Apprentissage Profond

Notes de Cours

M1–M2 — 2025–2026

Yaë Ulrich Gaba

« L'apprentissage profond permet à des modèles computationnels composés de multiples couches de traitement d'apprendre des représentations de données avec plusieurs niveaux d'abstraction. » — Yann LeCun

25 mars 2026

Table des matières

Préface

L'apprentissage profond a transformé de manière fondamentale notre approche de l'intelligence artificielle. En quelques années, les réseaux de neurones profonds sont passés de curiosités académiques à des outils omniprésents : reconnaissance d'images, traduction automatique, génération de texte, découverte de médicaments, prédiction de structures protéiques.

Derrière ces avancées spectaculaires se cachent des fondements mathématiques profonds. L'apprentissage profond se situe au carrefour de l'algèbre linéaire, de l'analyse, des probabilités et de l'optimisation. Comprendre ces fondements est essentiel pour concevoir de nouvelles architectures, diagnostiquer des problèmes d'entraînement et repousser les limites de l'état de l'art.

Ce cours s'adresse aux étudiants de Master en mathématiques, apprentissage automatique et science des données. Il suppose une maîtrise de l'algèbre linéaire, des probabilités et des fondements de l'apprentissage automatique. Chaque chapitre suit une progression rigoureuse :

1. Dérivation mathématique des résultats clés.
2. Implémentation en PyTorch avec boucles d'entraînement complètes.
3. Analyse des hyperparamÉtres et études d'ablation.
4. Connexion avec l'état de l'art actuel.
5. Exercices gradués : mathématique ($\star$), implémentation ($\star\star$), projet de recherche ($\star\star\star$).

Références principales.

- Goodfellow, Bengio & Courville — *Deep Learning*, MIT Press, 2016 (disponible gratuitement en ligne).
- Bishop & Bishop — *Deep Learning : Foundations and Concepts*, Springer, 2024.
- Zhang, Lipton, Li & Smola — *Dive into Deep Learning*, d2l.ai (interactif).
- LeCun, Bengio & Hinton — « Deep learning », *Nature*, 2015.

Chapitre 1

Réseaux de neurones artificiels — Fondements

Intuition

Un réseau de neurones artificiel est une fonction paramétrique qui apprend à transformer des entrées en sorties en ajustant progressivement ses paramètres. Comme un enfant qui apprend à reconnaître des chats en voyant des milliers d'images, le réseau affine ses représentations internes à chaque exemple.

1.1 Le neurone formel

1.1.1 Modèle mathématique

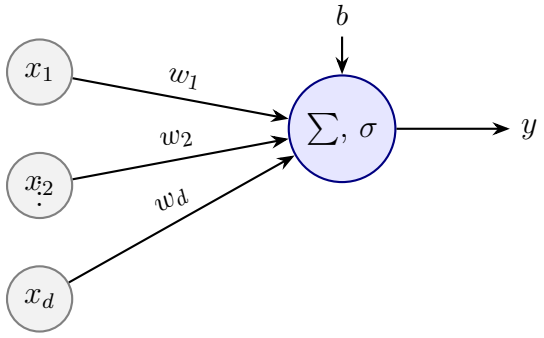
Le neurone formel, inspiré du modèle de McCulloch-Pitts (1943), calcule une somme pondérée de ses entrées suivie d'une fonction d'activation :

Neurone formel

Soit $\boldsymbol{x} = (x_1, \ldots, x_d)^\top \in \mathbb{R}^d$ le vecteur d'entrée, $\boldsymbol{w} = (w_1, \ldots, w_d)^\top \in \mathbb{R}^d$ le vecteur de poids et $b \in \mathbb{R}$ le biais. La sortie du neurone est :

$$y = \sigma\left(\sum_{i=1}^{d} w_i x_i + b\right) = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b)$$

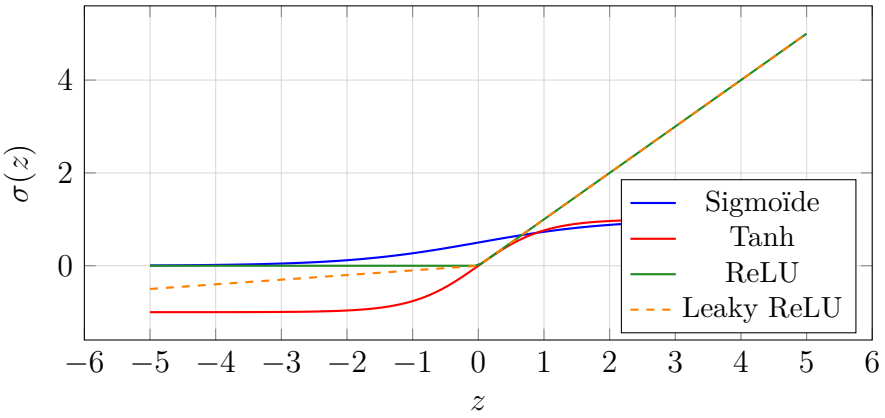
où $\sigma : \mathbb{R} \to \mathbb{R}$ est la fonction d'activation.



1.1.2 Fonctions d'activation

Les fonctions d'activation introduisent la non-linéarité essentielle au pouvoir expressif du réseau :

Nom	Formule	Image	Usage
Sigmoïde	$\sigma(z) = \frac{1}{1+e^{-z}}$	$(0,1)$	Sortie binaire
Tanh	$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$(-1,1)$	Couches cachées (historique)
ReLU	$\mathrm{ReLU}(z) = \max(0,z)$	$[0,+\infty)$	Standard moderne
Leaky ReLU	$\max(\alpha z, z)$, $\alpha \approx 0.01$	$\mathbb{R}$	Évite neurones morts
GELU	$z \cdot \Phi(z)$	$\mathbb{R}$	Transformers
Softmax	$\frac{e^{z_i}}{\sum_j e^{z_j}}$	$(0,1)^K$	Classification multi-classe



Attention

La sigmoïde souffre du problème de **gradient évanescent** : pour $|z| \gg 0$, $\sigma'(z) \approx 0$, ce qui bloque l'apprentissage dans les couches profondes. C'est pourquoi ReLU est aujourd'hui le choix par défaut.

1.2 Perceptron multicouche (MLP)

1.2.1 Architecture

Un perceptron multicouche empile L couches de neurones. La couche ℓ effectue la transformation :

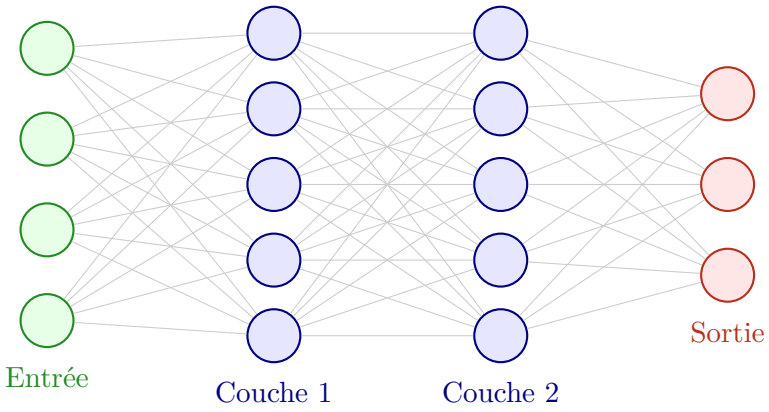
$$\boldsymbol{h}^{(\ell)} = \sigma^{(\ell)}\big(W^{(\ell)}\boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}\big), \quad \ell = 1, \dots, L$$

où $\boldsymbol{h}^{(0)} = \boldsymbol{x}$ est l'entrée et $\boldsymbol{h}^{(L)} = \hat{\boldsymbol{y}}$ est la sortie.

Définition 1.1 (Réseau entièrement connecté)**.** Un MLP à L couches de largeurs $n_0, n_1, \dots, n_L$ est la composition :

$$f_\theta(\boldsymbol{x}) = \sigma^{(L)} \circ A^{(L)} \circ \sigma^{(L-1)} \circ A^{(L-1)} \circ \cdots \circ \sigma^{(1)} \circ A^{(1)}(\boldsymbol{x})$$

où $A^{(\ell)}(\boldsymbol{z}) = W^{(\ell)}\boldsymbol{z} + \boldsymbol{b}^{(\ell)}$ est une transformation affine avec $W^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$, et $\theta = \{W^{(\ell)}, \boldsymbol{b}^{(\ell)}\}_{\ell=1}^{L}$ désigne l'ensemble des paramètres.



1.2.2 Nombre de paramètres

Pour un MLP de couches $(n_0, n_1, \ldots, n_L)$, le nombre total de paramètres est :

$$|\theta| = \sum_{\ell=1}^{L} (n_{\ell-1} \cdot n_\ell + n_\ell) = \sum_{\ell=1}^{L} n_\ell (n_{\ell-1} + 1)$$

Exemple 1.2. Un MLP $(784, 256, 128, 10)$ pour MNIST a $784 \times 256 + 256 + 256 \times 128 + 128 + 128 \times 10 + 10 = 235\,146$ paramètres.

1.3 Fonctions de perte

1.3.1 Régression : erreur quadratique moyenne

Pour la régression, on minimise l'erreur quadratique moyenne (MSE) :

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

1.3.2 Classification : entropie croisée

Pour la classification à K classes avec sortie softmax :

$$\mathcal{L}_{\text{CE}}(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log \hat{y}_{ik}$$

où $y_{ik} \in \{0, 1\}$ est l'encodage one-hot et $\hat{y}_{ik} = \text{softmax}(z_{ik})$.

Lien avec le maximum de vraisemblance

Minimiser l'entropie croisée est équivalent à maximiser la log-vraisemblance sous un modèle catégorique :

$$\hat{\theta}_{\text{MLE}} = \arg\max_\theta \sum_{i=1}^{N} \log p(y_i \mid \boldsymbol{x}_i; \theta) = \arg\min_\theta \mathcal{L}_{\text{CE}}(\theta)$$

1.3.3 Classification binaire : entropie croisée binaire

Pour $K = 2$ avec sortie sigmoïde :

$$\mathcal{L}_{\text{BCE}}(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

1.4 Théorème d'approximation universelle

Théorème 1.3 (Cybenko, 1989 ; Hornik, 1991)**.** *Soit σ une fonction d'activation continue et non polynomiale. Pour toute fonction continue $f : [0,1]^d \to \mathbb{R}$ et tout $\varepsilon > 0$, il existe $n \in \mathbb{N}$, des poids w_{ij}, b_j, c_j tels que :*

$$\left| f(\boldsymbol{x}) - \sum_{j=1}^{n} c_j\, \sigma\left(\sum_{i=1}^{d} w_{ij} x_i + b_j\right)\right| < \varepsilon \quad \forall \boldsymbol{x} \in [0,1]^d$$

Intuition

Le théorème d'approximation universelle dit qu'un MLP à une seule couche cachée *suffisamment large* peut approximer toute fonction continue. Cependant, il ne dit rien sur :

- La **largeur** n nécessaire (qui peut être exponentiellement grande).
- La **facilité d'apprentissage** des poids optimaux.
- L'avantage de la **profondeur** (couches multiples) par rapport à la largeur.

En pratique, les réseaux profonds atteignent de meilleures performances avec beaucoup moins de paramètres que les réseaux larges et peu profonds.

1.5 Initialisation des poids

Une mauvaise initialisation peut provoquer l'explosion ou l'évanescence des activations dès la passe avant.

