Fondations mathématiques de l'IA moderne

Un atelier de 5 jours

Yaé Ulrich Gaba

2026

Table des matières

Préface

L'IA moderne évolue vite, mais les idées qui font tourner les systèmes d'aujourd'hui sont stables et accessibles. La même poignée de structures mathématiques — espaces vectoriels, gradients, distributions de probabilité, biais inductif, généralisation — apparaît dans chaque modèle que l'on déploie, de la régression logistique au modèle de diffusion.

Cet atelier de 5 jours construit cette intuition mathématique de bout en bout. Nous partons de rappels d'algèbre linéaire et de calcul différentiel, nous construisons vers les structures derrière les perceptrons multicouches, les autoencodeurs et les modèles de diffusion, et nous finissons par la discipline d'évaluation de ce que nous avons construit. L'objectif n'est pas de faire des participants des spécialistes du deep learning. Il est de leur donner les structures mentales pour évaluer une méthode, dialoguer avec une équipe technique, formuler un problème métier en problème d'apprentissage, et décider quand — ou quand ne pas — déployer un outil d'IA.

Public. Ingénieurs, chefs de produit, analystes, responsables d'équipes techniques qui prennent des décisions sur l'IA. Praticiens qui collaborent avec des équipes ML et veulent une compréhension réelle des fondations. Chercheurs d'autres disciplines qui veulent passer du discours sur l'IA à la pratique.

Ce que vous saurez faire après l'atelier.

- Lire la description d'une architecture de réseau de neurones et comprendre ce que chaque pièce fait mathématiquement.
- Raisonner sur pourquoi un modèle est entraîné de telle ou telle manière — et reconnaître quand un choix d'entraînement signale un problème.
- Traduire un problème réel dans le langage structuré d'un problème d'apprentissage.
- Identifier le biais inductif intégré à un modèle et juger s'il correspond aux données.
- Reconnaître les modes d'échec classiques (sur-apprentissage, décalage de distribution, perte de calibration) et les remèdes classiques.
- Décider quand l'IA est le bon outil, quand une méthode plus simple suffit, et quand davantage de données ou un autre cadrage est la vraie solution.

Prérequis. À l'aise avec les mathématiques de lycée et de premier cycle (vecteurs, fonctions, dérivées, probabilités de base). Une certaine familiarité avec Python aide pour les travaux pratiques mais n'est pas requise pour les cours.

Format. Cinq jours. Chaque journée associe un cours à un travail pratique sous forme de notebook Jupyter. Tout le code tourne sur Google Colab (offre gratuite avec GPU) ou tout environnement Python 3.10+ local avec PyTorch et NumPy.

Yaé Ulrich Gaba
Cotonou, 2026

Chapitre 1

Algèbre linéaire et modèles paramétriques

> *« Un réseau de neurones est un empilement d'applications linéaires entrecoupées de non-linéarités. La mathématique est comprise depuis un siècle ; la surprise d'ingénierie, c'est que ça marche à l'échelle. »*

Un modèle d'IA moderne est, dans son essence, une fonction. Il prend en entrée un vecteur $\boldsymbol{x} \in \mathbb{R}^d$ — les pixels d'une image, le plongement d'une phrase, les attributs d'un client — et renvoie une sortie : une étiquette de classe, une probabilité, un autre vecteur. Le vocabulaire de cette fonction est celui de l'algèbre linéaire. Les espaces vectoriels décrivent où vivent les données. Les applications linéaires les transportent d'un espace à un autre. Les ingrédients non-linéaires (ReLU, softmax) sont la ponctuation. Presque tout le reste — profondeur, largeur, dimension de plongement, espace latent, têtes d'attention — est un choix de composition d'applications linéaires.

Ce chapitre installe la colonne vertébrale algébrique de l'atelier. Nous ne démontrerons pas le théorème spectral. Nous nommerons les structures, donnerons l'intuition géométrique, et écrirons le peu de code nécessaire pour les voir à l'œuvre. Le Jour 2 introduira les gradients pour rendre ces modèles entraînables ; le Jour 3 ajoutera la probabilité pour les rendre génératifs. Mais toutes les structures sont déjà ici.

1.1 Espaces vectoriels, version courte

Un *espace vectoriel* sur $\mathbb{R}$ est un ensemble V où l'on peut additionner des éléments et les multiplier par des réels, avec les règles habituelles (associativité, distributivité, élément neutre). Chaque exemple concret qui nous intéresse dans cet atelier est une variante de $\mathbb{R}^d$: des d-uplets ordonnés de réels.

Définition 1.1 (L'exemple standard)**.** *$\mathbb{R}^d$ est l'ensemble des d-uplets $\boldsymbol{x} = (x_1, \ldots, x_d)$ avec l'addition et la multiplication scalaire faites composante par composante.*

Exemple 1.1. *Une image en niveaux de gris 28×28 pixels est un élément de $\mathbb{R}^{784}$ une fois aplatie. Le plongement d'une phrase par un petit modèle de langue est un élément de $\mathbb{R}^{768}$. La ligne d'un utilisateur dans la table d'attributs d'un système de recommandation*

est un élément de $\mathbb{R}^{50}$ environ. Ces objets ont l'air très différents dans le monde, mais ils sont identiques pour la mathématique — ce sont des points dans un espace vectoriel, et le modèle les traite ainsi.

La *dimension* d'un espace vectoriel est le nombre de directions indépendantes selon lesquelles on peut se déplacer. $\mathbb{R}^{784}$ en a 784, une par pixel. En pratique, les données ne remplissent pas l'espace : presque tous les vecteurs de $\mathbb{R}^{784}$ ressemblent à du bruit aléatoire, et presque aucun ne ressemble à un chiffre. Cette observation — les données réelles vivent sur un sous-ensemble de dimension beaucoup plus basse — est l'*hypothèse de la variété*, et c'est la raison pour laquelle les plongements fonctionnent.

Intuition

Un espace vectoriel de grande dimension est essentiellement vide. L'IA moderne exploite cela : même quand l'espace ambiant a 10^4 dimensions, la structure qui nous intéresse (l'ensemble des images plausibles, l'ensemble des phrases sensées) vit sur un sous-ensemble minuscule et courbe. Apprendre ce sous-ensemble, c'est une grande partie de ce que fait l'entraînement.

1.2 Applications linéaires et matrices

Une *application linéaire* $f : \mathbb{R}^d \to \mathbb{R}^k$ est une fonction qui respecte l'addition et la multiplication scalaire :

$$f(\boldsymbol{x} + \boldsymbol{y}) = f(\boldsymbol{x}) + f(\boldsymbol{y}), \qquad f(\alpha \boldsymbol{x}) = \alpha f(\boldsymbol{x}).$$

Toute telle application est donnée par une matrice $W \in \mathbb{R}^{k \times d}$: $f(\boldsymbol{x}) = W\boldsymbol{x}$. Choisissez une base et l'application devient une liste de nombres.

Définition 1.2 (Application affine)**.** *Une application affine est une application linéaire plus un décalage constant : $\boldsymbol{x} \mapsto W\boldsymbol{x} + \boldsymbol{b}$, où $\boldsymbol{b} \in \mathbb{R}^k$ est le biais. Chaque couche d'un réseau de neurones standard est une application affine suivie d'une non-linéarité.*

La matrice W fait trois choses simultanément : elle dilate, elle pivote, et elle projette sur un sous-espace (éventuellement de dimension plus basse). La décomposition en valeurs singulières rend cela précis — tout W s'écrit $U\Sigma V^\top$ où U et V sont des rotations et Σ est une dilatation diagonale positive. Nous n'aurons pas besoin du détail de la SVD, mais l'image vaut d'être retenue :

Intuition

Chaque couche linéaire d'un réseau de neurones pivote l'espace d'entrée, étire certaines directions et en rétrécit d'autres, et jette éventuellement des directions (celles dont la valeur singulière est nulle). Empiler des couches empile ces opérations. Les non-linéarités sont ce qui empêche la pile de s'effondrer en une seule application linéaire.

1.3 Du perceptron au MLP

Le réseau de neurones le plus simple est un seul neurone, le *perceptron* (Rosenblatt, 1958) :

$$y = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b)$$

où $\boldsymbol{w} \in \mathbb{R}^d$ est un vecteur de poids, $b \in \mathbb{R}$ un biais, et σ une fonction non-linéaire. Avec $\sigma(z) = 1/(1 + e^{-z})$ (la sigmoïde logistique), c'est exactement une régression logistique habillée d'un autre vocabulaire.

Proposition 1.1 (Pourquoi une seule couche ne suffit pas)**.** *Une seule couche affine ne peut séparer l'espace d'entrée que par un hyperplan. La fonction XOR — sortie 1 si exactement une des deux entrées binaires vaut 1 — ne peut être exprimée par aucun perceptron seul. C'est la critique originale de Minsky et Papert (1969) qui a brièvement tué le champ.*

La solution est la composition. Un *perceptron multicouche* (MLP) empile des applications affines avec des non-linéarités entre elles. Un MLP à deux couches de $\mathbb{R}^d$ vers $\mathbb{R}^k$ est :

$$\boldsymbol{h} = \sigma(W_1 \boldsymbol{x} + \boldsymbol{b}_1), \qquad \boldsymbol{y} = W_2 \boldsymbol{h} + \boldsymbol{b}_2$$

où $W_1 \in \mathbb{R}^{m \times d}$, $W_2 \in \mathbb{R}^{k \times m}$, et m est la largeur de la couche cachée. La non-linéarité σ est appliquée composante par composante. Les réseaux modernes utilisent $\sigma(z) = \max(0, z)$, l'unité linéaire rectifiée (ReLU), presque partout.

> **Note**
>
> Le théorème d'approximation universelle (Cybenko, 1989 ; Hornik, 1991) dit qu'un MLP même à une seule couche cachée, avec suffisamment de largeur, peut approximer toute fonction continue sur un compact avec une précision arbitraire. C'est une affirmation d'*existence*, pas d'*apprenabilité* ni d'efficacité. Les réseaux profonds modernes ne sont pas profonds parce que les réseaux peu profonds ne peuvent pas représenter la fonction. Ils sont profonds parce que la profondeur rend les bonnes fonctions plus faciles à trouver par descente de gradient.

1.3.1 Compter les paramètres

Prenez un MLP avec dimension d'entrée $d = 784$, une couche cachée de largeur $m = 256$, et dimension de sortie $k = 10$. Le compte des paramètres :

$$\underbrace{784 \cdot 256 + 256}_{\text{première couche : } W_1, \boldsymbol{b}_1} + \underbrace{256 \cdot 10 + 10}_{\text{deuxième couche : } W_2, \boldsymbol{b}_2} = 203\,530.$$

Un petit classifieur MNIST a déjà quelques centaines de milliers de paramètres. GPT-3 en a 175 milliards. Les nombres grandissent vite, mais la structure ne change pas — chaque couche reste une application affine suivie d'une non-linéarité.

