D_{\text{KL}}$$

$$\text{ELBO} = \mathbb{E}_{q_\phi}[\log p_\theta(\mathbf{x}|\mathbf{z})] - D_{\text{KL}}(q_\phi \| p)$$

Segmentation

$$\mathcal{L}_{\text{Dice}} = 1 - \frac{2 \sum_i p_i y_i + \epsilon}{\sum_i p_i + \sum_i y_i + \epsilon}$$

$$\mathcal{L} = \mathcal{L}_{\text{BCE}} + \mathcal{L}_{\text{Dice}}$$

Glossaire

Français	Anglais
apprentissage supervisé	supervised learning
jeu de données	dataset
exemple, individu	sample
variable, caractéristique	feature
étiquette, cible	label, target
entraînement / validation	training / validation
fonction de coût, perte	loss, cost function
descente de gradient	gradient descent
taux d'apprentissage	learning rate
lot	batch
époque	epoch
sous-apprentissage	underfitting
sur-apprentissage	overfitting
régularisation	regularization
fuite de données	data leakage

Français	Anglais
noyau, filtre	kernel, filter
carte d'activation	feature map
pas de déplacement	stride
remplissage	padding
sous-échantillonnage	pooling, downsampling
sur-échantillonnage	upsampling
champ réceptif	receptive field
partage de poids	weight sharing
connexion résiduelle	skip / residual connection
apprentissage par transfert	transfer learning
réglage fin	fine-tuning
goulot d'étranglement	bottleneck
espace latent	latent space
segmentation sémantique	semantic segmentation
astuce de reparamétrisation	reparameterization trick

Ressources

Livres

- Géron, *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow* — le plus accessible, très pratique.
- Goodfellow, Bengio, Courville, *Deep Learning* — la référence théorique (gratuit en ligne).
- Bishop, *Pattern Recognition and Machine Learning* — pour le versant probabiliste.
- Prince, *Understanding Deep Learning* (2023) — moderne et très bien illustré (gratuit en ligne).

Cours et ressources en ligne

- *CS231n* (Stanford) — la référence sur les CNN.
- *fast.ai* — approche pratique, du code dès la première heure.
- Documentation `scikit-learn` — le guide utilisateur est un cours à lui seul.
- Tutoriels officiels `PyTorch`.
- *Distill.pub* — articles visuels (dont un excellent sur les artefacts en damier).

Articles fondateurs

LeCun *et al.* 1998 (LeNet) • Krizhevsky *et al.* 2012 (AlexNet) • Ronneberger *et al.* 2015 (U-Net) • He *et al.* 2015 (ResNet) • Kingma & Welling 2013 (VAE).

La boîte à outils

NumPy (tableaux) • **pandas** (tabulaire) • **Matplotlib** (figures) • **Pillow / OpenCV** (images) • **scikit-learn** (ML classique, splits, métriques) • **PyTorch** + **torchvision** (réseaux, GPU, modèles pré-entraînés) • **Colab / Kaggle** (GPU gratuit) • **TensorBoard / W&B** (suivi).

Check-list finale

Avant d'entraîner

- ☐ T, E, P sont-ils écrits noir sur blanc ?
- ☐ Le test est-il isolé *avant* toute autre opération ?
- ☐ Ai-je regardé mes données ? affiché un batch ?
- ☐ Y a-t-il des doublons, des fuites, des groupes ?
- ☐ Les classes sont-elles équilibrées ? Ai-je stratifié ?
- ☐ Ai-je une baseline bête à battre ?
- ☐ Les seeds sont-elles fixées ?

Pendant et après

- ☐ Le modèle peut-il surapprendre 10 exemples ?
- ☐ Est-ce que je trace train *et* validation ?
- ☐ Les hyperparamètres sont-ils réglés sur la *validation* ?
- ☐ `model.eval()` en évaluation ?
- ☐ La métrique correspond-elle au vrai problème ?
- ☐ Ai-je regardé les erreurs, pas seulement les chiffres ?
- ☐ Le test n'a-t-il été ouvert qu'une seule fois ?

Si vous ne deviez retenir qu'une chose : **le Machine Learning est une méthode expérimentale**.
Un modèle sans protocole d'évaluation honnête n'est pas un résultat, c'est une opinion.

Merci !

Questions ?

Pierre Lague

pierre.lague@inria.fr