Initialisation de Xavier/Glorot (2010)

Pour préserver la variance des activations à travers les couches avec des activations symétriques (tanh, sigmoïde linéaire) :

$$W_{ij}^{(\ell)} \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{\ell-1} + n_\ell}}, \sqrt{\frac{6}{n_{\ell-1} + n_\ell}}\right)$$

Initialisation de He/Kaiming (2015)

Pour les activations ReLU (qui annulent la moitié des activations) :

$$W_{ij}^{(\ell)} \sim \mathcal{N}\left(0, \frac{2}{n_{\ell-1}}\right)$$

1.6 Implémentation PyTorch

MLP en PyTorch — Classification MNIST

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Hyperparamètres
BATCH_SIZE = 128
LR = 1e-3
EPOCHS = 10
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Données MNIST
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
train_data = datasets.MNIST('./data', train=True, download=True,
                            transform=transform)
test_data = datasets.MNIST('./data', train=False, transform=transform)
train_loader = DataLoader(train_data, batch_size=BATCH_SIZE,
↪  shuffle=True)
test_loader = DataLoader(test_data, batch_size=BATCH_SIZE)

# Modèle MLP
class MLP(nn.Module):
    def __init__(self, input_dim=784, hidden_dims=[256, 128],
                 num_classes=10):
        super().__init__()
        layers = []
        prev = input_dim
        for h in hidden_dims:
            layers += [nn.Linear(prev, h), nn.ReLU()]
            prev = h
        layers.append(nn.Linear(prev, num_classes))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
```

```
        return self.net(x.view(x.size(0), -1))  # Aplatir

model = MLP().to(DEVICE)
print(f"Paramètres : {sum(p.numel() for p in model.parameters()):,}")

# Entraînement
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=LR)

for epoch in range(1, EPOCHS + 1):
    model.train()
    total_loss = 0
    for X, y in train_loader:
        X, y = X.to(DEVICE), y.to(DEVICE)
        optimizer.zero_grad()
        loss = criterion(model(X), y)
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * X.size(0)

    # Évaluation
    model.eval()
    correct = 0
    with torch.no_grad():
        for X, y in test_loader:
            X, y = X.to(DEVICE), y.to(DEVICE)
            correct += (model(X).argmax(1) == y).sum().item()

    acc = correct / len(test_data)
    print(f"Epoch {epoch:2d} | Loss: {total_loss/len(train_data):.4f}"
          f" | Test Acc: {acc:.4f}")
```

Sortie

```
Paramètres : 235,146
Epoch  1 | Loss: 0.2534 | Test Acc: 0.9612
Epoch  2 | Loss: 0.1042 | Test Acc: 0.9710
...
Epoch 10 | Loss: 0.0198 | Test Acc: 0.9793
```

1.7 Étude d'ablation : influence de la largeur et de la profondeur

Étude d'ablation

```python
# Comparaison de différentes architectures MLP sur MNIST
configs = {
    "Shallow-wide":  [1024],
    "2-layers":      [256, 128],
    "3-layers":      [256, 128, 64],
    "Deep-narrow":   [64, 64, 64, 64],
}

results = {}
for name, hidden in configs.items():
    model = MLP(hidden_dims=hidden).to(DEVICE)
    n_params = sum(p.numel() for p in model.parameters())
    # ... entraîner et évaluer ...
    results[name] = {"params": n_params, "acc": test_acc}
    print(f"{name:20s} | {n_params:>8,} params | Acc: {test_acc:.4f}")
```

Sortie

```
Shallow-wide         |  812,042 params | Acc: 0.9781
2-layers             |  235,146 params | Acc: 0.9793
3-layers             |  243,338 params | Acc: 0.9802
Deep-narrow          |   63,434 params | Acc: 0.9745
```

Remarque 1.4. L'architecture à 3 couches atteint la meilleure précision avec un nombre raisonnable de paramètres. Le réseau large et peu profond utilise 3× plus de paramètres pour un résultat similaire. Le réseau profond et étroit, bien qu'économique, perd en performance — la largeur minimale par couche importe.

1.8 Connexion à l'état de l'art

État de l'art

- **MLP-Mixer** (Tolstikhin et al., 2021) : architecture entièrement MLP pour la vision, rivalisant avec les CNN et les ViT en utilisant uniquement des mélanges de tokens et de canaux.
- **KAN** (Kolmogorov-Arnold Networks, Liu et al., 2024) : remplacent les poids fixes par des fonctions apprenables sur les arêtes, améliorant l'interprétabilité et l'efficacité sur des problèmes scientifiques.
- **Scaling laws** (Kaplan et al., 2020) : la performance d'un réseau suit des lois de puissance en fonction du nombre de paramètres, de la taille des données et du budget de calcul.

1.9 Résumé du chapitre

Formules clés

- **Neurone formel** : $y = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b)$
- **MLP** : $f_\theta = \sigma^{(L)} \circ A^{(L)} \circ \cdots \circ \sigma^{(1)} \circ A^{(1)}$
- **Perte MSE** : $\frac{1}{N} \sum_i \|y_i - \hat{y}_i\|^2$
- **Perte CE** : $-\frac{1}{N} \sum_i \sum_k y_{ik} \log \hat{y}_{ik}$
- **Init Xavier** : $W \sim \mathcal{U}(-\sqrt{6/(n_{\text{in}} + n_{\text{out}})}, +\cdot)$
- **Init He** : $W \sim \mathcal{N}(0, 2/n_{\text{in}})$
- **UAT** : un MLP à 1 couche cachée peut approximer toute fonction continue

1.10 Exercices

Exercice 1.1 ($\star$ — Dérivation)**.** Montrer que la dérivée de la sigmoïde vérifie $\sigma'(z) = \sigma(z)(1 - \sigma(z))$. En déduire que $\max_z \sigma'(z) = 1/4$.

Exercice 1.2 ($\star$ — Comptage de paramètres)**.** Calculer le nombre exact de paramètres d'un MLP $(1024, 512, 256, 128, 10)$. Comparer avec un réseau $(1024, 2048, 10)$ de même budget paramétrique approximatif.

Exercice 1.3 ($\star\star$ — Implémentation)**.** Implémenter un MLP en PyTorch pour la classification Fashion-MNIST (10 classes de vêtements). Comparer les performances avec ReLU, Leaky ReLU et GELU. Tracer les courbes d'apprentissage.

Exercice 1.4 ($\star\star$ — Étude d'ablation)**.** Réaliser une étude d'ablation systématique sur MNIST en faisant varier : (a) le nombre de couches de 1 à 6, (b) la largeur de 32 à 512. Tracer une carte de chaleur (heatmap) précision vs (profondeur, largeur).

Exercice 1.5 ($\star\star\star$ — Projet de recherche)**.** Implémenter un MLP-Mixer pour CIFAR-10 en suivant l'article de Tolstikhin et al. (2021). Comparer ses performances avec un MLP classique et un CNN simple. Analyser l'influence de la taille des patchs et de la dimension cachée.

Chapitre 2

Rétropropagation et optimisation

Intuition

La rétropropagation est l'algorithme qui permet au réseau d'*apprendre*. Elle répond à une question simple : « Comment chaque poids influence-t-il l'erreur finale ? » En calculant efficacement tous les gradients par la règle de chaîne, elle transforme un problème en apparence insoluble (10^6 paramètres) en un calcul linéaire en le nombre de paramètres.

2.1 Descente de gradient

2.1.1 Principe

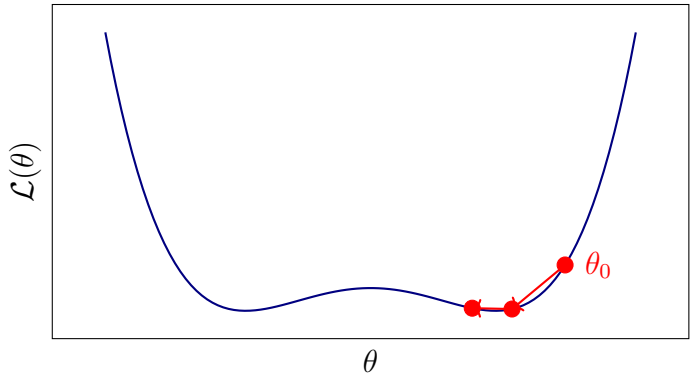
On cherche à minimiser la perte $\mathcal{L}(\theta)$ par rapport aux paramètres θ :

$$\theta^* = \arg\min_{\theta} \mathcal{L}(\theta)$$

La descente de gradient itère la mise à jour :

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

où $\eta > 0$ est le **taux d'apprentissage**.



2.1.2 Descente de gradient stochastique (SGD)

En pratique, calculer $\nabla_\theta \mathcal{L}$ sur tout le dataset est coûteux. La SGD utilise des **mini-batchs** $\mathcal{B} \subset \{1, \dots, N\}$ de taille B :

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_\theta \ell(f_\theta(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

Proposition 2.1. Le gradient stochastique est un estimateur non biaisé du gradient complet :

$$\mathbb{E}_{\mathcal{B}}\left[\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_\theta \ell_i\right] = \nabla_\theta \mathcal{L}(\theta)$$

Sa variance décroît en $\mathcal{O}(1/B)$.

2.2 Rétropropagation

2.2.1 Règle de chaîne

Soit $f = f_L \circ f_{L-1} \circ \cdots \circ f_1$ une composition de fonctions. La règle de chaîne donne :

$$\frac{\partial \mathcal{L}}{\partial \theta^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(L)}} \cdot \frac{\partial \boldsymbol{h}^{(L)}}{\partial \boldsymbol{h}^{(L-1)}} \cdots \frac{\partial \boldsymbol{h}^{(\ell+1)}}{\partial \boldsymbol{h}^{(\ell)}} \cdot \frac{\partial \boldsymbol{h}^{(\ell)}}{\partial \theta^{(\ell)}}$$

Dérivation de la rétropropagation pour un MLP

Passe avant. Pour $\ell = 1, \dots, L$:

$$\begin{aligned} \boldsymbol{z}^{(\ell)} &= W^{(\ell)} \boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)} \\ \boldsymbol{h}^{(\ell)} &= \sigma(\boldsymbol{z}^{(\ell)}) \end{aligned}$$

Passe arrière. On définit $\boldsymbol{\delta}^{(\ell)} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{(\ell)}}$. Pour la dernière couche :

$$\boldsymbol{\delta}^{(L)} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(L)}} \odot \sigma'(\boldsymbol{z}^{(L)})$$

Pour les couches $\ell = L-1, \dots, 1$ (propagation arrière) :

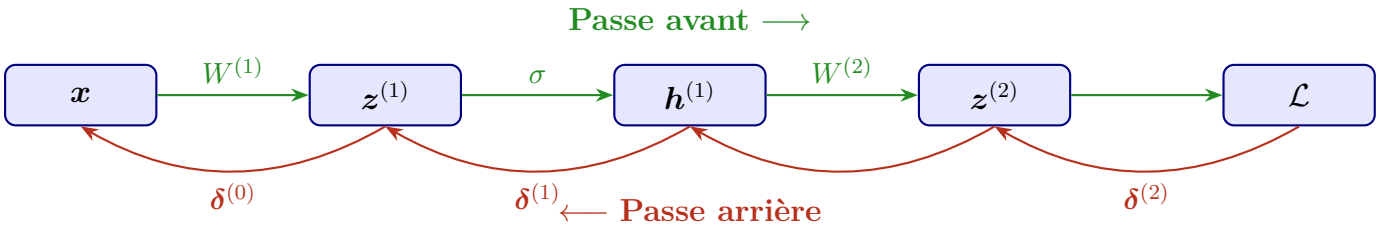
$$\boldsymbol{\delta}^{(\ell)} = \left(W^{(\ell+1)\top} \boldsymbol{\delta}^{(\ell+1)}\right) \odot \sigma'(\boldsymbol{z}^{(\ell)})$$

Les gradients des paramètres sont alors :

$$\frac{\partial \mathcal{L}}{\partial W^{(\ell)}} = \boldsymbol{\delta}^{(\ell)} \boldsymbol{h}^{(\ell-1)\top}, \qquad \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)}$$

2.2.2 Complexité

Proposition 2.2. La rétropropagation calcule **tous** les gradients en $\mathcal{O}(|\theta|)$, soit le même coût que la passe avant. C'est un cas particulier de la **différentiation automatique en mode inverse**.



2.3 Graphe de calcul et autograd

PyTorch construit dynamiquement un **graphe de calcul** (DAG) lors de la passe avant. Chaque opération enregistre comment calculer son gradient. L'appel à `loss.backward()` parcourt ce graphe en sens inverse.

Autograd en action