1.3.2 Le MLP en code

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

class MLPDeuxCouches(nn.Module):
    def __init__(self, d_in: int, d_cache: int, d_out: int):
        super().__init__()
        self.fc1 = nn.Linear(d_in, d_cache)   # application affine : W1 x + b1
        self.fc2 = nn.Linear(d_cache, d_out)  # application affine : W2 h + b2

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        h = torch.relu(self.fc1(x))           # non-linéarité
        y = self.fc2(h)
        return y

modele = MLPDeuxCouches(d_in=784, d_cache=256, d_out=10)
n_params = sum(p.numel() for p in modele.parameters())
print(f"Nombre de paramètres : {n_params:,}")
```

La couche `nn.Linear` contient exactement le W et le $\boldsymbol{b}$ des formules ci-dessus. Tout le reste de cet atelier sera des variations sur ce thème : plus de couches, des non-linéarités différentes, du partage de poids (CNN), des connexions structurées (GNN), de l'attention (Transformers). L'unité de base est l'application affine.

Exercice

Un bloc résiduel dans un ResNet a la forme $\boldsymbol{y} = \boldsymbol{x} + F(\boldsymbol{x})$, où F est un petit sous-réseau. Argumentez de façon informelle pourquoi cela devrait rendre l'entraînement plus facile que la composition brute $\boldsymbol{y} = F(\boldsymbol{x})$. (Indice : que se passe-t-il si F retourne zéro ? Qu'est-ce que cela implique pour le gradient ?)

1.4 Plongements : vecteurs comme représentations

Jusqu'ici nous avons traité l'entrée $\boldsymbol{x}$ comme si elle était déjà un vecteur. Mais l'entrée réelle est souvent autre chose : un mot, un identifiant d'utilisateur, une catégorie discrète. La transformer en vecteur est la première décision que prend le modèle.

Définition 1.3 (Plongement)**.** *Un plongement est une fonction d'un ensemble discret V (vocabulaire, ensemble d'utilisateurs, ensemble de catégories) vers un espace vectoriel $\mathbb{R}^d$. Concrètement, c'est une matrice $E \in \mathbb{R}^{|V| \times d}$: chaque ligne est la représentation vectorielle d'un élément.*

La matrice de plongement est un paramètre du modèle. Elle est apprise. Après entraînement, des éléments similaires se retrouvent proches dans l'espace de plongement, et la géométrie de cet espace porte le sens que le modèle a découvert.

Exemple 1.2. *Word2Vec (Mikolov et al., 2013) entraîne une matrice de plongement pour les mots anglais de sorte que la similarité cosinus de deux vecteurs de mots suit leur similarité sémantique. La régularité géométrique célèbre* $\boldsymbol{e}_{king} - \boldsymbol{e}_{man} + \boldsymbol{e}_{woman} \approx \boldsymbol{e}_{queen}$ *est une observation empirique sur l'espace résultant, pas un objectif d'entraînement. L'objectif d'entraînement était bien plus simple : prédire un mot à partir de son contexte.*

Dans les modèles de langue modernes, le plongement d'entrée est la première couche (une consultation dans une matrice de plongement), et le reste du réseau opère sur ces vecteurs. L'astuce est que le même espace vectoriel fait double emploi : il est l'espace d'entrée *et* la mémoire de travail du modèle.

Attention

« Plongement » est surchargé en apprentissage automatique. Cela peut désigner une représentation d'entrée apprise (Word2Vec, le plongement de jeton d'un LLM), une activation cachée à l'intérieur d'un réseau (« le plongement de cette image est la sortie de l'avant-dernière couche »), ou une représentation entraînée séparément utilisée comme attribut dans un modèle aval (sentence-transformers). Les trois sont des vecteurs. Les différences tiennent à comment ils sont produits et à quoi ils servent.

1.5 Autoencodeurs et géométrie de l'espace latent

Un *autoencodeur* apprend une représentation compressée de son entrée en essayant de reproduire cette entrée à partir de la compression. Il a deux parties :

$$\underbrace{\boldsymbol{z} = f_{\text{enc}}(\boldsymbol{x})}_{\text{encodeur, } \mathbb{R}^d \to \mathbb{R}^k} \qquad \underbrace{\hat{\boldsymbol{x}} = f_{\text{dec}}(\boldsymbol{z})}_{\text{décodeur, } \mathbb{R}^k \to \mathbb{R}^d}$$

avec $k < d$. L'objectif d'entraînement est la reconstruction : minimiser $\|\boldsymbol{x} - \hat{\boldsymbol{x}}\|^2$ sur un jeu de données. Comme le goulot $\boldsymbol{z}$ est de dimension plus basse que $\boldsymbol{x}$, l'autoencodeur ne peut pas mémoriser l'entrée ; il doit trouver de la structure.

Intuition

L'ACP (PCA) est un autoencodeur linéaire. Si f_{enc} et f_{dec} sont contraints à être des applications linéaires et qu'on minimise l'erreur quadratique de reconstruction, la solution optimale projette sur les k premières composantes principales. Les autoencodeurs non-linéaires généralisent cette idée : ils apprennent une surface courbe de dimension k à l'intérieur de $\mathbb{R}^d$ qui épouse les données.

L'espace du goulot $\mathbb{R}^k$ s'appelle l'*espace latent*. C'est là que vit la compréhension compressée des données par le modèle. Deux questions valent généralement la peine d'être posées sur tout espace latent :

- **Que signifie chaque dimension ?** Souvent rien en soi — les axes sont arbitraires — mais des directions dans l'espace correspondent souvent à des facteurs sémantiques (éclairage, pose, identité pour un jeu de visages ; sujet, sentiment, temps pour un jeu de textes).

- **Comment transporte-t-il entre points voisins ?** Si de petits déplacements dans $\boldsymbol{z}$ produisent de petits changements plausibles dans $\hat{\boldsymbol{x}}$, le modèle a appris la bonne variété. Si de petits déplacements produisent des sauts ou du non-sens, il ne l'a pas apprise.

Le Jour 3 reviendra aux autoencodeurs dans leur forme probabiliste (autoencodeurs variationnels, VAE) comme l'une des passerelles vers la modélisation générative. La structure autoencodeur — encodeur, latent, décodeur — est aussi exactement la structure d'un modèle de diffusion déguisé.

1.6 Ce que vous avez vu

Tout modèle paramétrique de cet atelier est construit à partir de trois pièces : des vecteurs (les données et les représentations intermédiaires), des applications affines (les couches, avec leurs poids et biais), et des non-linéarités (les activations qui empêchent l'effondrement). Les architectures modernes — CNN, RNN, Transformers, GNN, modèles de diffusion — sont des variations sur quelles applications affines sont liées ensemble et quelles non-linéarités sont insérées où. Les noms changent. La mathématique est la même.

Ce que nous n'avons pas encore fait, c'est rendre quoi que ce soit *apprenable*. Pour l'instant, les poids W et les biais $\boldsymbol{b}$ sont juste des nombres ; le modèle est une fonction. Le Jour 2 introduit le gradient, la fonction de perte et la machinerie d'optimisation qui transforme cette fonction en quelque chose qui peut être ajusté aux données. La structure reste la même. L'entraînement, c'est ce qui la remplit.

Travail pratique

Ouvrez le notebook du Jour 1 (`Day1_LinearAlgebra_ParametricModels.ipynb`). Vous allez :

1. Implémenter un MLP à deux couches à la main en NumPy — en écrivant explicitement les produits matriciels et ReLU — et vérifier que le compte de paramètres correspond à ce que nous avons calculé.

2. Entraîner le même MLP en PyTorch sur MNIST, obtenant environ 97 % de précision sur l'ensemble de test en moins d'une minute.

3. Extraire les activations de la couche cachée pour l'ensemble de test, les projeter en 2D avec l'ACP, et colorer les points par leur vraie classe de chiffre. Vous verrez des amas — le réseau a appris une représentation où les chiffres similaires vivent près les uns des autres.

4. Entraîner un petit autoencodeur sur MNIST avec un espace latent de dimension 2. Visualiser directement l'espace latent et marcher le long de lignes droites dans cet espace : vous verrez des chiffres se métamorphoser progressivement en d'autres chiffres.

À la fin du TP, vous aurez construit la machinerie que le reste de l'atelier suppose : un MLP qui fonctionne, un autoencodeur qui fonctionne, et l'habitude d'inspecter ce qu'un réseau a appris en regardant ses représentations internes.

Chapitre 2

Calcul différentiel et optimisation

> *« Entraîner un réseau de neurones, c'est appliquer la règle de la chaîne, soigneusement, plusieurs fois par seconde. »*

À la fin du Jour 1, nous avions un modèle : un MLP, un empilement d'applications affines et de ReLU, dont les poids W et les biais $\boldsymbol{b}$ étaient de simples nombres aléatoires. Le modèle était une fonction. Il ne faisait rien d'utile. Pour le rendre utile, il faut ajuster ces nombres pour que les sorties du modèle correspondent aux données — et il faut le faire automatiquement, parce qu'il y a trop de nombres pour les régler à la main.

Ce chapitre traite de cet ajustement. La mathématique est le calcul différentiel : la dérivée dit comment la sortie d'une fonction change quand on perturbe ses entrées, et le gradient généralise cette idée aux fonctions de plusieurs variables. L'algorithme est la descente de gradient : faire un petit pas dans la direction qui diminue la perte. L'astuce d'ingénierie qui rend cela tractable pour des réseaux à un million de paramètres est la règle de la chaîne, appliquée récursivement — la rétropropagation. À la fin de la journée, vous aurez implémenté la rétropropagation à la main, en NumPy, pour le MLP construit hier.

2.1 La fonction de perte

Avant de pouvoir optimiser quoi que ce soit, il faut dire ce qu'on optimise. Il nous faut un seul nombre réel qui mesure *à quel point* le modèle se trompe sur un jeu de données. Ce nombre est la *perte*.

Définition 2.1 (Fonction de perte)**.** *Étant donné un modèle f_θ de paramètres θ et un jeu de données $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$, une fonction de perte $\mathcal{L}(\theta)$ est une fonction à valeurs réelles de θ, petite quand le modèle ajuste bien les données et grande quand ce n'est pas le cas. L'entraînement est le problème d'optimisation* $\min_\theta \mathcal{L}(\theta)$.

Deux pertes couvrent la majeure partie de cet atelier :

Erreur quadratique moyenne (régression). Pour des cibles à valeurs réelles :

$$\mathcal{L}_{\text{MSE}}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \| f_\theta(\boldsymbol{x}_i) - y_i \|^2.$$