```python
import torch

# Tenseurs avec suivi de gradient
x = torch.tensor([2.0, 3.0], requires_grad=True)
w = torch.tensor([1.0, -1.0], requires_grad=True)

# Passe avant : graphe construit automatiquement
z = x @ w           # produit scalaire
y = z.sigmoid()     # activation
loss = (y - 1)**2   # perte

# Passe arrière : tous les gradients calculés
loss.backward()
print(f"dL/dx = {x.grad}")   # gradient par rapport à x
print(f"dL/dw = {w.grad}")   # gradient par rapport à w
```

Sortie

```
dL/dx = tensor([-0.0689,  0.0689])
dL/dw = tensor([-0.1378,  0.2067])
```

2.4 Optimiseurs avancés

2.4.1 Momentum

Le momentum accumule une moyenne mobile des gradients passés pour accélérer la convergence et lisser les oscillations :

$$\boldsymbol{v}_{t+1} = \mu \boldsymbol{v}_t + \nabla_\theta \mathcal{L}(\theta_t)$$
$$\theta_{t+1} = \theta_t - \eta \boldsymbol{v}_{t+1}$$

avec $\mu \in [0, 1)$ (typiquement $\mu = 0.9$).

2.4.2 RMSProp

RMSProp adapte le taux d'apprentissage par paramètre en normalisant par la moyenne mobile des carrés des gradients :

$$\boldsymbol{s}_{t+1} = \beta \boldsymbol{s}_t + (1-\beta)\,(\nabla_\theta \mathcal{L})^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\boldsymbol{s}_{t+1}} + \varepsilon} \odot \nabla_\theta \mathcal{L}$$

2.4.3 Adam

Adam[10] combine le momentum et l'adaptation du taux d'apprentissage :

Algorithme Adam

$$\boldsymbol{m}_{t+1} = \beta_1 \boldsymbol{m}_t + (1-\beta_1)\,\boldsymbol{g}_t \qquad \text{(premier moment)}$$
$$\boldsymbol{v}_{t+1} = \beta_2 \boldsymbol{v}_t + (1-\beta_2)\,\boldsymbol{g}_t^2 \qquad \text{(second moment)}$$
$$\hat{\boldsymbol{m}}_{t+1} = \frac{\boldsymbol{m}_{t+1}}{1-\beta_1^{t+1}} \qquad \text{(correction du biais)}$$
$$\hat{\boldsymbol{v}}_{t+1} = \frac{\boldsymbol{v}_{t+1}}{1-\beta_2^{t+1}}$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}_{t+1}} + \varepsilon}\,\hat{\boldsymbol{m}}_{t+1}$$

Valeurs par défaut : $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$.

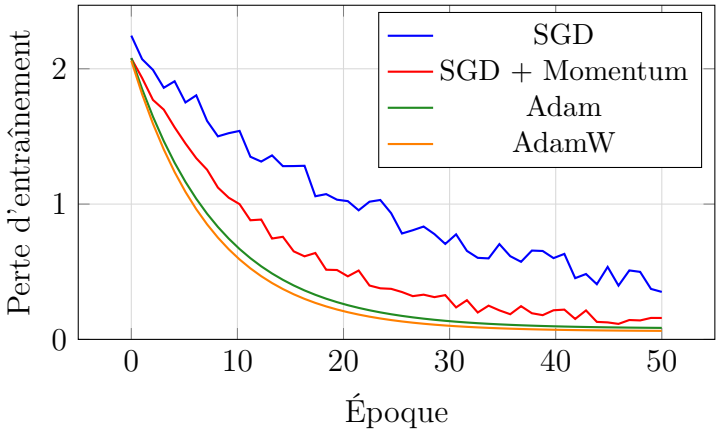
2.4.4 AdamW

AdamW découple la régularisation L_2 de l'adaptation du taux d'apprentissage :

$$\theta_{t+1} = (1-\eta\lambda)\,\theta_t - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}_{t+1}} + \varepsilon}\,\hat{\boldsymbol{m}}_{t+1}$$

C'est l'optimiseur standard pour l'entraînement des Transformers.

2.4.5 Comparaison visuelle



2.5 Planification du taux d'apprentissage

2.5.1 Décroissance par paliers (Step Decay)

$$\eta_t = \eta_0 \cdot \gamma^{\lfloor t/T \rfloor}$$

avec $\gamma = 0.1$ tous les $T = 30$ époques, par exemple.

2.5.2 Décroissance cosinus

$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})\left(1 + \cos\frac{\pi t}{T}\right)$$

2.5.3 Warmup linéaire + décroissance cosinus

Utilisé par défaut pour les Transformers : augmentation linéaire de η pendant T_w itérations, puis décroissance cosinus.

Planificateurs en PyTorch

```python
from torch.optim.lr_scheduler import (
    StepLR, CosineAnnealingLR, OneCycleLR
)

optimizer = optim.AdamW(model.parameters(), lr=1e-3, weight_decay=0.01)

# Décroissance par paliers
scheduler_step = StepLR(optimizer, step_size=30, gamma=0.1)

# Cosine annealing
scheduler_cos = CosineAnnealingLR(optimizer, T_max=100, eta_min=1e-6)

# One-cycle (warmup + cosine decay)
scheduler_1cycle = OneCycleLR(
    optimizer, max_lr=1e-3, total_steps=len(train_loader) * EPOCHS
)

# Dans la boucle d'entraînement :
for epoch in range(EPOCHS):
    for X, y in train_loader:
        optimizer.zero_grad()
        loss = criterion(model(X.to(DEVICE)), y.to(DEVICE))
        loss.backward()
        optimizer.step()
        scheduler_1cycle.step()  # par itération pour OneCycle
    scheduler_cos.step()         # par époque pour CosineAnnealing
```

2.6 Problèmes de gradients

2.6.1 Gradient évanescent

Dans un réseau à L couches avec des sigmoïdes, le gradient de la couche ℓ contient le produit :

$$\prod_{k=\ell+1}^{L} \sigma'(z^{(k)}) \cdot W^{(k)}$$

Comme $|\sigma'(z)| \leq 1/4$, ce produit décroît exponentiellement avec $L - \ell$, empêchant l'apprentissage des premières couches.

2.6.2 Gradient explosif

Si $\|W^{(k)}\| > 1/|\sigma'_{\max}|$, le produit croît exponentiellement. Solution : le **gradient clipping** :

$$\boldsymbol{g} \leftarrow \begin{cases} \boldsymbol{g} & \text{si } \|\boldsymbol{g}\| \leq c \\ c \cdot \frac{\boldsymbol{g}}{\|\boldsymbol{g}\|} & \text{sinon} \end{cases}$$

Gradient clipping en PyTorch

```
# Après loss.backward(), avant optimizer.step()
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
```

2.6.3 Solutions modernes

1. **ReLU** : $\sigma'(z) = 1$ pour $z > 0$, pas d'atténuation.
2. **Connexions résiduelles** (chapitre 4) : $\boldsymbol{h}^{(\ell)} = F(\boldsymbol{h}^{(\ell-1)}) + \boldsymbol{h}^{(\ell-1)}$.
3. **Normalisation** : BatchNorm, LayerNorm (chapitre 3).
4. **Initialisation adaptée** : Xavier, He (section 1.5).

2.7 Implémentation complète : rétropropagation manuelle

Rétropropagation manuelle (sans autograd)

```
import numpy as np

class ManualMLP:
    """MLP 2 couches avec backprop manuelle."""
    def __init__(self, d_in, d_hid, d_out):
        # Initialisation He
        self.W1 = np.random.randn(d_hid, d_in) * np.sqrt(2/d_in)
        self.b1 = np.zeros(d_hid)
        self.W2 = np.random.randn(d_out, d_hid) * np.sqrt(2/d_hid)
```

```
        self.b2 = np.zeros(d_out)

    def relu(self, z):
        return np.maximum(0, z)

    def softmax(self, z):
        e = np.exp(z - z.max(axis=1, keepdims=True))
        return e / e.sum(axis=1, keepdims=True)

    def forward(self, X):
        self.z1 = X @ self.W1.T + self.b1          # (N, d_hid)
        self.h1 = self.relu(self.z1)                # (N, d_hid)
        self.z2 = self.h1 @ self.W2.T + self.b2     # (N, d_out)
        self.probs = self.softmax(self.z2)          # (N, d_out)
        return self.probs

    def backward(self, X, y_onehot, lr=0.01):
        N = X.shape[0]
        # δ2 = ∂L/∂z2 = probs - y (pour CE + softmax)
        delta2 = (self.probs - y_onehot) / N        # (N, d_out)
        dW2 = delta2.T @ self.h1                    # (d_out, d_hid)
        db2 = delta2.sum(axis=0)                    # (d_out,)

        # δ1 = (δ2 @ W2) ⊙ ReLU'(z1)
        delta1 = (delta2 @ self.W2) * (self.z1 > 0)  # (N, d_hid)
        dW1 = delta1.T @ X                           # (d_hid, d_in)
        db1 = delta1.sum(axis=0)                     # (d_hid,)

        # Mise à jour
        self.W2 -= lr * dW2
        self.b2 -= lr * db2
        self.W1 -= lr * dW1
        self.b1 -= lr * db1

    def loss(self, y_onehot):
        return -np.mean(np.sum(y_onehot * np.log(self.probs + 1e-8),
                               axis=1))

# Test sur données synthétiques
np.random.seed(42)
X = np.random.randn(1000, 20)
y_true = (X[:, 0] + X[:, 1] > 0).astype(int)
y_oh = np.eye(2)[y_true]

mlp = ManualMLP(20, 64, 2)
for epoch in range(100):
    mlp.forward(X)
    if epoch % 20 == 0:
        acc = (mlp.probs.argmax(1) == y_true).mean()
        print(f"Epoch {epoch:3d} | Loss: {mlp.loss(y_oh):.4f}"
              f" | Acc: {acc:.4f}")
```

```
    mlp.backward(X, y_oh, lr=0.1)
```

Sortie

```
Epoch   0 | Loss: 0.7234 | Acc: 0.4930
Epoch  20 | Loss: 0.2156 | Acc: 0.9240
Epoch  40 | Loss: 0.1052 | Acc: 0.9670
Epoch  60 | Loss: 0.0614 | Acc: 0.9810
Epoch  80 | Loss: 0.0402 | Acc: 0.9880
```

2.8 Étude d'ablation : optimiseurs

Comparaison d'optimiseurs sur MNIST

```
optimizers = {
    "SGD":           optim.SGD(model.parameters(), lr=0.01),
    "SGD+Momentum": optim.SGD(model.parameters(), lr=0.01, momentum=0.9),
    "Adam":          optim.Adam(model.parameters(), lr=1e-3),
    "AdamW":         optim.AdamW(model.parameters(), lr=1e-3,
                                 weight_decay=0.01),
}

for name, opt in optimizers.items():
    model_copy = MLP().to(DEVICE)
    opt = type(opt)(model_copy.parameters(), **opt.defaults)
    # Entraîner 10 époques, enregistrer loss et accuracy
    history = train_and_eval(model_copy, opt, train_loader,
                             test_loader, epochs=10)
    print(f"{name:15s} | Final Acc: {history['test_acc'][-1]:.4f}")
```

Sortie

```
SGD             | Final Acc: 0.9532
SGD+Momentum    | Final Acc: 0.9721
Adam            | Final Acc: 0.9793
AdamW           | Final Acc: 0.9801
```

2.9 Connexion à l'état de l'art

État de l'art

- **LAMB/LARS** (You et al., 2020) : optimiseurs adaptés pour l'entraînement distribué avec de très grands batchs ($B > 8192$).
- **Lion** (Chen et al., 2024) : optimiseur découvert par recherche évolutionnaire, utilisant uniquement le signe du gradient, plus efficace en mémoire qu'Adam.

- **Muon** (Jordan et al., 2024) : optimiseur basé sur l'orthogonalisation du momentum, améliorant la convergence pour les grands modèles de langage.
- **μP** (Yang & Hu, 2022) : paramétrisation maximale permettant de transférer les hyperparamètres d'un petit modèle vers un grand modèle.

2.10 Résumé du chapitre

Formules clés

- **Descente de gradient** : $\theta_{t+1} = \theta_t - \eta \nabla \mathcal{L}$
- **SGD mini-batch** : gradient sur un sous-ensemble $\mathcal{B}$
- **Rétropropagation** : $\boldsymbol{\delta}^{(\ell)} = (W^{(\ell+1)\top} \boldsymbol{\delta}^{(\ell+1)}) \odot \sigma'(\boldsymbol{z}^{(\ell)})$
- **Gradient des poids** : $\partial \mathcal{L} / \partial W^{(\ell)} = \boldsymbol{\delta}^{(\ell)} \boldsymbol{h}^{(\ell-1)\top}$
- **Adam** : combine momentum ($\boldsymbol{m}$) et adaptation ($\boldsymbol{v}$) avec correction du biais
- **Cosine annealing** : $\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})(1 + \cos \frac{\pi t}{T})$
- **Gradient clipping** : $\boldsymbol{g} \leftarrow c \cdot \boldsymbol{g} / \|\boldsymbol{g}\|$ si $\|\boldsymbol{g}\| > c$

2.11 Exercices

Exercice 2.1 ($\star$ — Dérivation)**.** Dériver les équations de rétropropagation pour un MLP à 3 couches avec activation tanh, en explicitant toutes les dimensions des matrices Jacobiennes.

Exercice 2.2 ($\star$ — Analyse)**.** Montrer que l'estimateur SGD mini-batch a une variance $\mathrm{Var}[\hat{g}] = \sigma^2 / B$ où σ^2 est la variance du gradient individuel et B la taille du batch. En déduire le compromis entre taille de batch et bruit du gradient.

Exercice 2.3 ($\star\star$ — Implémentation)**.** Implémenter la rétropropagation *sans autograd* pour un MLP à 3 couches avec ReLU. Vérifier vos gradients par différences finies : $\frac{\partial \mathcal{L}}{\partial w} \approx \frac{\mathcal{L}(w+\varepsilon) - \mathcal{L}(w-\varepsilon)}{2\varepsilon}$.

Exercice 2.4 ($\star\star$ — Comparaison)**.** Comparer SGD, SGD+Momentum, Adam et AdamW sur Fashion-MNIST avec un MLP $(784, 512, 256, 10)$. Pour chaque optimiseur, effectuer une recherche sur grille du taux d'apprentissage dans $\{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$. Tracer les courbes de perte et de précision.

Exercice 2.5 ($\star\star\star$ — Projet de recherche)**.** Reproduire l'expérience de transfert d'hyperparamètres de μP (Yang & Hu, 2022) : entraîner un petit MLP, transférer le taux d'apprentissage optimal vers un modèle $10\times$ plus grand. Comparer avec un tuning direct sur le grand modèle.

Chapitre 3

Régularisation et Techniques de Généralisation

Ce chapitre présente les techniques fondamentales permettant d'améliorer la capacité de généralisation des réseaux de neurones profonds. Nous dérivons mathématiquement les méthodes de régularisation classiques (L1, L2, Dropout, Batch Normalization) et explorons les approches modernes utilisées dans les architectures état de l'art.

3.1 Surapprentissage et sous-apprentissage

3.1.1 Décomposition biais-variance

Considérons un modèle $\hat{f}$ entraîné sur un ensemble $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$ où $y = f(x) + \epsilon$ avec $\epsilon \sim \mathcal{N}(0, \sigma^2)$.

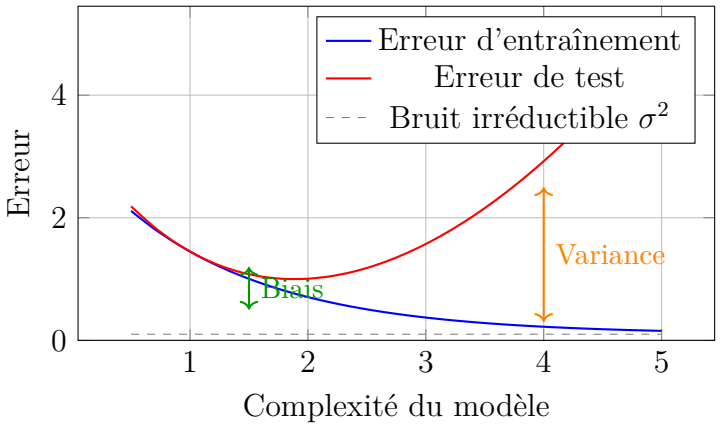
Décomposition biais-variance

L'erreur quadratique moyenne en un point x se décompose comme :

$$\begin{aligned}\mathbb{E}\big[(y - \hat{f}(x))^2\big] &= \mathbb{E}\big[(f(x) + \epsilon - \hat{f}(x))^2\big] \\ &= \big(f(x) - \mathbb{E}[\hat{f}(x)]\big)^2 + \mathbb{E}\big[(\hat{f}(x) - \mathbb{E}[\hat{f}(x)])^2\big] + \sigma^2 \\ &= \underbrace{\mathrm{Biais}^2(\hat{f}(x))}_{\text{erreur systématique}} + \underbrace{\mathrm{Var}(\hat{f}(x))}_{\text{sensibilité aux données}} + \underbrace{\sigma^2}_{\text{bruit irréductible}} \end{aligned} \tag{3.1}$$

Définition 3.1 (Surapprentissage (Overfitting))**.** Un modèle est en **surapprentissage** lorsque l'erreur d'entraînement est significativement inférieure à l'erreur de test : $\mathcal{L}_{\text{train}} \ll \mathcal{L}_{\text{test}}$. Cela correspond à une variance élevée dans la décomposition biais-variance.

Définition 3.2 (Sous-apprentissage (Underfitting))**.** Un modèle est en **sous-apprentissage** lorsque l'erreur d'entraînement est elle-même élevée : $\mathcal{L}_{\text{train}} \approx \mathcal{L}_{\text{test}} \gg \mathcal{L}^*$. Cela correspond à un biais élevé.



3.2 Régularisation L2 (Ridge)

3.2.1 Formulation

La régularisation L2 ajoute une pénalité quadratique sur les poids :

Fonction de coût régularisée L2

$$\tilde{\mathcal{L}}(\theta) = \mathcal{L}(\theta) + \frac{\lambda}{2}\|\theta\|_2^2 = \mathcal{L}(\theta) + \frac{\lambda}{2}\sum_i \theta_i^2 \tag{3.2}$$

3.2.2 Dérivation du gradient

Le gradient de la fonction régularisée est :

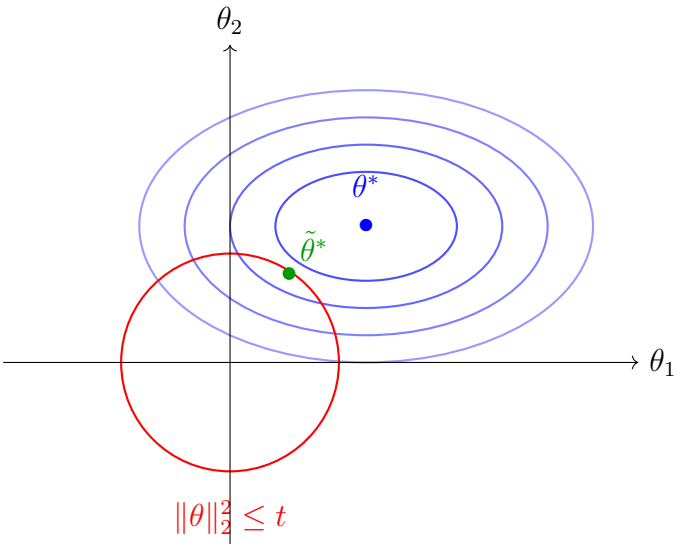
$$\nabla_\theta \tilde{\mathcal{L}}(\theta) = \nabla_\theta \mathcal{L}(\theta) + \lambda\theta \tag{3.3}$$

La règle de mise à jour avec descente de gradient devient :

$$\begin{aligned} \theta_{t+1} &= \theta_t - \eta\nabla_\theta \tilde{\mathcal{L}}(\theta_t) \\ &= \theta_t - \eta\nabla_\theta \mathcal{L}(\theta_t) - \eta\lambda\theta_t \\ &= (1-\eta\lambda)\theta_t - \eta\nabla_\theta \mathcal{L}(\theta_t) \end{aligned} \tag{3.4}$$

Le facteur $(1-\eta\lambda)$ réduit les poids à chaque itération, d'où le nom « weight decay ».

3.2.3 Interprétation géométrique



Effet géométrique de L2

La régularisation L2 contraint les poids à rester dans une boule euclidienne. La solution régularisée $\tilde{\theta}^*$ se trouve à l'intersection des courbes de niveau de $\mathcal{L}$ et de la sphère $\|\theta\|_2 = r$.

3.3 Régularisation L1 (Lasso)

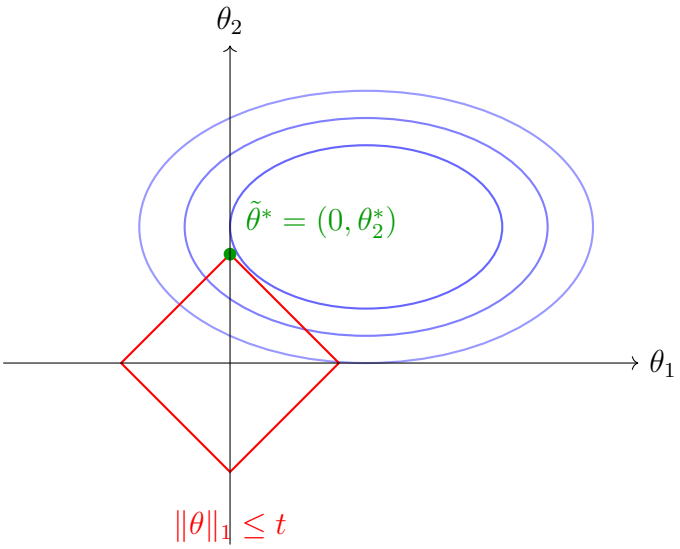
Fonction de coût régularisée L1

$$\tilde{\mathcal{L}}(\theta) = \mathcal{L}(\theta) + \lambda\|\theta\|_1 = \mathcal{L}(\theta) + \lambda \sum_i |\theta_i| \tag{3.5}$$

Le sous-gradient de la pénalité L1 est :

$$\frac{\partial}{\partial \theta_i} \lambda|\theta_i| = \lambda \cdot \text{sign}(\theta_i) = \begin{cases} +\lambda & \text{si } \theta_i > 0 \\ -\lambda & \text{si } \theta_i < 0 \\ [-\lambda, +\lambda] & \text{si } \theta_i = 0 \end{cases} \tag{3.6}$$

Remarque 3.3 (Parcimonie de L1). La régularisation L1 produit des solutions **parcimonieuses** (sparse) : certains poids sont exactement zéro. Géométriquement, la contrainte $\|\theta\|_1 \leq t$ forme un losange dont les sommets sont alignés avec les axes, favorisant les solutions sur les axes de coordonnées.



3.4 Dropout

3.4.1 Masquage de Bernoulli

Le Dropout [16] désactive aléatoirement une fraction p des neurones pendant l'entraînement. Pour chaque neurone j de la couche l, on échantillonne un masque :

$$m_j^{(l)} \sim \text{Bernoulli}(1-p) \tag{3.7}$$

La sortie masquée est :

$$\tilde{h}_j^{(l)} = m_j^{(l)} \cdot h_j^{(l)} \tag{3.8}$$

3.4.2 Espérance et Dropout inversé

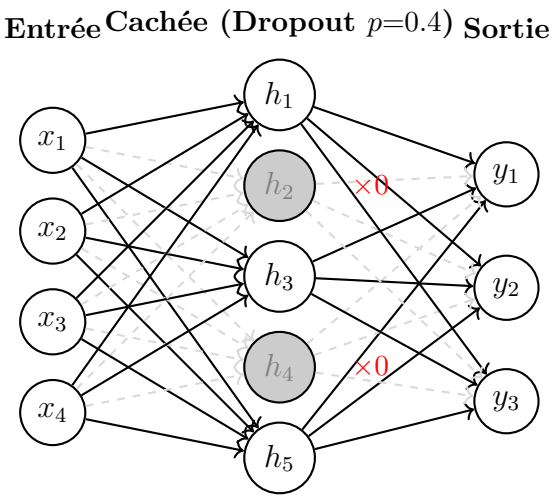
Théorème 3.4 (Espérance du Dropout)**.** *L'espérance de la sortie masquée est :*

$$\mathbb{E}[\tilde{h}_j^{(l)}] = \mathbb{E}[m_j^{(l)}] \cdot h_j^{(l)} = (1-p) \cdot h_j^{(l)} \tag{3.9}$$

Pour maintenir la même espérance entre entraînement et inférence, le ***Dropout inversé*** *divise par* $(1-p)$ *pendant l'entraînement :*

$$\tilde{h}_j^{(l)} = \frac{m_j^{(l)}}{1-p} \cdot h_j^{(l)} \tag{3.10}$$

Ainsi $\mathbb{E}[\tilde{h}_j^{(l)}] = h_j^{(l)}$ *et aucune modification n'est nécessaire à l'inférence.*



3.5 Batch Normalization

3.5.1 Problème du décalage interne des covariables

L'entraînement des réseaux profonds est compliqué par le fait que la distribution des entrées de chaque couche change à mesure que les poids des couches précédentes évoluent. Ce phénomène, appelé *Internal Covariate Shift* [17], ralentit la convergence.

3.5.2 Passe avant (Forward)

Pour un mini-batch $\mathcal{B} = \{x_1, \ldots, x_m\}$:

Batch Normalization – Passe avant

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{3.11}$$

$$\sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \tag{3.12}$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \tag{3.13}$$

$$y_i = \gamma \hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \tag{3.14}$$

où γ et β sont des paramètres apprenables, et $\epsilon > 0$ est un terme de stabilité numérique.

3.5.3 Passe arrière (Backward)

Proposition 3.5 (Gradients de la Batch Normalization)**.** En notant $\frac{\partial \mathcal{L}}{\partial y_i} = \delta_i$, les gradients de la BatchNorm sont :

$$\frac{\partial \mathcal{L}}{\partial \gamma} = \sum_{i=1}^{m} \delta_i \hat{x}_i \tag{3.15}$$

$$\frac{\partial \mathcal{L}}{\partial \beta} = \sum_{i=1}^{m} \delta_i \tag{3.16}$$

$$\frac{\partial \mathcal{L}}{\partial \hat{x}_i} = \delta_i \cdot \gamma \tag{3.17}$$

$$\frac{\partial \mathcal{L}}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial \hat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \left(-\frac{1}{2}\right) (\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2} \tag{3.18}$$

$$\frac{\partial \mathcal{L}}{\partial \mu_{\mathcal{B}}} = \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{3.19}$$

$$\frac{\partial \mathcal{L}}{\partial x_i} = \frac{\partial \mathcal{L}}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial \mathcal{L}}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial \mathcal{L}}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m} \tag{3.20}$$

3.6 Layer Normalization

Contrairement à la Batch Normalization qui normalise sur la dimension du batch, la Layer Normalization [18] normalise sur la dimension des caractéristiques. Pour une entrée $\mathbf{x} \in \mathbb{R}^d$:

Layer Normalization

$$\mu = \frac{1}{d} \sum_{j=1}^{d} x_j, \qquad \sigma^2 = \frac{1}{d} \sum_{j=1}^{d} (x_j - \mu)^2 \tag{3.21}$$

$$\mathrm{LN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{3.22}$$

Remarque 3.6 (BatchNorm vs LayerNorm).

- **BatchNorm** : normalise sur le batch, chaque canal indépendamment. Utilisé principalement dans les CNN (cf. chapitre 4).
- **LayerNorm** : normalise sur les caractéristiques, chaque échantillon indépendamment. Utilisé dans les Transformers (cf. chapitre 6).
- LayerNorm ne dépend pas de la taille du batch et fonctionne identiquement en entraînement et en inférence.

3.7 Augmentation des données

3.7.1 Transformations classiques

Pour les images : rotations, symétries horizontales, recadrages aléatoires, variations de luminosité, ajout de bruit.

3.7.2 Mixup

Mixup

Étant donnés deux exemples (x_i, y_i) et (x_j, y_j), Mixup [19] crée un nouvel exemple :

$$\tilde{x} = \lambda x_i + (1 - \lambda) x_j \tag{3.23}$$
$$\tilde{y} = \lambda y_i + (1 - \lambda) y_j \tag{3.24}$$

où $\lambda \sim \text{Beta}(\alpha, \alpha)$ avec $\alpha > 0$.

3.7.3 CutMix

CutMix

CutMix [20] remplace une région rectangulaire de x_i par la région correspondante de x_j :

$$\tilde{x} = \mathbf{M} \odot x_i + (1 - \mathbf{M}) \odot x_j \tag{3.25}$$
$$\tilde{y} = \lambda y_i + (1 - \lambda) y_j \tag{3.26}$$

où $\mathbf{M} \in \{0, 1\}^{H \times W}$ est un masque binaire et $\lambda = 1 - \frac{r_w \cdot r_h}{W \cdot H}$ représente la proportion de l'image originale.

3.8 Arrêt précoce (Early Stopping)

Définition 3.7 (Arrêt précoce)**.** L'arrêt précoce consiste à stopper l'entraînement lorsque l'erreur de validation cesse de diminuer pendant un nombre prédéfini d'époques (« patience »).

Théorème 3.8 (Équivalence avec la régularisation L2)**.** *Pour un modèle linéaire quadratique avec descente de gradient, l'arrêt à l'itération T est approximativement équivalent à une régularisation L2 avec $\lambda \approx \frac{1}{\eta T}$, où η est le taux d'apprentissage.*

Démonstration *: Soit $\mathcal{L}(\theta) = \frac{1}{2}(\theta - \theta^*)^\top H (\theta - \theta^*)$ avec H la Hessienne. En partant de $\theta_0 = 0$, après T itérations :*

$$\theta_T = (I - (I - \eta H)^T)\theta^* \tag{3.27}$$

Pour la régularisation L2, la solution est $\tilde{\theta} = (H + \lambda I)^{-1} H \theta^*$. En utilisant la décomposition spectrale de $H = Q \Lambda Q^\top$, on montre que les deux solutions coïncident quand $1 - (1 - \eta\lambda_i)^T \approx \frac{\lambda_i}{\lambda_i + 1/(\eta T)}$.

3.9 Weight Decay vs L2 dans Adam

Différence cruciale

Weight decay et régularisation L2 ne sont **pas équivalents** pour les optimiseurs adaptatifs comme Adam !

3.9.1 Régularisation L2 dans Adam

Avec L2 dans Adam, le gradient modifié est $g_t = \nabla \mathcal{L}(\theta_t) + \lambda \theta_t$. Ce gradient est ensuite normalisé par les moments d'Adam :

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)(g_t + \lambda \theta_t) \tag{3.28}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(g_t + \lambda \theta_t)^2 \tag{3.29}$$

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{3.30}$$

Le terme de régularisation $\lambda \theta_t$ est **atténué** par la normalisation adaptative $\frac{1}{\sqrt{\hat{v}_t}+\epsilon}$.

3.9.2 AdamW : Weight Decay découplé

AdamW [21] applique le weight decay directement :

AdamW

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{3.31}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{3.32}$$

$$\theta_{t+1} = (1 - \eta\lambda)\theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{3.33}$$

Utiliser AdamW

En pratique, **AdamW** (weight decay découplé) doit être préféré à Adam avec régularisation L2. AdamW est l'optimiseur standard pour les Transformers modernes.

3.10 Implémentation PyTorch

Boucle d'entraînement régularisée complète

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# --- Mod`ele avec Dropout et BatchNorm ---
class RegularizedNet(nn.Module):
```

```python
    def __init__(self, input_dim=784, hidden_dim=256,
                 num_classes=10, dropout_rate=0.5):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.BatchNorm1d(hidden_dim),
            nn.ReLU(),
            nn.Dropout(p=dropout_rate),
            nn.Linear(hidden_dim, hidden_dim),
            nn.BatchNorm1d(hidden_dim),
            nn.ReLU(),
            nn.Dropout(p=dropout_rate),
            nn.Linear(hidden_dim, num_classes)
        )

    def forward(self, x):
        return self.layers(x.view(x.size(0), -1))

# --- Configuration ---
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = RegularizedNet(dropout_rate=0.3).to(device)

# AdamW avec weight decay d'ecoupl'e
optimizer = optim.AdamW(model.parameters(), lr=1e-3,
                        weight_decay=1e-4)
criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=50)

# --- Data augmentation ---
transform_train = transforms.Compose([
    transforms.RandomRotation(10),
    transforms.RandomAffine(0, translate=(0.1, 0.1)),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

train_dataset = datasets.MNIST(root='./data', train=True,
                               transform=transform_train, download=True)
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
val_loader = DataLoader(
    datasets.MNIST(root='./data', train=False,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize((0.1307,), (0.3081,))
                   ])),
    batch_size=256, shuffle=False
)

# --- Early Stopping ---
class EarlyStopping:
    def __init__(self, patience=10, min_delta=1e-4):
```

```
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = float('inf')
        self.should_stop = False

    def __call__(self, val_loss):
        if val_loss < self.best_loss - self.min_delta:
            self.best_loss = val_loss
            self.counter = 0
        else:
            self.counter += 1
            if self.counter >= self.patience:
                self.should_stop = True

early_stopping = EarlyStopping(patience=10)

# --- Boucle d'entra^inement ---
for epoch in range(50):
    model.train()
    train_loss = 0.0
    for inputs, targets in train_loader:
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        train_loss += loss.item()

    # Validation
    model.eval()
    val_loss = 0.0
    correct = 0
    with torch.no_grad():
        for inputs, targets in val_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            val_loss += criterion(outputs, targets).item()
            correct += (outputs.argmax(1) == targets).sum().item()

    val_loss /= len(val_loader)
    accuracy = correct / len(val_loader.dataset)
    scheduler.step()

    print(f"Epoch {epoch+1}:
    ↪  train_loss={train_loss/len(train_loader):.4f}, "
          f"val_loss={val_loss:.4f}, accuracy={accuracy:.4f}")

    early_stopping(val_loss)
```

```
    if early_stopping.should_stop:
        print("Arr^et pr'ecoce d'eclench'e.")
        break
```