C'est la distance euclidienne au carré entre prédiction et cible, moyennée sur le jeu de données.

Entropie croisée (classification). Pour des étiquettes de classe $y_i \in \{1, \dots, K\}$, le modèle produit une distribution de probabilités $f_\theta(\boldsymbol{x}_i) \in \Delta^{K-1}$ sur les classes (via softmax), et :

$$\mathcal{L}_{\text{CE}}(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \log f_\theta(\boldsymbol{x}_i)_{y_i}.$$

C'est la log-vraisemblance négative sous la distribution prédite par le modèle. Le Jour 3 expliquera pourquoi *cette* perte et pas une autre.

Intuition

La perte est un paysage. Les paramètres θ sont des coordonnées. L'entraînement est l'acte de trouver un point bas. Le paysage a des millions de dimensions (une par paramètre), mais c'est par ailleurs juste une fonction $\mathcal{L} : \mathbb{R}^P \to \mathbb{R}$. Tout ce chapitre traite de comment marcher en bas de ce paysage efficacement.

2.2 Dérivées et gradients

Pour une fonction scalaire $f : \mathbb{R} \to \mathbb{R}$, la dérivée $f'(x)$ est le taux de variation en x : si l'on perturbe x de ϵ, la sortie change d'environ $f'(x) \cdot \epsilon$. Le gradient généralise cela aux fonctions de plusieurs variables.

Définition 2.2 (Gradient)**.** *Pour une fonction $f : \mathbb{R}^d \to \mathbb{R}$, le gradient en $\boldsymbol{x}$ est le vecteur des dérivées partielles :*

$$\nabla f(\boldsymbol{x}) = \left(\frac{\partial f}{\partial x_1}(\boldsymbol{x}),\ \frac{\partial f}{\partial x_2}(\boldsymbol{x}),\ \dots,\ \frac{\partial f}{\partial x_d}(\boldsymbol{x}) \right).$$

Le gradient pointe dans la direction de plus grande augmentation*. La direction opposée, $-\nabla f(\boldsymbol{x})$, est celle de plus grande diminution.*

C'est le fait géométrique clé derrière tout algorithme d'optimisation de cet atelier : pour diminuer une fonction, suivre son gradient négatif.

Exemple 2.1. *Soit $f(x_1, x_2) = x_1^2 + 3x_2^2$. Le gradient est $\nabla f = (2x_1, 6x_2)$. Au point $(1, 1)$, le gradient est $(2, 6)$. La fonction augmente le plus rapidement dans la direction $(2, 6)/\|(2, 6)\|$ et diminue le plus rapidement dans la direction opposée. Les deux axes pointent vers l'extérieur du minimum en $(0, 0)$, mais la direction x_2 est plus pentue, parce que la fonction s'incurve plus fortement selon x_2.*

2.3 Descente de gradient

L'algorithme d'optimisation le plus simple est celui que suggère la section précédente : commencer quelque part, calculer le gradient, faire un pas dans la direction du gradient négatif, recommencer.

Définition 2.3 (Descente de gradient)**.** *Étant donné un taux d'apprentissage $\eta > 0$ et un point initial θ_0, la descente de gradient produit une suite*

$$\theta_{t+1} = \theta_t - \eta \nabla \mathcal{L}(\theta_t).$$

Le taux d'apprentissage η contrôle la taille du pas. Si η est trop petit, l'entraînement est lent. Si η est trop grand, les pas dépassent et la perte oscille ou diverge.

Attention

Le taux d'apprentissage est l'hyperparamètre le plus déterminant en deep learning. « Mon modèle ne s'entraîne pas », en pratique, c'est presque toujours « mon taux d'apprentissage est mauvais ». Le TP d'aujourd'hui inclut un balayage du taux d'apprentissage pour que vous le voyiez directement.

2.4 La règle de la chaîne et la rétropropagation

Pour un MLP à deux couches, la perte est une composition de fonctions :

$$\mathcal{L}(\theta) = \ell\big(f_2(f_1(\boldsymbol{x}))\big)$$

où f_1 est la première couche affine-plus-ReLU, f_2 est la seconde, et ℓ est la perte appliquée à la sortie. Pour faire une descente de gradient sur $\mathcal{L}$, nous avons besoin du gradient par rapport à chaque paramètre dans f_1 et f_2.

La règle de la chaîne du calcul à une variable se généralise aux fonctions à valeurs vectorielles. Si $\boldsymbol{y} = g(\boldsymbol{u})$ et $\boldsymbol{u} = h(\boldsymbol{x})$, alors

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{u}} \cdot \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}}$$

où les dérivées partielles sont des matrices jacobiennes et le produit est une multiplication matricielle. Appliquée de manière répétée à la composition ci-dessus, cela nous permet de calculer $\partial \mathcal{L}/\partial \theta$ pour chaque paramètre.

Intuition

La passe avant calcule la perte des entrées vers la sortie scalaire. La passe arrière calcule les dérivées dans la direction opposée : elle part de la perte scalaire (dont la dérivée par rapport à elle-même est 1) et propage les dérivées partielles en arrière à travers chaque couche, multipliant des jacobiennes au passage. Le travail total de la passe arrière est à peu près le même que celui de la passe avant — un facteur 2 à 3, selon le réseau. C'est pourquoi l'entraînement est faisable.

2.4.1 Rétropropagation pour un MLP à deux couches, à la main

Pour le MLP à deux couches du Jour 1 :

$$\begin{aligned} \boldsymbol{h}_{\text{pre}} &= W_1 \boldsymbol{x} + \boldsymbol{b}_1 \\ \boldsymbol{h} &= \text{ReLU}(\boldsymbol{h}_{\text{pre}}) \\ \boldsymbol{y} &= W_2 \boldsymbol{h} + \boldsymbol{b}_2 \end{aligned}$$

avec une perte d'entropie croisée après softmax, les gradients ont une forme close. Soient $\boldsymbol{p} = \mathrm{softmax}(\boldsymbol{y})$ et $\boldsymbol{y}^*$ la cible en encodage one-hot. Alors :

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} &= \boldsymbol{p} - \boldsymbol{y}^* && \text{(la composition softmax-CE se simplifie)} \\
\frac{\partial \mathcal{L}}{\partial W_2} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} \cdot \boldsymbol{h}^\top, \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}_2} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}} &= W_2^\top \cdot \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_{\text{pre}}} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}} \odot \mathbf{1}[\boldsymbol{h}_{\text{pre}} > 0] && \text{(gradient de ReLU)} \\
\frac{\partial \mathcal{L}}{\partial W_1} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_{\text{pre}}} \cdot \boldsymbol{x}^\top, \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}_1} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_{\text{pre}}}.
\end{aligned}
$$

Cinq lignes. Voilà la rétropropagation pour un MLP à deux couches. Le TP implémente ces formules en NumPy et les vérifie contre l'autograd de PyTorch au bit près.

2.4.2 Pourquoi on n'écrit jamais ce code en pratique

Pour un MLP à deux couches, on peut dériver ces formules. Pour un ResNet à cinquante couches, ou un Transformer avec attention croisée, l'algèbre devient désespérée. Les frameworks modernes (PyTorch, JAX, TensorFlow) enregistrent le graphe de calcul de la passe avant et le rejouent en arrière, appliquant la règle de la chaîne à chaque nœud. Chaque opération primitive (matmul, ReLU, softmax) a sa rétropropagation codée à la main ; le framework les compose automatiquement. C'est ce qu'on appelle la *programmation différentiable*.

Note

Vous n'écrirez jamais la passe arrière pour un modèle de production. Mais vous devez comprendre ce qu'elle est, car presque chaque échec d'entraînement — gradients qui s'évanouissent, gradients qui explosent, ReLU mortes, compromis de *gradient checkpointing* — est une histoire de rétropropagation. « Ma perte vaut NaN » est une histoire de rétropropagation.

2.5 Descente de gradient stochastique

La perte est une somme sur le jeu de données :

$$
\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell_i(\theta).
$$

Calculer $\nabla \mathcal{L}$ exactement nécessite une passe complète sur le jeu de données. Pour MNIST, c'est 60 000 exemples par étape. Pour ImageNet, 1,2 million. Pour un LLM pré-entraîné sur 10^{12} jetons, le nombre n'a plus de sens.

Définition 2.4 (Descente de gradient stochastique, SGD)**.** *À chaque étape, échantillonner un petit lot $B \subset \{1, \ldots, N\}$ et mettre à jour avec le gradient du lot :*

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{1}{|B|} \sum_{i \in B} \nabla \ell_i(\theta_t).$$

Le gradient du lot est une estimation bruitée mais non biaisée du gradient complet.

Deux choses font que la SGD fonctionne en pratique :

1. **Calcul.** Un lot de 128 tient en mémoire, s'exécute en millisecondes, et donne un pas utilisable. La descente de gradient complète serait cent fois plus lente par étape pour le même budget de calcul.
2. **Statistique.** Le bruit dans l'estimation du gradient agit comme une régularisation implicite, aidant l'optimiseur à s'échapper des minima étroits. C'est en partie pourquoi les réseaux de neurones généralisent (Hardt et al., 2016 ; Keskar et al., 2017).

2.6 Adam : moment et taux d'apprentissage adaptatifs

La SGD simple est l'optimiseur le plus simple, mais rarement le meilleur en pratique. Adam (Kingma & Ba, 2015) ajoute deux ingrédients :

Moment. Au lieu de faire un pas dans la direction du gradient courant, faire un pas dans la direction d'une moyenne exponentiellement pondérée des gradients récents. Cela lisse le bruit et accélère la progression selon les directions cohérentes.

Taux d'apprentissage par paramètre. Chaque paramètre reçoit son propre taux d'apprentissage effectif, basé sur une estimation glissante de la magnitude de *ses* gradients récents. Les paramètres aux gradients constamment petits font des pas plus grands ; les paramètres aux gradients constamment grands font des pas plus petits.

La règle de mise à jour, avec $\beta_1 = 0,9$, $\beta_2 = 0,999$, $\epsilon = 10^{-8}$:

$$\begin{aligned}
\boldsymbol{m}_t &= \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1) \nabla \mathcal{L}(\theta_t) && \text{(moment)} \\
\boldsymbol{v}_t &= \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2) (\nabla \mathcal{L}(\theta_t))^2 && \text{(moyenne glissante du gradient au carré)} \\
\theta_{t+1} &= \theta_t - \eta \cdot \frac{\hat{\boldsymbol{m}}_t}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon}
\end{aligned}$$

où $\hat{\boldsymbol{m}}, \hat{\boldsymbol{v}}$ sont des versions corrigées du biais de $\boldsymbol{m}, \boldsymbol{v}$. La racine carrée et la division sont élément par élément.

Adam est l'optimiseur par défaut pour presque tous les réseaux modernes. La SGD avec moment bat encore Adam sur certains benchmarks de vision (Wilson et al., 2017), mais c'est la tolérance d'Adam aux mauvais choix d'hyperparamètres qui en fait le défaut pratique.