3.11 Étude d'ablation

Étude d'ablation : taux de Dropout et Weight Decay

```
import itertools
import pandas as pd

results = []
dropout_rates = [0.0, 0.1, 0.3, 0.5, 0.7]
weight_decays = [0.0, 1e-5, 1e-4, 1e-3, 1e-2]

for dr, wd in itertools.product(dropout_rates, weight_decays):
    model = RegularizedNet(dropout_rate=dr).to(device)
    optimizer = optim.AdamW(model.parameters(), lr=1e-3,
                            weight_decay=wd)
    # Entra^iner pendant 20 'epoques (simplifi'e)
    for epoch in range(20):
        model.train()
        for inputs, targets in train_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            optimizer.zero_grad()
            loss = criterion(model(inputs), targets)
            loss.backward()
            optimizer.step()

    # 'Evaluer
    model.eval()
    correct = 0
    with torch.no_grad():
        for inputs, targets in val_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            correct += (model(inputs).argmax(1) == targets).sum().item()
    acc = correct / len(val_loader.dataset)
    results.append({'dropout': dr, 'weight_decay': wd, 'accuracy': acc})

df = pd.DataFrame(results)
pivot = df.pivot(index='dropout', columns='weight_decay',
 ↪  values='accuracy')
print(pivot.to_string(float_format='{:.4f}'.format))
```

Sortie

```
weight_decay    0.0     1e-05   0.0001  0.001   0.01
dropout
0.0           0.9812  0.9821  0.9835  0.9801  0.9754
0.1           0.9838  0.9842  0.9851  0.9828  0.9773
0.3           0.9845  0.9849  0.9856  0.9839  0.9790
0.5           0.9831  0.9835  0.9843  0.9821  0.9779
0.7           0.9762  0.9770  0.9778  0.9755  0.9701
```

Remarque 3.9 (Observations).