2.7 Lire une courbe d'apprentissage

Une courbe d'apprentissage trace la perte (ou la précision) en fonction du pas d'entraînement. C'est l'instrument de diagnostic du deep learning. Quatre formes caractéristiques :

- **Saine :** la perte d'entraînement diminue régulièrement, la perte de validation diminue aussi et la suit à faible écart. Le modèle apprend, et ce qu'il apprend se généralise.
- **Sur-apprentissage :** la perte d'entraînement diminue, la perte de validation diminue un moment puis remonte. Le modèle mémorise les exemples d'entraînement au lieu d'apprendre le motif sous-jacent. Le Jour 5 y revient.
- **Sous-apprentissage :** les deux pertes plafonnent en haut. Le modèle manque de capacité, l'optimiseur est bloqué, ou le taux d'apprentissage est mauvais.
- **Divergence :** la perte explose vers le haut, souvent vers l'infini. Le taux d'apprentissage est trop grand, ou la perte est calculée sur une région non différentiable (log de zéro, division par zéro dans l'attention).

Intuition

Lire la courbe d'apprentissage est la première chose à faire quand l'entraînement ne fonctionne pas. La plupart des pathologies ont une forme caractéristique, et la courbe vous dit laquelle vous regardez en quelques secondes. Le TP d'aujourd'hui produit chacune de ces formes délibérément, pour que vous les reconnaissiez la prochaine fois.

2.8 Ce que vous avez vu

Entraîner un réseau de neurones, c'est la descente de gradient sur un paysage de perte de grande dimension, rendue tractable par la règle de la chaîne (rétropropagation) et rendue pratique par les mini-lots stochastiques et les optimiseurs adaptatifs comme Adam. La mathématique est le calcul différentiel ; l'ingénierie est la différentiation automatique. Chaque framework moderne vous donne la seconde gratuitement, mais seule la première vous laisse raisonner sur ce qui va mal quand quelque chose va mal.

Le Jour 3 ajoute l'ingrédient manquant : la probabilité. Nous avons parlé de « prédire » des sorties et de « faire correspondre » des cibles, mais les modèles génératifs font quelque chose de plus étrange — ils apprennent une distribution de probabilités entière et y échantillonnent. Cela exige de prendre la fonction de perte au sérieux comme une vraisemblance.

Travail pratique

Ouvrez le notebook du Jour 2 (`Day2_Calculus_Optimization.ipynb`). Vous allez :

1. Calculer le gradient d'une petite fonction à la main, puis vérifier contre l'autograd de PyTorch.
2. Ajouter une passe arrière manuelle au MLP NumPy du Jour 1, puis vérifier chaque gradient contre l'autograd de PyTorch à la dernière décimale — la *gradient check* standard utilisée pour déboguer une nouvelle couche.
3. Entraîner le MLP manuel sur MNIST avec une SGD simple et votre propre boucle d'entraînement. Aucun framework.
4. Remplacer la boucle d'entraînement manuelle par PyTorch + Adam et observer la différence.
5. Balayer le taux d'apprentissage sur $\{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}\}$ et tracer les courbes d'apprentissage résultantes sur un seul axe. Vous verrez les quatre formes caractéristiques ci-dessus.

À la fin du TP, vous aurez écrit et vérifié une passe arrière, entraîné un réseau à partir de premiers principes, et pris l'habitude de regarder la courbe d'apprentissage *avant* d'accuser le modèle.

Chapitre 3

Probabilités et modélisation générative

> *« Un modèle génératif est une distribution de probabilités dont on peut tirer des échantillons. Tout le reste — la fonction de perte, l'architecture, l'algorithme d'entraînement — est au service de l'apprentissage de cette distribution à partir de données. »*

Les Jours 1 et 2 ont construit une fonction : un MLP qui projette des entrées vers des sorties, dont les poids sont entraînés par descente de gradient sur une perte. Le cadre était déterministe. Une entrée donnée produisait une sortie donnée. Cette image suffit pour la classification et la régression, mais elle ne capture pas ce que font les modèles génératifs modernes. Un modèle de diffusion ne prédit pas une seule sortie à partir d'une entrée ; il échantillonne une image (ou plusieurs images différentes) depuis une distribution apprise. Un modèle de langue ne prédit pas un seul jeton suivant ; il donne une distribution sur le vocabulaire, et l'échantillonneur y tire.

Pour parler de cela, il nous faut la probabilité. Ce chapitre est le pont : il réintroduit la fonction de perte du Jour 2 comme une vraisemblance, développe l'autoencodeur variationnel comme le modèle génératif non trivial le plus simple, et se termine par l'image du débruitage par diffusion qui sous-tend essentiellement tout générateur d'images et d'audio à l'état de l'art construit ces cinq dernières années.

3.1 Distributions, densités, et ce qu'on cherche à apprendre

Une distribution de probabilités attribue un poids à chaque résultat possible. Pour un ensemble fini, une distribution est une liste de nombres positifs qui somment à 1. Pour un espace continu comme $\mathbb{R}^d$, c'est une fonction de densité $p(\boldsymbol{x}) \geq 0$ dont l'intégrale sur tout l'espace vaut 1.

Définition 3.1 (Densité)**.** *Une densité de probabilité $p : \mathbb{R}^d \to \mathbb{R}_{\geq 0}$ satisfait $\int p(\boldsymbol{x})\, d\boldsymbol{x} = 1$. Pour toute région $A \subseteq \mathbb{R}^d$, la probabilité qu'un échantillon tombe dans A est $\int_A p(\boldsymbol{x})\, d\boldsymbol{x}$.*

Les données qui nous intéressent — images, textes, molécules — viennent d'une distribution inconnue p_{data}. La modélisation générative signifie : estimer p_{data} à partir d'échantillons, assez bien pour générer de nouveaux échantillons qui ressemblent à ceux qui en viennent.

Intuition

Pour MNIST, p_{data} est une densité sur $\mathbb{R}^{784}$. Presque chaque point de $\mathbb{R}^{784}$ est une image de bruit avec une densité essentiellement nulle. Les vrais chiffres vivent sur un sous-ensemble minuscule et courbe (la variété du Jour 1), où la densité est énorme. Un modèle génératif est quelque chose qui a intériorisé la forme de cette région de haute densité.

3.2 Vraisemblance : une perte avec une signification

Supposons que nous ayons une famille paramétrique de densités $\{p_\theta : \theta \in \Theta\}$. Étant donnés des échantillons $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ tirés de p_{data}, la *vraisemblance* est la densité de probabilité que le modèle attribue aux données :

$$\mathcal{L}_{\text{vrais}}(\theta) = \prod_{i=1}^{N} p_\theta(\boldsymbol{x}_i).$$

L'*estimation par maximum de vraisemblance* (MLE) choisit le θ qui rend les données les plus plausibles sous p_θ. En pratique, on travaille avec la log-vraisemblance négative, parce qu'elle transforme le produit en somme et donne quelque chose à minimiser :

$$-\log \mathcal{L}_{\text{vrais}}(\theta) = -\sum_{i=1}^{N} \log p_\theta(\boldsymbol{x}_i).$$

Proposition 3.1 (L'entropie croisée est la log-vraisemblance négative)**.** *Pour un classifieur de sortie softmax $p_\theta(y \mid \boldsymbol{x})$, la perte d'entropie croisée du Jour 2,*

$$\mathcal{L}_{CE}(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \log p_\theta(y_i \mid \boldsymbol{x}_i),$$

est exactement $-\frac{1}{N} \log \mathcal{L}_{vrais}(\theta)$. Entraîner un classifieur avec l'entropie croisée, c'est faire du maximum de vraisemblance sur la distribution conditionnelle $p_\theta(y \mid \boldsymbol{x})$.

Le classifieur du Jour 2 est déjà un modèle génératif — d'étiquettes conditionnellement aux entrées. Pour générer des *entrées*, il nous faut un modèle de la distribution des entrées elle-même. C'est l'objet du reste de ce chapitre.

3.3 Divergence de KL : distance entre distributions

Pour comparer deux distributions, on utilise la divergence de Kullback–Leibler :

$$D_{\text{KL}}(q \,\|\, p) = \int q(\boldsymbol{x}) \log \frac{q(\boldsymbol{x})}{p(\boldsymbol{x})} \, d\boldsymbol{x} = \mathbb{E}_{\boldsymbol{x} \sim q}[\log q(\boldsymbol{x}) - \log p(\boldsymbol{x})] \,.$$

La KL est positive ou nulle, nulle si et seulement si $p = q$, et asymétrique. Ce n'est pas une métrique — $D_{\mathrm{KL}}(p \,\|\, q) \neq D_{\mathrm{KL}}(q \,\|\, p)$ en général — mais c'est l'objet naturel quand l'une des distributions est les données et l'autre est le modèle.

Proposition 3.2 (La MLE minimise la KL avec les données)**.** *À une constante près qui ne dépend pas de θ, la log-vraisemblance négative est égale à $D_{\mathrm{KL}}(p_{data} \,\|\, p_\theta)$. Le maximum de vraisemblance est une minimisation approchée de la KL entre la distribution des données et celle du modèle.*

C'est l'énoncé le plus propre de ce que fait l'entraînement d'un modèle génératif : rendre la distribution du modèle proche de la distribution des données au sens de la KL. Le reste du chapitre est deux histoires sur *comment* faire cette minimisation quand p_θ est un réseau de neurones.

3.4 Autoencodeurs variationnels

Rappelez-vous l'autoencodeur du Jour 1 : un encodeur $f_{\mathrm{enc}} : \mathbb{R}^d \to \mathbb{R}^k$ et un décodeur $f_{\mathrm{dec}} : \mathbb{R}^k \to \mathbb{R}^d$, avec $k < d$, entraînés à minimiser $\|\boldsymbol{x} - f_{\mathrm{dec}}(f_{\mathrm{enc}}(\boldsymbol{x}))\|^2$. L'autoencodeur apprend un espace latent, mais il n'a pas de notion de probabilité — on ne peut pas y échantillonner. L'autoencodeur variationnel (VAE, Kingma & Welling, 2014) est la cousine probabiliste.

3.4.1 Le modèle

Le VAE suppose que les données sont générées comme suit :

1. Échantillonner un vecteur latent $\boldsymbol{z}$ depuis un prior simple, typiquement $p(\boldsymbol{z}) = \mathcal{N}(\mathbf{0}, I)$.
2. Faire passer $\boldsymbol{z}$ par un décodeur neuronal f_{dec} pour obtenir les paramètres d'une distribution $p_\theta(\boldsymbol{x} \mid \boldsymbol{z})$ sur les données (par exemple, une gaussienne de moyenne $f_{\mathrm{dec}}(\boldsymbol{z})$).
3. Échantillonner $\boldsymbol{x}$ depuis $p_\theta(\boldsymbol{x} \mid \boldsymbol{z})$.

La distribution du modèle est la marginale $p_\theta(\boldsymbol{x}) = \int p_\theta(\boldsymbol{x} \mid \boldsymbol{z})\, p(\boldsymbol{z})\, d\boldsymbol{z}$. Cette intégrale est intractable pour un décodeur neuronal, donc on ne peut pas évaluer $\log p_\theta(\boldsymbol{x})$ directement. Le VAE introduit un second réseau de neurones — un *encodeur* $q_\phi(\boldsymbol{z} \mid \boldsymbol{x})$, typiquement gaussien de moyenne et variance produites par f_{enc} — et optimise une borne inférieure sur la log-vraisemblance.