- Le Dropout à $p = 0.3$ avec weight decay $\lambda = 10^{-4}$ donne les meilleurs résultats sur MNIST.
- Un Dropout trop élevé ($p = 0.7$) dégrade les performances en limitant trop la capacité du réseau.
- Le weight decay à 10^{-2} est trop agressif et nuit aux performances pour tous les taux de Dropout.

3.12 Techniques état de l'art

Techniques de régularisation modernes

Stochastic Depth [22] : désactive aléatoirement des blocs résiduels entiers (voir chapitre 4). Chaque bloc l a une probabilité de survie $p_l = 1 - \frac{l}{L}(1 - p_L)$.

DropPath : variante de Stochastic Depth appliquée par chemin dans les architectures multi-branches. Utilisé dans EfficientNet et les Vision Transformers.

Label Smoothing [23] : remplace les cibles one-hot par :

$$y_k^{\text{LS}} = (1 - \alpha)y_k + \frac{\alpha}{K} \tag{3.34}$$

où K est le nombre de classes et $\alpha \in [0, 1]$ le paramètre de lissage (typiquement $\alpha = 0.1$).

R-Drop [24] : minimise la divergence KL entre deux passes forward avec Dropout différent :

$$\mathcal{L}_{\text{R-Drop}} = \mathcal{L}_{\text{CE}} + \alpha \cdot \frac{1}{2}\big(\text{KL}(p_1 \| p_2) + \text{KL}(p_2 \| p_1)\big) \tag{3.35}$$

3.13 Exercices

Exercice 3.1 (Décomposition biais-variance ★)**.** Soit $f(x) = \sin(\pi x)$ et $\hat{f}$ un polynôme de degré d ajusté sur $N = 10$ points bruités ($\sigma = 0.3$).

1. Démontrer la décomposition biais-variance (équation 3.1).

2. Simuler en Python l'erreur pour $d \in \{1, 3, 5, 9, 15\}$ sur 100 jeux de données. Tracer biais^2, variance et erreur totale.

3. Quel degré offre le meilleur compromis ?

Exercice 3.2 (Régularisation Elastic Net ★★). L'Elastic Net combine L1 et L2 : $\Omega(\theta) = \alpha\|\theta\|_1 + \frac{1-\alpha}{2}\|\theta\|_2^2$.

1. Dériver la règle de mise à jour avec descente de gradient proximale pour la partie L1.

2. Implémenter cette régularisation en PyTorch en modifiant manuellement les gradients.

3. Comparer les solutions avec L1 seul, L2 seul, et Elastic Net sur un problème de régression linéaire avec $p = 100$ variables dont 10 pertinentes.

Exercice 3.3 (Dropout comme ensemble de modèles ★★). 1. Montrer qu'un réseau avec Dropout de taux p sur n unités implémente implicitement un ensemble de 2^n sous-réseaux.

2. Pour un réseau à une couche cachée (d unités) avec Dropout, dériver l'approximation de la moyenne géométrique du poids $W_{\text{test}} = (1 - p)W_{\text{train}}$.

3. Implémenter le MC-Dropout (Monte Carlo Dropout) pour estimer l'incertitude prédictive et comparer avec le Dropout standard sur un problème de régression.

Exercice 3.4 (Implémentation de BatchNorm à partir de zéro ★★★). 1. Implémenter la Batch Normalization complète (forward et backward) sans utiliser `nn.BatchNorm1d`.

2. Vérifier la correction du backward avec `torch.autograd.gradcheck`.

3. Comparer la vitesse de convergence avec et sans BatchNorm sur CIFAR-10.

4. Étudier l'effet de la taille du batch sur la stabilité de BatchNorm (batch sizes : 4, 16, 64, 256).

Exercice 3.5 (AdamW vs Adam + L2 ★★★). 1. Démontrer formellement que pour SGD, weight decay et L2 sont équivalents avec $\lambda_{\text{wd}} = \frac{\lambda_{L2}}{\eta}$.

2. Montrer que cette équivalence ne tient plus pour Adam en exhibant un contre-exemple analytique (modèle à 2 paramètres avec des magnitudes de gradient très différentes).

3. Entraîner un modèle Transformer (petit) avec Adam+L2 et AdamW, en variant le coefficient de régularisation. Tracer les courbes de généralisation et comparer.

Chapitre 4

Réseaux de Neurones Convolutifs (CNN)

Ce chapitre présente les réseaux de neurones convolutifs, une famille d'architectures spécialisées pour le traitement de données structurées spatialement. Nous dérivons les opérations fondamentales, étudions les architectures historiques et modernes, et démontrons mathématiquement les propriétés qui font le succès des connexions résiduelles.

4.1 Motivations

Les réseaux entièrement connectés présentent trois limitations majeures pour le traitement des images :

Définition 4.1 (Principes inductifs des CNN)**.** Les CNN exploitent trois principes :

1. **Localité** : les caractéristiques visuelles sont définies par des motifs locaux (arêtes, textures).

2. **Invariance par translation** : un motif visuel peut apparaître n'importe où dans l'image.

3. **Partage des paramètres** : le même filtre est appliqué à toutes les positions spatiales, réduisant drastiquement le nombre de paramètres.

Remarque 4.2 (Réduction des paramètres)*.* Pour une image $224 \times 224 \times 3$, une couche entièrement connectée vers 1000 neurones nécessite $224 \times 224 \times 3 \times 1000 \approx 150\text{M}$ paramètres. Une couche convolutive avec 64 filtres 3×3 n'en requiert que $3 \times 3 \times 3 \times 64 = 1\,728$ paramètres.

4.2 Opération de convolution

4.2.1 Convolution discrète 2D

Pour une entrée $\mathbf{X} \in \mathbb{R}^{H \times W}$ et un noyau $\mathbf{K} \in \mathbb{R}^{k_H \times k_W}$, la convolution discrète (plus précisément la corrélation croisée) est :

Convolution 2D

$$(\mathbf{X} * \mathbf{K})[i,j] = \sum_{m=0}^{k_H-1} \sum_{n=0}^{k_W-1} \mathbf{X}[i+m, j+n] \cdot \mathbf{K}[m,n] \tag{4.1}$$

4.2.2 Taille de sortie

Théorème 4.3 (Formule de la taille de sortie)**.** *Pour une entrée de taille $H \times W$, un noyau $k \times k$, un padding p et un stride s, la taille de sortie est :*

$$H_{out} = \left\lfloor \frac{H - k + 2p}{s} \right\rfloor + 1, \qquad W_{out} = \left\lfloor \frac{W - k + 2p}{s} \right\rfloor + 1 \tag{4.2}$$

Exemple 4.4 (Calcul de taille)**.** Entrée 32×32, noyau 5×5, padding $p = 2$, stride $s = 1$:

$$H_{\text{out}} = \left\lfloor \frac{32 - 5 + 4}{1} \right\rfloor + 1 = 32$$

Le padding $p = \lfloor k/2 \rfloor$ préserve la taille spatiale (« same padding »).

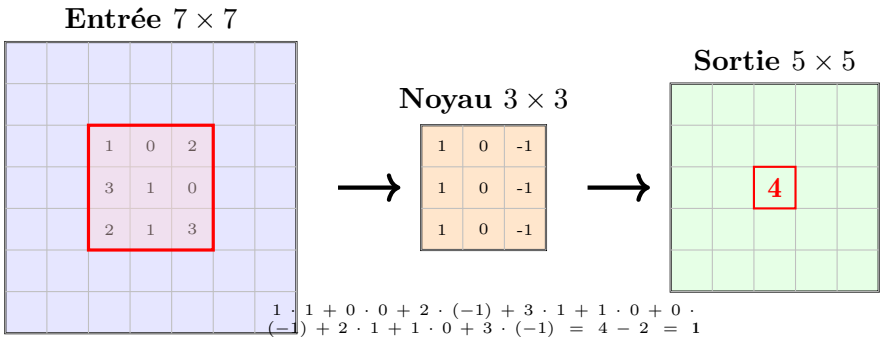
4.2.3 Convolution multi-canaux

Pour C_{in} canaux d'entrée et C_{out} filtres :

$$\mathbf{Y}[c_{\text{out}}, i, j] = b_{c_{\text{out}}} + \sum_{c_{\text{in}}=0}^{C_{\text{in}}-1} \sum_{m=0}^{k_H-1} \sum_{n=0}^{k_W-1} \mathbf{X}[c_{\text{in}}, i+m, j+n] \cdot \mathbf{K}[c_{\text{out}}, c_{\text{in}}, m, n] \tag{4.3}$$

Nombre total de paramètres : $C_{\text{out}} \times (C_{\text{in}} \times k_H \times k_W + 1)$.

4.2.4 Diagramme de l'opération de convolution



4.3 Couches de Pooling

Opérations de pooling

Pour une fenêtre de pooling $k \times k$:

$$\text{Max Pooling :} \quad y[i,j] = \max_{0 \le m,n < k} x[i \cdot s + m,\ j \cdot s + n] \tag{4.4}$$

$$\text{Average Pooling :} \quad y[i,j] = \frac{1}{k^2} \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} x[i \cdot s + m,\ j \cdot s + n] \tag{4.5}$$

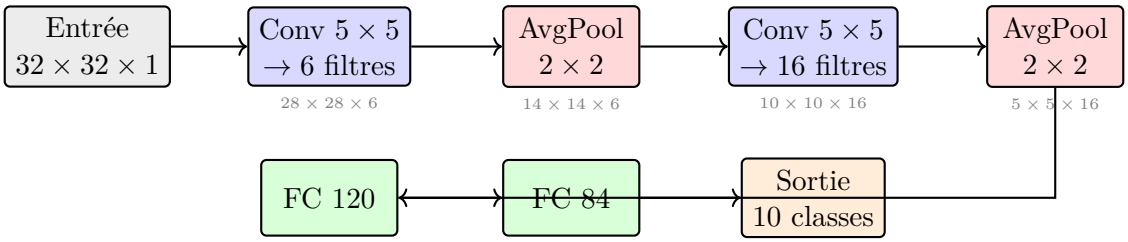
Définition 4.5 (Global Average Pooling (GAP))**.** Le GAP calcule la moyenne sur toute la carte de caractéristiques :

$$\text{GAP}(\mathbf{X})[c] = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} \mathbf{X}[c,i,j] \tag{4.6}$$

Il remplace les couches entièrement connectées finales dans les architectures modernes, réduisant le nombre de paramètres et le surapprentissage.

4.4 Architectures classiques

4.4.1 LeNet-5 (1998)



4.4.2 AlexNet (2012)

AlexNet [25] a marqué le retour des réseaux de neurones en vision par ordinateur, remportant le défi ImageNet 2012 avec une erreur top-5 de 15.3% (contre 26.2% pour les méthodes classiques). Innovations clés : ReLU, Dropout (cf. chapitre 3), entraînement sur GPU.