3.4.2 La borne inférieure (ELBO)

Pour tout choix de q_ϕ,

$$\log p_\theta(\boldsymbol{x}) \geq \underbrace{\mathbb{E}_{q_\phi(\boldsymbol{z}\mid\boldsymbol{x})}[\log p_\theta(\boldsymbol{x} \mid \boldsymbol{z})]}_{\text{terme de reconstruction}} - \underbrace{D_{\mathrm{KL}}(q_\phi(\boldsymbol{z} \mid \boldsymbol{x}) \,\|\, p(\boldsymbol{z}))}_{\text{régularisation vers le prior}}.$$

C'est l'ELBO. Maximiser l'ELBO par rapport à θ et ϕ conjointement est l'objectif d'entraînement. Le premier terme veut que le décodeur reconstruise $\boldsymbol{x}$ correctement à partir d'un $\boldsymbol{z}$ tiré de l'encodeur. Le second terme veut que la postérieure de l'encodeur ressemble au prior.

Intuition

Le VAE est un autoencodeur avec deux changements. D'abord, l'encodeur produit une distribution, pas un seul $\boldsymbol{z}$ — on échantillonne pour obtenir le latent. Ensuite, un terme de perte supplémentaire tire les distributions de l'encodeur vers le prior gaussien standard. Ensemble, ces changements rendent l'espace latent *génératif* : des échantillons du prior, décodés, ressemblent à des données, parce que l'encodeur a été poussé à utiliser la même région de l'espace latent que celle couverte par le prior.

3.4.3 L'astuce de reparamétrisation

L'espérance $\mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x})}[\cdot]$ est prise sur un $\boldsymbol{z}$ aléatoire, et l'encodeur produit les paramètres de cet aléa. On ne peut pas différentier directement à travers un échantillonnage aléatoire. L'astuce (Kingma & Welling, 2014) : écrire $\boldsymbol{z} = \boldsymbol{\mu}_\phi(\boldsymbol{x}) + \boldsymbol{\sigma}_\phi(\boldsymbol{x}) \odot \boldsymbol{\epsilon}$ avec $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, I)$. Maintenant l'aléa est dans $\boldsymbol{\epsilon}$, qui n'a pas de paramètres ; $\boldsymbol{\mu}_\phi$ et $\boldsymbol{\sigma}_\phi$ sont déterministes et différentiables. La rétropropagation passe.

Le TP d'aujourd'hui entraîne un petit VAE sur MNIST et visualise à la fois l'espace latent et ce qui se passe quand on décode un chemin lisse à travers cet espace.

3.5 Diffusion débruitante

Les modèles de diffusion (Sohl-Dickstein et al., 2015 ; Ho et al., 2020) prennent une autre route vers la même destination. Au lieu d'essayer d'apprendre la distribution des données directement, ils apprennent à *inverser* un processus de bruitage fixe.

3.5.1 Le processus avant

Partez d'une vraie image $\boldsymbol{x}_0$. Définissez une suite de versions corrompues $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T$ en ajoutant une petite quantité de bruit gaussien à chaque étape :

$$\boldsymbol{x}_t = \sqrt{1-\beta_t}\,\boldsymbol{x}_{t-1} + \sqrt{\beta_t}\,\boldsymbol{\epsilon}_t, \qquad \boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, I).$$

Les variances β_t sont petites et choisies de sorte qu'à $t = T$ (souvent $T = 1000$), $\boldsymbol{x}_T$ soit essentiellement du bruit gaussien pur. Ce processus avant n'a aucun paramètre ; c'est juste un planning de bruit.

Une conséquence en forme close utile : pour tout t, on peut écrire $\boldsymbol{x}_t$ directement en fonction de $\boldsymbol{x}_0$ et d'un seul vecteur de bruit :

$$\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\,\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, I),$$

où $\bar{\alpha}_t = \prod_{s=1}^{t}(1-\beta_s)$. C'est ce qui rend l'entraînement efficace — pas besoin de simuler la chaîne pas à pas.

3.5.2 Le processus arrière

On entraîne un réseau de neurones $\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)$ qui prend une image bruitée et un pas de temps, et prédit le bruit qui a été ajouté. La perte d'entraînement est une simple MSE :

$$\mathcal{L}_{\text{diff}}(\theta) = \mathbb{E}_{\boldsymbol{x}_0,t,\boldsymbol{\epsilon}}\left[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\|^2\right]$$

où $\boldsymbol{x}_t$ est la version bruitée de $\boldsymbol{x}_0$ au pas t, donnée par la formule en forme close ci-dessus. Une fois que le réseau sait prédire le bruit, on peut le soustraire : étant donné un $\boldsymbol{x}_t$ bruité, on récupère une estimation de $\boldsymbol{x}_0$ (ou, plus précisément, de $\boldsymbol{x}_{t-1}$).

Note

L'objectif d'entraînement est juste une MSE sur la prédiction du bruit. Pas de log-vraisemblance, pas de divergence KL dans la perte, pas de borne variationnelle à négocier. L'avantage empirique de la diffusion sur les VAE tient en grande partie au fait que cette perte se comporte beaucoup mieux que l'ELBO. La raison profonde qui explique son succès vient du lien avec le *score matching* (Hyvärinen, 2005 ; Song & Ermon, 2019) : la prédiction du bruit est, à un facteur d'échelle connu près, une estimation du gradient de la log-densité de la distribution des données au point bruité.

3.5.3 Échantillonnage

Pour générer une nouvelle image, partez d'un bruit pur $\boldsymbol{x}_T \sim \mathcal{N}(\mathbf{0}, I)$ et exécutez le processus arrière : à chaque pas, utilisez le réseau pour prédire le bruit, puis avancez d'un pas vers une image propre. Après T pas, vous avez un échantillon de l'approximation par le modèle de p_{data}. Les échantillonneurs modernes (DDIM, Karras et al.) réduisent le nombre de pas requis de milliers à des dizaines, avec peu de perte de qualité.

Intuition

Un modèle de diffusion entraîné est un débruiteur qui sait à quoi les données ressemblent à chaque niveau de bruit. L'échantillonnage est juste un débruitage à l'envers, partant du bruit pur et finissant à quelque chose que le modèle considère comme un échantillon valide. Le TP d'aujourd'hui entraîne cette image à partir de zéro sur une spirale 2D, pour que vous puissiez regarder le bruit devenir une spirale en temps réel.

3.6 Pourquoi la probabilité change l'image

Les réseaux déterministes des Jours 1 et 2 étaient des approximateurs de fonctions. Les réseaux probabilistes du Jour 3 sont des approximateurs de distributions. La différence a des conséquences :

- Un classifieur vous dit la probabilité de chaque étiquette. Un modèle de diffusion vous dit la probabilité de chaque image possible.
- Un classifieur est interrogé une fois par entrée. Un modèle génératif est échantillonné — il produit quelque chose de différent à chaque appel.
- L'échec d'un classifieur, ce sont des prédictions fausses. L'échec d'un modèle génératif, c'est de produire des échantillons que la distribution des données ne produirait jamais, ou d'échouer à couvrir des régions qu'elle couvre.

Les objets mathématiques (perte, gradient, optimiseur) sont les mêmes. L'interprétation est différente, et les choix de design qui en découlent — quoi modéliser, sur quoi conditionner, comment échantillonner — sont différents.

3.7 Ce que vous avez vu

La perte d'entropie croisée du Jour 2 est une log-vraisemblance négative. Le maximum de vraisemblance est une minimisation approchée de la KL entre données et modèle. Le VAE optimise une borne inférieure tractable sur la log-vraisemblance et utilise une astuce d'échantillonnage reparamétré pour propager le gradient à travers l'aléa. Les modèles de diffusion contournent entièrement la vraisemblance, entraînant un débruiteur par simple MSE et échantillonnant en inversant le processus de bruitage.

Ces trois idées — vraisemblance, ELBO, score de débruitage — couvrent essentiellement tout modèle génératif utilisé en production : modèles de langue (vraisemblance du prochain jeton), Stable Diffusion (débruitage), VAE d'images (étapes de compression perceptuelle de la *latent diffusion*). Le Jour 4 revient sur une question que nous avons esquivée : quand les données ont une structure (graphes, séquences, images sur une grille régulière), quelles architectures encodent cette structure dans le modèle ?

Travail pratique

Ouvrez le notebook du Jour 3 (`Day3_Probability_GenerativeModels.ipynb`). Vous allez :

1. Échantillonner depuis une gaussienne, en ajuster une aux données, et calculer la KL entre deux gaussiennes à la main et par Monte Carlo — pour voir pourquoi la forme close compte.
2. Entraîner un petit VAE sur MNIST. Visualiser l'espace latent 2D, y échantillonner, et décoder un chemin entre deux latents pour regarder un chiffre se transformer en un autre.
3. Implémenter un modèle de diffusion débruitant à partir de zéro sur un jeu de données en spirale 2D. Visualiser le processus de bruitage avant et le processus de génération arrière pas à pas.

4. Entraîner un petit modèle de diffusion sur MNIST, échantillonner quelques chiffres, et inspecter les niveaux de bruit intermédiaires.

À la fin du TP, vous aurez construit deux modèles génératifs à partir de zéro et entraîné les deux avec succès sur des données réelles, en utilisant seulement NumPy, PyTorch, et la mathématique de ce chapitre.

Chapitre 4

Modélisation pour l'IA

> *« L'architecture que vous choisissez est le prior que vous placez sur quelles fonctions sont faciles à apprendre. Tout le reste est de l'optimisation d'hyper-paramètres par-dessus ce prior. »*

Les Jours 1 à 3 ont construit la machinerie : vecteurs, gradients, vraisemblances. Chaque modèle a été un MLP — le défaut d'approximation universelle, capable en principe de représenter n'importe quoi mais capable en pratique de représenter très peu avant que son nombre de paramètres n'explose. Les systèmes réels utilisent des CNN, des GNN, des Transformers, des U-Net, de l'attention à positions relatives. Ce ne sont pas des choix d'ingénierie arbitraires. Chaque architecture est un *prior* : un énoncé sur les types de fonctions qui sont plausibles, cuisiné dans le modèle avant qu'aucune donnée d'entraînement ne soit vue.

Ce chapitre traite de comment ce prior est encodé. Le nom général est *biais inductif*. Le vocabulaire technique précis est *invariance* et *équivariance*. La question pratique est : étant donné un problème réel, comment choisir — ou concevoir — une architecture dont les biais correspondent à la structure de vos données ?

4.1 Biais inductif : le prior qu'une architecture encode