4.4.3 VGGNet (2014)

VGGNet [26] a démontré l'importance de la profondeur avec une architecture homogène utilisant uniquement des convolutions 3×3.

Remarque 4.6 (Champ réceptif de convolutions empilées). Deux convolutions 3×3 empilées ont un champ réceptif de 5×5, et trois convolutions 3×3 ont un champ de 7×7, mais avec moins de paramètres : $3 \times (3^2 C^2) = 27C^2$ vs $7^2 C^2 = 49C^2$.

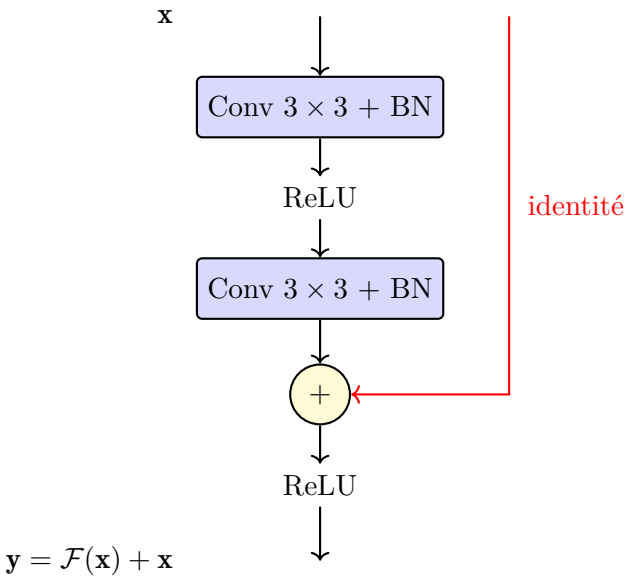
4.4.4 ResNet (2015)

Connexion résiduelle

Bloc résiduel

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + \mathbf{x} \tag{4.7}$$

où $\mathcal{F}(\mathbf{x}, \{W_i\})$ représente la transformation résiduelle (typiquement Conv-BN-ReLU-Conv-BN).



Preuve de la préservation du flux de gradient

Théorème 4.7 (Flux de gradient dans ResNet)**.** *Soit un réseau de L blocs résiduels. La sortie du bloc l est $\mathbf{x}_{l+1} = \mathcal{F}_l(\mathbf{x}_l) + \mathbf{x}_l$. Alors la sortie du bloc L peut s'exprimer comme :*

$$\mathbf{x}_L = \mathbf{x}_l + \sum_{i=l}^{L-1} \mathcal{F}_i(\mathbf{x}_i) \tag{4.8}$$

Le gradient de la perte $\mathcal{L}$ par rapport à $\mathbf{x}_l$ est :

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \mathbf{x}_l} &= \frac{\partial \mathcal{L}}{\partial \mathbf{x}_L} \cdot \frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_l} \\ &= \frac{\partial \mathcal{L}}{\partial \mathbf{x}_L} \left(1 + \frac{\partial}{\partial \mathbf{x}_l} \sum_{i=l}^{L-1} \mathcal{F}_i(\mathbf{x}_i) \right) \end{aligned} \tag{4.9}$$

Démonstration. *Par récurrence, $\mathbf{x}_{l+1} = \mathcal{F}_l(\mathbf{x}_l) + \mathbf{x}_l$ et $\mathbf{x}_{l+2} = \mathcal{F}_{l+1}(\mathbf{x}_{l+1}) + \mathbf{x}_{l+1} = \mathcal{F}_{l+1}(\mathbf{x}_{l+1}) + \mathcal{F}_l(\mathbf{x}_l) + \mathbf{x}_l$. Par induction, on obtient (4.8). En dérivant et en appliquant la règle de la chaîne, le terme $\mathbf{1}$ dans (4.9) garantit que le gradient ne s'annule jamais complètement, même si $\frac{\partial \mathcal{F}_i}{\partial \mathbf{x}_l} \to 0$.*

Pourquoi ResNet fonctionne

Le terme « +1 » dans l'équation (4.9) crée un « autoroute à gradient » (gradient highway) qui contourne les couches non linéaires. Le réseau n'a besoin d'apprendre que le **résidu** $\mathcal{F}(\mathbf{x}) = \mathbf{y} - \mathbf{x}$, qui est souvent proche de zéro et donc plus facile à apprendre.

4.5 Transfert d'apprentissage et fine-tuning

Définition 4.8 (Transfert d'apprentissage)**.** Le transfert d'apprentissage consiste à réutiliser un modèle pré-entraîné sur un grand jeu de données (ex. ImageNet) pour une nouvelle tâche, éventuellement avec un jeu de données plus petit.

Stratégies courantes :

1. **Extracteur de caractéristiques** : geler toutes les couches convolutives et n'entraîner que le classifieur final.
2. **Fine-tuning partiel** : dégeler les dernières couches convolutives avec un taux d'apprentissage réduit.
3. **Fine-tuning complet** : entraîner tout le réseau avec un taux d'apprentissage faible.

Taux d'apprentissage pour le fine-tuning

Utiliser un taux d'apprentissage 10 à 100 fois plus petit que pour l'entraînement initial. Les techniques de régularisation (chapitre 3) sont cruciales pour éviter le surapprentissage lors du fine-tuning.

4.6 Implémentation PyTorch

CNN pour CIFAR-10 avec entraînement complet

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# --- Architecture CNN ---
class CIFAR10Net(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        self.features = nn.Sequential(
            # Bloc 1: 32x32x3 -> 32x32x64 -> 16x16x64
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
```

```
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2),
            nn.Dropout2d(0.25),

            # Bloc 2: 16x16x64 -> 16x16x128 -> 8x8x128
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2),
            nn.Dropout2d(0.25),

            # Bloc 3: 8x8x128 -> 8x8x256 -> 4x4x256
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2),
            nn.Dropout2d(0.25),
        )
        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),   # Global Average Pooling
            nn.Flatten(),
            nn.Linear(256, 128),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(128, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x

# --- Pr'eparation des donn'ees ---
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465),
                         (0.2470, 0.2435, 0.2616))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
```

```
    transforms.Normalize((0.4914, 0.4822, 0.4465),
                         (0.2470, 0.2435, 0.2616))
])

train_set = datasets.CIFAR10('./data', train=True,
                             download=True, transform=transform_train)
test_set = datasets.CIFAR10('./data', train=False,
                            transform=transform_test)
train_loader = DataLoader(train_set, batch_size=128,
                          shuffle=True, num_workers=2)
test_loader = DataLoader(test_set, batch_size=256,
                         shuffle=False, num_workers=2)

# --- Entra^inement ---
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = CIFAR10Net().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-4)
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=100)

print(f"Param`etres: {sum(p.numel() for p in model.parameters()):,}")

for epoch in range(100):
    # Entra^inement
    model.train()
    train_loss, correct, total = 0.0, 0, 0
    for inputs, targets in train_loader:
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * inputs.size(0)
        correct += (outputs.argmax(1) == targets).sum().item()
        total += targets.size(0)

    # 'Evaluation
    model.eval()
    test_correct, test_total = 0, 0
    with torch.no_grad():
        for inputs, targets in test_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            test_correct += (outputs.argmax(1) == targets).sum().item()
            test_total += targets.size(0)

    scheduler.step()
    if (epoch + 1) % 10 == 0:
        print(f"Epoch {epoch+1:3d} | "
              f"Train Loss: {train_loss/total:.4f} | "
```

```
            f"Train Acc: {correct/total:.4f} | "
            f"Test Acc: {test_correct/test_total:.4f}")
```

Sortie

```
Param`etres: 952,202
Epoch  10 | Train Loss: 0.8234 | Train Acc: 0.7142 | Test Acc: 0.7856
Epoch  20 | Train Loss: 0.5321 | Train Acc: 0.8134 | Test Acc: 0.8523
Epoch  30 | Train Loss: 0.4012 | Train Acc: 0.8612 | Test Acc: 0.8802
Epoch  40 | Train Loss: 0.3245 | Train Acc: 0.8876 | Test Acc: 0.8945
Epoch  50 | Train Loss: 0.2678 | Train Acc: 0.9056 | Test Acc: 0.9034
Epoch  60 | Train Loss: 0.2234 | Train Acc: 0.9213 | Test Acc: 0.9098
Epoch  70 | Train Loss: 0.1878 | Train Acc: 0.9345 | Test Acc: 0.9145
Epoch  80 | Train Loss: 0.1534 | Train Acc: 0.9467 | Test Acc: 0.9178
Epoch  90 | Train Loss: 0.1312 | Train Acc: 0.9534 | Test Acc: 0.9201
Epoch 100 | Train Loss: 0.1198 | Train Acc: 0.9578 | Test Acc: 0.9218
```

4.7 Transfer learning avec un modèle pré-entraîné

Fine-tuning d'un ResNet-18 pré-entraîné

```python
import torchvision.models as models

# Charger ResNet-18 pr'e-entra^in'e sur ImageNet
model = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)

# Geler les couches convolutives
for param in model.parameters():
    param.requires_grad = False

# Remplacer la derni`ere couche FC
num_features = model.fc.in_features
model.fc = nn.Sequential(
    nn.Dropout(0.3),
    nn.Linear(num_features, 10)
)
model = model.to(device)

# Seule la derni`ere couche est entra^in'ee
optimizer = optim.Adam(model.fc.parameters(), lr=1e-3)

# Phase 2 : d'egeler les 2 derniers blocs pour fine-tuning
for param in model.layer3.parameters():
    param.requires_grad = True
for param in model.layer4.parameters():
    param.requires_grad = True
```

```
optimizer = optim.AdamW([
    {'params': model.layer3.parameters(), 'lr': 1e-5},
    {'params': model.layer4.parameters(), 'lr': 1e-4},
    {'params': model.fc.parameters(), 'lr': 1e-3},
], weight_decay=1e-4)
```

4.8 Étude d'ablation

Ablation : taille du noyau, nombre de filtres, profondeur

```
import pandas as pd

def build_cnn(kernel_size, num_filters, num_blocks):
    """Construit un CNN param'etrable pour l'ablation."""
    layers = []
    in_channels = 3
    for i in range(num_blocks):
        out_channels = num_filters * (2 ** min(i, 2))
        pad = kernel_size // 2
        layers.extend([
            nn.Conv2d(in_channels, out_channels, kernel_size,
            ↪  padding=pad),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(2, 2),
        ])
        in_channels = out_channels
    layers.extend([nn.AdaptiveAvgPool2d(1), nn.Flatten(),
                   nn.Linear(in_channels, 10)])
    return nn.Sequential(*layers)

configs = [
    {'kernel_size': 3, 'num_filters': 32, 'num_blocks': 3},
    {'kernel_size': 5, 'num_filters': 32, 'num_blocks': 3},
    {'kernel_size': 3, 'num_filters': 64, 'num_blocks': 3},
    {'kernel_size': 3, 'num_filters': 32, 'num_blocks': 5},
]

results = []
for cfg in configs:
    model = build_cnn(**cfg).to(device)
    n_params = sum(p.numel() for p in model.parameters())
    # Entra^iner 30 'epoques (simplifi'e)
    optimizer = optim.AdamW(model.parameters(), lr=1e-3)
    best_acc = 0.0
    for epoch in range(30):
        model.train()
        for x, y in train_loader:
```