Deux modèles peuvent avoir le même pouvoir expressif — le même ensemble de fonctions qu'ils peuvent en principe représenter — et pourtant apprendre très différemment des mêmes données. La différence, c'est avec quelle facilité chacun peut trouver une bonne fonction par descente de gradient. Cette différence est le biais inductif.

Définition 4.1 (Biais inductif)**.** *L'ensemble des hypothèses, encodées dans l'architecture et dans la perte, qui déterminent quelles fonctions le modèle préfère en l'absence de données. De manière équivalente : le prior sur les fonctions induit par l'entraînement du modèle.*

Le théorème d'approximation universelle du Jour 1 dit qu'un MLP à une couche cachée peut en principe approximer toute fonction continue. Il ne dit rien sur combien de paramètres cela demande, ni sur combien de données sont nécessaires pour trouver la bonne

fonction. Pour un problème de classification d'images avec 10^9 arrangements de pixels, un MLP de largeur suffisante existe — vous n'aurez ni assez de données ni assez de temps pour le trouver. Un CNN, avec le bon biais inductif, le trouve.

Intuition

Le biais inductif, c'est la différence entre *peut représenter* et *peut apprendre à partir de données réalistes*. L'architecture est la partie du modèle où vous, le concepteur, êtes autorisé à dire : « Je sais quelque chose sur ce problème. Voici la structure. »

4.2 Contraintes et régularisation

Le premier type de biais est celui que nous avons introduit la semaine passée sans le nommer : la régularisation. L'objectif d'entraînement non contraint est juste la perte sur les données d'entraînement. Ajouter une pénalité change le problème :

$$\min_{\theta} \; \mathcal{L}(\theta) + \lambda \, R(\theta).$$

R encode quels types de θ nous préférons en l'absence de données — typiquement des paramètres simples, à norme petite, ou parcimonieux. λ contrôle à quel point cette préférence domine.

Régularisation L^2 (*weight decay*). $R(\theta) = \|\theta\|^2$. Tire les poids vers zéro. Équivaut, sous une interprétation bayésienne, à un prior gaussien sur les poids.

Régularisation L^1. $R(\theta) = \|\theta\|_1$. Tire les poids vers zéro et tend à les rendre exactement nuls. Utile quand la parcimonie est réellement significative (sélection de variables, *compressed sensing*).

Dropout (Srivastava et al., 2014). À chaque pas d'entraînement, mettre aléatoirement une fraction des activations cachées à zéro. Le modèle entraîné se comporte comme un ensemble. Ce n'est pas une pénalité dans la perte, mais joue le même rôle en pratique.

La régularisation est la forme la plus douce du biais inductif : elle change le paysage de la perte, elle ne change pas le modèle. Les deux sections suivantes décrivent des biais qui changent le modèle lui-même.

4.3 Invariance et équivariance

Le second type de biais est structurel. Il dit : *la fonction que nous voulons apprendre a une symétrie, et l'architecture doit respecter cette symétrie.*

Définition 4.2 (Invariance et équivariance)**.** *Une fonction f est* invariante *sous une transformation g si $f(g(\boldsymbol{x})) = f(\boldsymbol{x})$ pour tout $\boldsymbol{x}$. Elle est* équivariante *si $f(g(\boldsymbol{x})) = g(f(\boldsymbol{x}))$: appliquer la transformation à l'entrée revient à l'appliquer à la sortie.*

Les tâches de classification sont typiquement invariantes sous des transformations qui préservent l'identité : une image de chat reste un chat après translation, une phrase reste « sentiment positif » après passage à des synonymes, une molécule garde la même activité après rotation. La fonction que nous voulons apprendre doit respecter cette invariance, et une architecture qui l'intègre n'a pas à l'apprendre à partir des données.

Beaucoup de couches intermédiaires sont équivariantes plutôt qu'invariantes : une carte de caractéristiques doit se translater quand son entrée se translate, pour qu'une couche ultérieure puisse poolinger les caractéristiques translatées en quelque chose d'invariant. La recette standard est une pile de couches équivariantes terminée par une étape de pooling invariant.

4.4 CNN : équivariance par translation pour les grilles

Les données d'image vivent sur une grille régulière. Un chat translaté de dix pixels vers la droite est toujours un chat. Si nous voulons que l'architecture le sache, il faut l'équivariance par translation.

Proposition 4.1 (La convolution est équivariante par translation)**.** *Soit T_d la translation de d pixels et K un filtre convolutionnel. Alors $K \star (T_d \boldsymbol{x}) = T_d(K \star \boldsymbol{x})$ pour tout $\boldsymbol{x}$ et tout d. L'opération de convolution commute avec la translation ; par conséquent, toute fonction construite uniquement de convolutions et de non-linéarités ponctuelles est équivariante par translation.*

Une couche convolutionnelle avec des filtres $k \times k$ a $k \cdot k \cdot C_{\text{in}} \cdot C_{\text{out}}$ paramètres, quelle que soit la taille de l'image. Une couche MLP projetant la même image vers la même sortie aurait $H \cdot W \cdot C_{\text{in}} \cdot H \cdot W \cdot C_{\text{out}}$ paramètres — quatre ordres de grandeur de plus pour une image typique. La convolution est dramatiquement plus efficace en paramètres *parce que* les mêmes poids sont partagés à travers toutes les positions spatiales, ce qui est exactement ce que demande l'équivariance par translation.

Note

Les réseaux convolutionnels n'ont pas été conçus en ajustant des hyperparamètres jusqu'à ce que quelque chose marche sur ImageNet. Ils ont été conçus par Yann LeCun dans les années 1980 en partant de l'équivariance par translation et du partage de poids comme principes. Le succès des CNN sur les images est une confirmation que le bon biais inductif vaut plus que la seule échelle des données — bien que depuis 2020, les Transformers avec assez de données et assez d'échelle aient montré que même des biais inductifs très faibles peuvent être surmontés par un entraînement suffisant (Dosovitskiy et al., 2021).

4.5 GNN : équivariance par permutation pour les graphes

Les données de graphes ont une symétrie différente. Un graphe est un ensemble de nœuds connectés par des arêtes. Renuméroter les nœuds ($1 \to 7, 2 \to 3, \ldots$) donne le même

graphe en tant qu'objet mathématique. Une fonction sur les graphes doit traiter deux renumérotations comme équivalentes.

Définition 4.3 (Équivariance par permutation pour les caractéristiques de nœuds)**.** *Soit P une matrice de permutation agissant sur les nœuds d'un graphe. Une fonction f sur les caractéristiques de nœuds est équivariante par permutation si $f(P\boldsymbol{X}, PAP^\top) = Pf(\boldsymbol{X}, A)$, où $\boldsymbol{X}$ est la matrice de caractéristiques de nœuds et A est la matrice d'adjacence.*

Les réseaux de neurones sur graphes à passage de messages (Gilmer et al., 2017) atteignent cela en calculant la nouvelle représentation de chaque nœud comme une agrégation des représentations actuelles de ses voisins, en utilisant seulement des agrégations invariantes par permutation (somme, moyenne, max). La mise à jour pour le nœud v à la couche $\ell+1$ est :

$$\boldsymbol{h}_v^{(\ell+1)} = \phi\left(\boldsymbol{h}_v^{(\ell)}, \bigoplus_{u \in \mathcal{N}(v)} \psi(\boldsymbol{h}_v^{(\ell)}, \boldsymbol{h}_u^{(\ell)})\right)$$

où $\bigoplus$ est un agrégateur invariant par permutation (le plus souvent somme ou moyenne), et ϕ, ψ sont de petits réseaux de neurones (typiquement des MLP) à paramètres appris.

Pour des sorties au niveau du graphe (classer le graphe entier), un dernier pooling invariant par permutation est ajouté : sommer ou moyenner les représentations de nœuds.

4.6 Attention et Transformers

Un Transformer (Vaswani et al., 2017) traite l'entrée comme un ensemble non ordonné de jetons et calcule des interactions par paires via l'attention. L'opération d'attention est elle-même équivariante par permutation — l'encodage positionnel est ajouté précisément parce qu'on veut habituellement que le modèle connaisse l'ordre, et l'attention pure serait aveugle à l'ordre.

Définition 4.4 (Auto-attention)**.** *Étant donnée une séquence de vecteurs $\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]$, l'auto-attention calcule :*

$$\boldsymbol{Q} = \boldsymbol{X}W_Q, \quad \boldsymbol{K} = \boldsymbol{X}W_K, \quad \boldsymbol{V} = \boldsymbol{X}W_V, \quad \boldsymbol{A} = \mathrm{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}}\right), \quad \boldsymbol{Y} = \boldsymbol{A}\boldsymbol{V}.$$

$\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ sont des projections linéaires de l'entrée. $\boldsymbol{A}$ est une matrice $n \times n$ de poids par paires. Chaque sortie est une moyenne pondérée de toutes les entrées, où les poids dépendent du contenu.

Intuition

La convolution agrège l'information depuis des voisins spatiaux avec des poids fixes. L'attention agrège l'information depuis *toutes* les positions avec des poids qui dépendent du contenu. Une couche convolutionnelle demande « qu'y a-t-il ici ? » L'attention demande « qu'est-ce qui ici ressemble à ce que je cherche, et où ? » La première est structurelle, la seconde est associative. Les deux s'avèrent utiles, et l'image moderne (Vision Transformers, architectures hybrides, MoE) consiste

largement à les combiner.

4.7 Choisir une architecture : l'exercice de correspondance

Étant donné un problème réel, le mouvement de conception consiste à identifier les symétries des données et à choisir une architecture dont les biais inductifs leur correspondent :

- **Grille régulière avec symétrie de translation** (images, spectrogrammes audio, capteurs denses) → CNN.
- **Séquence où l'ordre compte** (texte, séries temporelles, formes d'onde audio) → RNN historiquement, Transformer en pratique aujourd'hui.
- **Graphe ou ensemble** (molécules, réseaux sociaux, nuages de points, systèmes de particules) → GNN, DeepSets, réseaux pour nuages de points.
- **Ensemble avec équivariance par translation/rotation** (molécules 3D, systèmes physiques avec lois de conservation) → réseaux équivariants (SE(3)-Transformers, EGNN).
- **Caractéristiques tabulaires hétérogènes** (lignes de base de données) → MLP ou arbres boostés. Le prior du deep learning vous rapporte peu ici.

Là où il n'y a pas de symétrie évidente, un MLP est le défaut sûr, et les arbres (XGBoost, LightGBM) battent fréquemment tout réseau de neurones. Le bon biais inductif fait un vrai travail ; le mauvais biais inductif est juste une contrainte arbitraire qui vous ralentit.

4.8 Traduire un problème réel en problème d'apprentissage

Le choix d'architecture est l'une de quatre décisions interdépendantes à prendre au moment de cadrer un problème d'apprentissage :

1. **Entrées.** Quelle information le modèle voit-il réellement ? Brute, ou featurisée ? À quelle granularité (pixel, jeton, transaction, jour) ?
2. **Cibles.** Que le modèle essaie-t-il de prédire ? Une classe ? Un réel ? Une distribution ? Une séquence ?
3. **Perte.** Comment mesure-t-on « faux » ? Entropie croisée, MSE, contrastif, surmesure ?

4. **Métrique d'évaluation.** Comment déclare-t-on le succès ? Souvent *pas* la même chose que la perte — la précision n'est pas différentiable, mais c'est ce qui intéresse l'utilisateur.

Quand un projet échoue, l'architecture n'est généralement pas le problème. Le problème est généralement que l'une de ces quatre était mauvaise. L'exemple célèbre : un modèle entraîné à prédire une pneumonie depuis des radios thoraciques qui a appris à prédire la machine à radio de l'hôpital, parce que cette information a fuité dans l'entrée. Mauvaises entrées. Solution différente de « ajouter plus de couches ».

Intuition

L'architecture est la plus petite des quatre décisions. Entrées, cibles, perte et métrique déterminent si le problème est bien posé ; l'architecture ne fait que déterminer si un problème bien posé peut être résolu efficacement. Une longue carrière en ML est surtout une question de bien cadrer — pas une question de choisir le bon nombre de couches.

4.9 Ce que vous avez vu

Chaque réseau de neurones est un approximateur de fonctions avec un prior intégré — le biais inductif. Les CNN encodent l'équivariance par translation pour des données en grille ; les GNN encodent l'équivariance par permutation pour les graphes ; les Transformers remplacent les priors structurels par une agrégation pondérée basée sur le contenu. Choisir une architecture, c'est choisir un prior, et ce choix compte bien plus que la taille du réseau. Autour de ce choix se trouve le problème plus vaste du cadrage : entrées, cibles, perte et métrique d'évaluation doivent toutes s'aligner avant qu'aucune architecture ne puisse faire un travail utile.

Le Jour 5, le chapitre final, pose la question que nous avons remise depuis le début : comment savons-nous que le modèle apprend réellement la bonne chose, et ne mémorise pas le jeu de données ou n'exploite pas un artefact ?

Travail pratique

Ouvrez le notebook du Jour 4 (`Day4_Modeling_InductiveBias.ipynb`). Vous allez :

1. Entraîner un MLP et un CNN avec des comptes de paramètres comparables sur MNIST, puis évaluer les deux sur une version translatée de l'ensemble de test. Vous verrez l'équivariance par translation du CNN payer quand l'entraînement et le test ne correspondent plus exactement.

2. Implémenter à la main un petit GNN à passage de messages et l'appliquer à un problème synthétique de classification de nœuds. Comparer à un MLP qui ignore la structure du graphe et n'utilise que les caractéristiques des nœuds.

3. Calculer l'auto-attention à partir de zéro sur une petite séquence jouet. Visualiser les poids d'attention en carte de chaleur et voir comment les projections requête/clé/valeur produisent une sélection basée sur le contenu.

À la fin du TP, vous aurez une preuve directe de l'affirmation centrale de ce chapitre — que le bon biais inductif fait un travail même avant qu'aucun entraînement ne commence.

Chapitre 5

Évaluer les modèles d'IA

> *« Un modèle à 99 % de précision sur le test n'est pas terminé. Un modèle est terminé quand vous pouvez défendre, devant une partie prenante, à la fois les cas où il fonctionne et les cas où il échoue. »*

Nous avons un modèle entraîné. C'est un empilement d'applications affines et de non-linéarités (Jour 1), ajusté aux données par descente de gradient sur une perte dérivée d'une vraisemblance (Jours 2 et 3), avec une architecture choisie pour correspondre à la structure des données (Jour 4). La perte d'entraînement est petite. La perte de test est petite. Le modèle est-il terminé ?

La réponse honnête est : généralement non. L'évaluation est la partie de l'apprentissage automatique qui reçoit le moins d'attention dans les cours et le plus d'attention dans les vrais projets. Chaque échec en production d'un système d'IA des dix dernières années — une voiture autonome qui n'a pas reconnu un camion blanc contre un ciel lumineux, un outil de recrutement qui a systématiquement déclassé les femmes, un modèle médical qui a appris à prédire l'hôpital au lieu de la maladie — est une histoire d'évaluation qui a mal tourné. Ce chapitre traite de comment faire l'évaluation correctement, à quoi ressemblent les modes d'échec standards, et comment décider si un modèle doit être déployé ou retourner à la planche à dessin.

5.1 Ce que la précision sur le test mesure réellement

Une précision sur le test est un nombre. Elle mesure la fraction de prédictions correctes sur un seul ensemble mis de côté, tiré de la même distribution que l'ensemble d'entraînement, évaluée avec la même métrique utilisée pour la sélection du modèle. Chacune de ces qualifications compte :

- **Un nombre.** Un seul scalaire écrase tout ce qui pourrait vous intéresser en un seul résumé. Où le modèle échoue-t-il ? Sur quelles classes ? Sous quelles entrées ? À quels seuils ? La précision sur le test ne dit rien.
- **Un seul ensemble mis de côté.** La fluctuation entre différents ensembles de test est réelle, particulièrement pour des ensembles de test petits. Un 92 % rapporté peut devenir 87 % la semaine prochaine.

- **Même distribution.** Des données de test tirées de la même distribution que les données d'entraînement vous renseignent sur l'*interpolation*, pas sur la généralisation aux cas qui vous intéressent en déploiement.
- **Même métrique utilisée pour la sélection du modèle.** Si vous avez choisi le modèle qui maximise la précision sur le test sur de nombreux essais, l'ensemble de test est devenu en pratique un ensemble d'entraînement, et vous l'avez sur-ajusté.

Intuition

La précision sur le test est une statistique nécessaire, pas suffisante. Elle vous dit que le modèle ajuste l'ensemble de test. Elle ne vous dit pas que le modèle a appris la tâche sous-jacente. Distinguer ces deux est essentiellement l'objet de ce chapitre.

5.2 La décomposition biais-variance

Pour un problème de régression avec perte d'erreur quadratique et un point de test fixe $\boldsymbol{x}_0$, la perte attendue d'un modèle $\hat{f}(\boldsymbol{x}_0)$ entraîné sur un ensemble d'entraînement aléatoire se décompose comme :

$$\mathbb{E}\big[(y_0 - \hat{f}(\boldsymbol{x}_0))^2\big] = \underbrace{(f(\boldsymbol{x}_0) - \mathbb{E}[\hat{f}(\boldsymbol{x}_0)])^2}_{\text{biais}^2} + \underbrace{\mathrm{Var}(\hat{f}(\boldsymbol{x}_0))}_{\text{variance}} + \underbrace{\sigma^2}_{\text{bruit irréductible}}$$

où f est la vraie fonction inconnue et σ^2 est la variance du bruit d'étiquette. L'espérance est prise à la fois sur l'ensemble d'entraînement aléatoire et sur l'étiquette de test aléatoire.

- **Biais** : à quel point le modèle se trompe, en moyenne, à travers de nombreuses exécutions d'entraînement. Un modèle trop simple pour capturer la vérité a un biais élevé — sous-apprentissage.
- **Variance** : à quel point le modèle fluctue d'une exécution d'entraînement à l'autre. Un modèle trop flexible par rapport à la taille du jeu de données a une variance élevée — sur-apprentissage.
- **Bruit irréductible** : ce qu'aucun modèle ne peut réduire, parce que l'étiquette elle-même est bruitée.

L'image classique dit : à mesure que la capacité du modèle augmente, le biais diminue et la variance augmente. Le compromis idéal se situe quelque part au milieu. Le TP d'aujourd'hui produit cette image explicitement en ajustant des polynômes de degré variable à un petit jeu de données.

Note

Pour les grands réseaux de neurones, l'image classique s'effondre. Le phénomène de *double descente* (Belkin et al., 2019) montre que l'erreur de test monte d'abord puis redescend à mesure que la capacité du modèle dépasse la taille du jeu de données. Le

deep learning moderne opère dans un régime où la capacité supplémentaire ne nuit pas systématiquement à la généralisation. C'est empirique, pas encore pleinement compris théoriquement, et c'est l'une des questions ouvertes du champ.

5.3 Découpages : entraînement, validation, test, et à quoi sert chacun

Pour estimer la généralisation sans contaminer le processus de sélection du modèle, la discipline standard consiste en trois jeux de données disjoints :

- **Ensemble d'entraînement** ($\sim$ 60–80 %). Les données que le modèle voit et ajuste.
- **Ensemble de validation** ($\sim$ 10–20 %). Les données utilisées pour choisir entre modèles candidats, hyperparamètres, architectures. Mises de côté de l'entraînement, mais le modélisateur interagit beaucoup avec elles.
- **Ensemble de test** ($\sim$ 10–20 %). Les données utilisées pour estimer la généralisation finale. Touchées une seule fois, après que toutes les décisions ont été prises.

⚠ Attention

L'ensemble de test est contaminé dès l'instant où vous prenez une décision basée sur sa performance. « Nous avons essayé 50 architectures et choisi celle avec la meilleure précision sur le test » a en pratique transformé l'ensemble de test en ensemble de validation. La précision sur le test rapportée est alors biaisée à la hausse, souvent substantiellement. La solution est la discipline — un vrai ensemble mis de côté que le modélisateur ne regarde pas avant la toute fin — et, lorsque c'est possible, un ensemble de test externe sur lequel l'équipe ne peut pas itérer.

Pour les petits jeux de données, la validation croisée à k plis fait tourner le rôle de validation à travers différents découpages pour que chaque exemple serve de validation une fois. Cela donne une meilleure estimation de la généralisation au prix de k fois plus d'entraînement. Pour des jeux de données au-delà de quelques dizaines de milliers d'exemples, un seul découpage entraînement/val/test suffit généralement.

5.4 Calibration : confiant n'est pas la même chose que juste

Un classifieur renvoie une probabilité par classe. « 90 % » devrait signifier que parmi toutes les prédictions faites avec 90 % de confiance, 90 % sont correctes. La *calibration* est de savoir si c'est vrai. Les réseaux profonds modernes ont tendance à être *trop confiants* : ils disent 99 % quand ils devraient dire 80 %.

Définition 5.1 (Diagramme de fiabilité)**.** *Grouper les prédictions en classes (*bins*) selon leur confiance (par exemple, 0,0–0,1, 0,1–0,2, …). Pour chaque classe, tracer (confiance moyenne dans la classe, fraction correcte dans la classe). Un modèle parfaitement calibré se trouve sur la diagonale.*

L'Erreur de Calibration Attendue (ECE) est la distance moyenne à la diagonale, pondérée par la population de chaque classe. C'est un seul nombre qui résume à quel point le modèle est mal calibré.

Intuition

La calibration compte quand la décision en aval dépend de la probabilité, pas seulement de l'argmax. Un système de triage médical qui signale « 70 % de probabilité de sepsis » a intérêt à le penser vraiment — le médecin traitera différemment que pour « 95 % de probabilité ». Un *ranker* de recherche peut être terriblement mal calibré sans que personne ne le remarque, parce que seul l'ordre compte. Décidez si votre application a besoin de probabilités calibrées, et mesurez si oui.

L'échelle de température (*temperature scaling*, Guo et al., 2017) est la solution la plus simple. Après entraînement, diviser les logits pré-softmax par un scalaire de température T, choisi sur un ensemble de validation pour minimiser la log-vraisemblance négative. Un scalaire. Règle la plupart des problèmes de calibration des réseaux modernes sans ré-entraînement.

5.5 Décalage de distribution : quand l'ensemble de test n'est pas l'ensemble de déploiement

La précision sur le test estime la performance sur des données *de la même distribution que les données d'entraînement.* Les modèles déployés voient des données d'une distribution différente — parce que le monde change, parce que le processus de collecte de données est différent, parce que les utilisateurs se comportent différemment de ce que le jeu de données suggérait.

Trois décalages courants :

- **Décalage de covariable.** $p_{\text{deploy}}(\boldsymbol{x}) \neq p_{\text{train}}(\boldsymbol{x})$, mais $p(y \mid \boldsymbol{x})$ est inchangée. Les types d'entrées changent ; la règle d'étiquetage ne change pas. Les changements de qualité de caméra entre entraînement et déploiement en sont un exemple courant.
- **Décalage d'étiquette.** $p_{\text{deploy}}(y) \neq p_{\text{train}}(y)$, mais $p(\boldsymbol{x} \mid y)$ est inchangée. La prévalence des classes change ; l'apparence de chaque classe ne change pas. Un modèle entraîné sur un jeu de données équilibré et déployé sur des données déséquilibrées échoue de cette façon.
- **Décalage de concept.** $p(y \mid \boldsymbol{x})$ lui-même change. La règle d'étiquetage a changé. Les systèmes de recommandation en souffrent constamment — ce que les utilisateurs *veulent* cette année n'est pas ce qu'ils voulaient l'an dernier.

Le TP d'aujourd'hui produit une expérience de décalage de covariable en faisant tourner l'ensemble de test et en mesurant la chute de précision. La chute est dramatique pour le MLP du Jour 4, moindre pour le CNN. C'est exactement l'histoire du biais inductif d'hier, racontée du côté de l'évaluation.

5.6 Exemples adversariaux et détection hors distribution

Certains échecs sont plus pointus que le décalage graduel de distribution. Les exemples adversariaux (Szegedy et al., 2014 ; Goodfellow et al., 2015) sont des entrées construites spécifiquement pour tromper le modèle — des perturbations imperceptiblement petites qui transforment une prédiction correcte confiante en une prédiction incorrecte confiante. Elles révèlent que le modèle a appris une fonction qui s'accorde avec les données sur les entrées naturelles mais en diverge ailleurs.

Les entrées hors distribution (OOD) ne sont pas adversariales, juste non familières : un modèle entraîné sur des chats et des chiens à qui l'on demande de classer un poisson. Un modèle bien élevé devrait signaler « je ne sais pas ». La plupart des réseaux modernes font l'inverse : ils attribuent confiamment l'une des classes entraînées.

Intuition

Un modèle entraîné sur un ensemble fermé de classes n'a aucune représentation de « autre ». Détecter qu'une entrée est OOD est un problème séparé, typiquement abordé en examinant la confiance softmax (une confiance basse signifie parfois OOD), l'énergie des logits, ou la distance dans l'espace de caractéristiques à l'exemple d'entraînement le plus proche. Le TP d'aujourd'hui montre le mode d'échec en utilisant un détecteur basé sur la confiance softmax appliqué à des entrées Fashion-MNIST sur un classifieur entraîné sur MNIST.

5.7 Le cadre de décision : déployer, collecter, ou reconcevoir

L'évaluation produit une décision. Il y en a trois :

1. **Déployer.** La performance sur le test est acceptable sur la population pertinente, la calibration est adéquate pour l'usage en aval, et les modes d'échec que vous avez mesurés sont tolérables. Déployer avec surveillance.

2. **Collecter plus de données.** Le modèle a la bonne structure mais pas assez d'exemples. Les courbes de validation croisée montent encore ; la perte de test plafonne pour la bonne raison. Ajouter des données étiquetées et continuer.

3. **Reconcevoir.** Le modèle est faux sur quelque chose de structurel — l'architecture ne correspond pas à la tâche, la perte ne capture pas ce que les utilisateurs veulent,

les cibles ne sont pas ce qui était censé être appris. Aucune quantité de données ne corrige cela. Retourner au cadrage.

Les deux modes d'échec de cette décision : *déployer quand il faudrait reconcevoir* (le modèle semble bien sur l'ensemble de test mais résout le mauvais problème) et *collecter quand il faudrait déployer* (le perfectionnisme retarde un modèle qui serait déjà utile). La vraie discipline d'évaluation consiste à distinguer ces cas, et la seule façon de le faire est de connaître les modes d'échec de votre modèle assez bien pour pouvoir défendre la décision devant une partie prenante sceptique.

5.8 Ce que nous avons construit au cours des cinq jours

L'atelier a commencé par des vecteurs. Un modèle en IA moderne est une fonction d'un espace vectoriel vers un autre, construite à partir d'applications affines et de non-linéarités (Jour 1). Il apprend par descente de gradient sur une fonction de perte qui résume à quel point il se trompe ; la règle de la chaîne rend ce calcul faisable pour des réseaux à un million de paramètres (Jour 2). Quand le but est la génération plutôt que la prédiction, la perte devient une vraisemblance et le modèle devient une distribution dont on peut tirer des échantillons — l'image unificatrice derrière les VAE, les modèles de langue et la diffusion (Jour 3). L'architecture encode un prior sur quelles fonctions sont faciles à apprendre ; CNN, GNN et Transformers sont trois priors différents pour trois types de données différents (Jour 4). Et l'évaluation est la discipline qui nous dit si tout ce qui précède fait réellement la bonne chose, ou seulement quelque chose qui semble juste sur un ensemble de test (Jour 5).

Les mêmes cinq idées — vecteurs et applications linéaires, apprentissage par gradient, vraisemblance, biais inductif, discipline d'évaluation — font tourner chaque modèle utilisé en production aujourd'hui, d'une régression logistique aux systèmes de classe GPT. Les implémentations grandissent ; les structures ne changent pas. Ce que vous avez construit au cours de ces cinq jours, c'est le vocabulaire pour lire un article, évaluer un fournisseur, auditer un système déployé, et décider quand un outil d'IA est la bonne réponse au problème devant vous.

Travail pratique

Ouvrez le notebook du Jour 5 (`Day5_Evaluation_Generalization.ipynb`). Vous allez :

1. Ajuster des polynômes de degré 1, 3, 9, 15 au même petit jeu de données, répété sur de nombreux ensembles d'entraînement aléatoires. Tracer biais et variance séparément pour voir la décomposition à l'œuvre.

2. Entraîner un classifieur sur MNIST et produire son diagramme de fiabilité. Calculer l'Erreur de Calibration Attendue. Appliquer l'échelle de température sur un découpage de validation et remesurer.

3. Prendre un CNN entraîné sur MNIST et l'évaluer sur MNIST tourné. Mesurer la chute de précision à mesure que l'angle de rotation augmente.

4. Utiliser la confiance softmax du même CNN comme détecteur hors distribution sur Fashion-MNIST. Tracer la distribution des confiances pour les entrées en distribution vs OOD et voir à quel point elles sont séparables (ou non).

À la fin du TP, vous aurez vu, avec vos propres chiffres, chacun des modes d'échec nommés dans ce chapitre, et vous aurez les outils pour les détecter dans vos propres modèles.

Annexe A : Installation de l'environnement

Option 1 : Google Colab (recommandée)

Aller sur https://colab.research.google.com. Sélectionner **Exécution > Modifier le type d'exécution > GPU T4** pour les TPs sur les modèles de diffusion (Jour 3) et l'apprentissage sur graphes (Jour 4). Toutes les bibliothèques s'installent depuis les cellules du notebook avec `pip install`.

Option 2 : Installation locale

Python 3.10 ou plus récent. Pour le TP de diffusion (Jour 3) et le TP d'apprentissage sur graphes (Jour 4), un GPU NVIDIA avec 6+ Go de VRAM accélère l'entraînement, mais chaque TP tourne en CPU en moins de 15 minutes.

Bibliothèques principales

```
pip install torch torchvision numpy matplotlib scikit-learn
pip install jupyter notebook
```

Annexe B : Bibliographie

- Goodfellow, Bengio, Courville, *Deep Learning* (MIT Press, accès libre).
- Bishop & Bishop, *Deep Learning : Foundations and Concepts* (Springer, 2024).
- Murphy, *Probabilistic Machine Learning* (MIT Press, accès libre).
- Strang, *Linear Algebra and Learning from Data* (Wellesley-Cambridge, 2019).
- Ho, Jain, Abbeel, « Denoising Diffusion Probabilistic Models », NeurIPS 2020, https://arxiv.org/abs/2006.11239.
- Vaswani et al., « Attention Is All You Need », NeurIPS 2017, https://arxiv.org/abs/1706.03762.