```
            x, y = x.to(device), y.to(device)
            optimizer.zero_grad()
            loss = criterion(model(x), y)
            loss.backward()
            optimizer.step()
        model.eval()
        correct = sum((model(x.to(device)).argmax(1) ==
        ↪  y.to(device)).sum().item()
                      for x, y in test_loader)
        best_acc = max(best_acc, correct / len(test_set))
    results.append({**cfg, 'params': n_params, 'accuracy': best_acc})

print(pd.DataFrame(results).to_string(index=False))
```

Sortie

kernel_size	num_filters	num_blocks	params	accuracy
3	32	3	52778	0.8734
5	32	3	143642	0.8812
3	64	3	207946	0.9015
3	32	5	218410	0.8956

Remarque 4.9 (Observations de l'ablation).

- Doubler le nombre de filtres ($32 \to 64$) améliore davantage les performances que d'augmenter la taille du noyau ou la profondeur.
- Les noyaux 5×5 apportent un gain marginal par rapport aux 3×3 pour un coût en paramètres significatif.
- La profondeur aide, mais avec des rendements décroissants sans connexions résiduelles.

4.9 Convolutions séparables en profondeur

Convolution séparable en profondeur

Une convolution standard $C_{\text{in}} \to C_{\text{out}}$ avec noyau $k \times k$ coûte $C_{\text{in}} \cdot C_{\text{out}} \cdot k^2$ multiplications par position. La convolution séparable décompose en :

1. **Depthwise** : un filtre $k \times k$ par canal, coût $C_{\text{in}} \cdot k^2$
2. **Pointwise** : convolution 1×1 pour combiner, coût $C_{\text{in}} \cdot C_{\text{out}}$

Ratio de réduction :

$$\frac{C_{\text{in}} \cdot k^2 + C_{\text{in}} \cdot C_{\text{out}}}{C_{\text{in}} \cdot C_{\text{out}} \cdot k^2} = \frac{1}{C_{\text{out}}} + \frac{1}{k^2} \tag{4.10}$$

Pour $C_{\text{out}} = 256$ et $k = 3$: réduction d'un facteur ≈ 8–$9\times$.

4.10 Techniques état de l'art

Architectures CNN modernes

EfficientNet [27] : mise à l'échelle composée équilibrée en profondeur, largeur et résolution :

$$d = \alpha^\phi, \quad w = \beta^\phi, \quad r = \gamma^\phi \quad \text{sous} \quad \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2 \tag{4.11}$$

ConvNeXt [28] : modernisation de ResNet avec des éléments empruntés aux Transformers (cf. chapitre 6) : noyaux 7×7 depthwise, LayerNorm, GELU, fewer activation functions. Atteint des performances comparables aux Vision Transformers.

Convolutions déformables [29] : apprennent des offsets spatiaux pour adapter la forme du champ réceptif :

$$y(\mathbf{p}_0) = \sum_{n=1}^{N} w_n \cdot x(\mathbf{p}_0 + \mathbf{p}_n + \Delta\mathbf{p}_n) \tag{4.12}$$

où $\Delta\mathbf{p}_n$ sont des offsets appris.

4.11 Exercices

Exercice 4.1 (Calculs de dimensions ★)**.** Un réseau reçoit des images $128 \times 128 \times 3$ et applique successivement :

1. Conv2d(3, 32, kernel_size=5, padding=2, stride=1)
2. MaxPool2d(2, 2)
3. Conv2d(32, 64, kernel_size=3, padding=0, stride=2)
4. Conv2d(64, 128, kernel_size=3, padding=1, stride=1)
5. AdaptiveAvgPool2d(1)

Calculer la taille de sortie et le nombre de paramètres à chaque couche.

Exercice 4.2 (Implémentation d'un bloc résiduel ★★)**.**
1. Implémenter un bloc résiduel complet (avec projection si les dimensions changent) en PyTorch.
2. Construire un mini-ResNet (8 couches) pour CIFAR-10.
3. Comparer la convergence avec et sans connexions résiduelles.
4. Tracer les gradients dans les premières couches pour les deux cas.

Exercice 4.3 (Visualisation des filtres ★★)**.**
1. Entraîner un CNN sur CIFAR-10 et visualiser les filtres de la première couche convolutive.

2. Utiliser les *feature maps* intermédiaires pour comprendre ce que chaque couche détecte.

3. Implémenter la technique de *Grad-CAM* pour visualiser les régions de l'image contribuant à la décision.

Exercice 4.4 (Convolutions séparables en profondeur ★★). 1. Implémenter une couche de convolution séparable en profondeur (`nn.Conv2d` avec `groups=in_channels` + conv 1×1).

2. Remplacer toutes les convolutions 3×3 du CNN CIFAR-10 par des convolutions séparables. Comparer le nombre de paramètres et la précision.

3. Mesurer le temps d'inférence des deux versions.

Exercice 4.5 (Preuve du champ réceptif ★★★). 1. Démontrer par récurrence que n couches convolutives $k \times k$ empilées avec stride $s = 1$ produisent un champ réceptif de taille $n(k-1)+1$.

2. Généraliser pour des strides et dilations arbitraires.

3. Calculer le champ réceptif effectif d'un ResNet-50 et comparer avec la taille de l'image d'entrée.

4. Discuter le lien entre champ réceptif et capacité de modélisation.

Chapitre 5

Réseaux Récurrents et LSTM

Ce chapitre étudie les réseaux de neurones récurrents (RNN), conçus pour traiter des données séquentielles. Nous dérivons les équations de propagation, analysons le problème fondamental de la disparition et de l'explosion des gradients, puis présentons les architectures LSTM et GRU qui résolvent partiellement ce problème.

5.1 Motivation : données séquentielles

De nombreux problèmes impliquent des données ordonnées dans le temps ou dans l'espace :

- **Séries temporelles** : prévision financière, météorologie, signaux physiologiques.
- **Traitement du langage naturel** : traduction, analyse de sentiment, génération de texte.
- **Audio** : reconnaissance vocale, synthèse musicale.

Les réseaux de neurones classiques (MLP, CNN du chapitre 4) traitent des entrées de taille fixe et ne capturent pas naturellement les dépendances temporelles.

Définition 5.1 (Séquence)**.** Une séquence de longueur T est un ensemble ordonné $\mathbf{x} = (x_1, x_2, \ldots, x_T)$ où chaque $x_t \in \mathbb{R}^d$ est un vecteur d'entrée au pas de temps t.

5.2 Architecture RNN

5.2.1 Équations de récurrence

Un RNN simple (« Vanilla RNN ») maintient un état caché $h_t \in \mathbb{R}^{d_h}$ qui résume l'information des pas de temps précédents :

RNN – Équations fondamentales

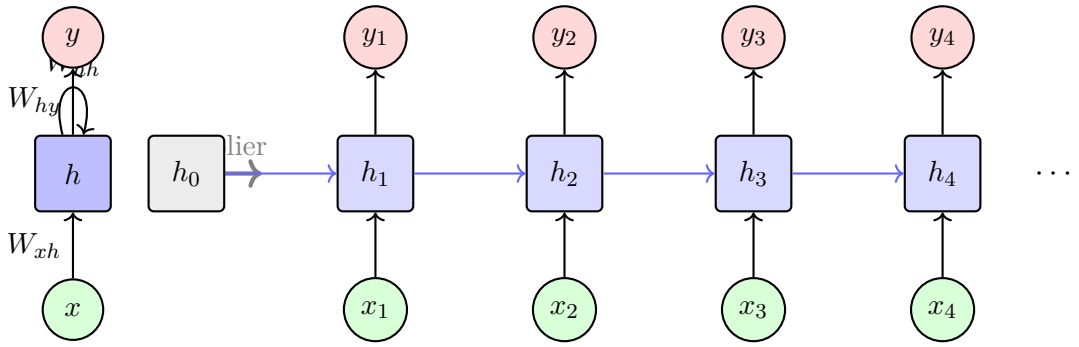
$$h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t + b_h) \tag{5.1}$$
$$y_t = W_{hy} h_t + b_y \tag{5.2}$$

où :

- $W_{xh} \in \mathbb{R}^{d_h \times d}$: poids entrée → caché
- $W_{hh} \in \mathbb{R}^{d_h \times d_h}$: poids caché → caché (récurrence)
- $W_{hy} \in \mathbb{R}^{d_y \times d_h}$: poids caché → sortie
- b_h, b_y : biais

Remarque 5.2 (Partage des paramètres). Les matrices W_{xh}, W_{hh} et W_{hy} sont **partagées** à travers tous les pas de temps, ce qui permet de traiter des séquences de longueur variable.

5.2.2 Graphe de calcul déplié



5.3 Rétropropagation à travers le temps (BPTT)

5.3.1 Dérivation complète

La perte totale sur une séquence de longueur T est :

$$\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}_t(y_t, \hat{y}_t) \tag{5.3}$$

Le gradient par rapport à W_{hh} nécessite de sommer les contributions de tous les pas de temps :

Théorème 5.3 (Gradient BPTT par rapport à W_{hh})**.**

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial W_{hh}} &= \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial W_{hh}} \\ &= \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \sum_{k=1}^{t} \left(\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_k}{\partial W_{hh}} \end{aligned} \tag{5.4}$$

Démonstration. Par la règle de la chaîne, h_t dépend de W_{hh} à la fois directement (au temps t) et indirectement (via $h_{t-1}, h_{t-2}, \ldots$). On a :

$$\frac{\partial \mathcal{L}_t}{\partial W_{hh}} = \frac{\partial \mathcal{L}_t}{\partial h_t} \cdot \frac{\partial^+ h_t}{\partial W_{hh}} + \frac{\partial \mathcal{L}_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial \mathcal{L}_{t-1}}{\partial W_{hh}} \tag{5.5}$$

En déroulant la récurrence, on obtient une somme de produits de Jacobiennes :

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^{t} W_{hh}^{\top} \cdot \mathrm{diag}(\tanh'(z_j)) \tag{5.6}$$

où $z_j = W_{hh} h_{j-1} + W_{xh} x_j + b_h$.

5.4 Disparition et explosion des gradients

5.4.1 Analyse par les valeurs singulières

Théorème 5.4 (Condition de disparition/explosion)**.** *Soit $\sigma_{\max}$ la plus grande valeur singulière de W_{hh} et $\gamma = \max |\tanh'(z)| \leq 1$. Alors :*

$$\left\| \frac{\partial h_t}{\partial h_k} \right\| = \left\| \prod_{j=k+1}^{t} W_{hh}^{\top} diag(\tanh'(z_j)) \right\| \leq (\sigma_{\max} \cdot \gamma)^{t-k} \tag{5.7}$$

- *Si $\sigma_{\max} \cdot \gamma < 1$: le gradient* ***décroît exponentiellement*** *avec $(t-k) \to$* disparition des gradients.
- *Si $\sigma_{\max} \cdot \gamma > 1$: le gradient* ***croît exponentiellement*** $\to$ explosion des gradients.

Conséquence pratique

Pour $t - k = 100$ et $\sigma_{\max}\gamma = 0.9$: $\|\partial h_t / \partial h_k\| \leq 0.9^{100} \approx 2.66 \times 10^{-5}$. Le RNN ne peut pas apprendre de dépendances au-delà de 10–20 pas de temps.

5.4.2 Gradient Clipping

Pour combattre l'explosion des gradients :

Gradient Clipping par norme

$$\hat{g} = \begin{cases} g & \text{si } \|g\| \leq \tau \\ \dfrac{\tau}{\|g\|} \cdot g & \text{si } \|g\| > \tau \end{cases} \tag{5.8}$$

5.5 Long Short-Term Memory (LSTM)

Le LSTM [30] résout le problème de la disparition des gradients en introduisant une **cellule mémoire** c_t et des **portes** (gates) qui contrôlent le flux d'information.

5.5.1 Les quatre portes

Équations du LSTM

$$\text{Porte d'oubli :} \quad f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{5.9}$$

$$\text{Porte d'entrée :} \quad i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{5.10}$$

$$\text{Candidat cellule :} \quad \tilde{c}_t = \tanh(W_c[h_{t-1}, x_t] + b_c) \tag{5.11}$$

$$\text{Mise à jour cellule :} \quad c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{5.12}$$

$$\text{Porte de sortie :} \quad o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{5.13}$$

$$\text{État caché :} \quad h_t = o_t \odot \tanh(c_t) \tag{5.14}$$

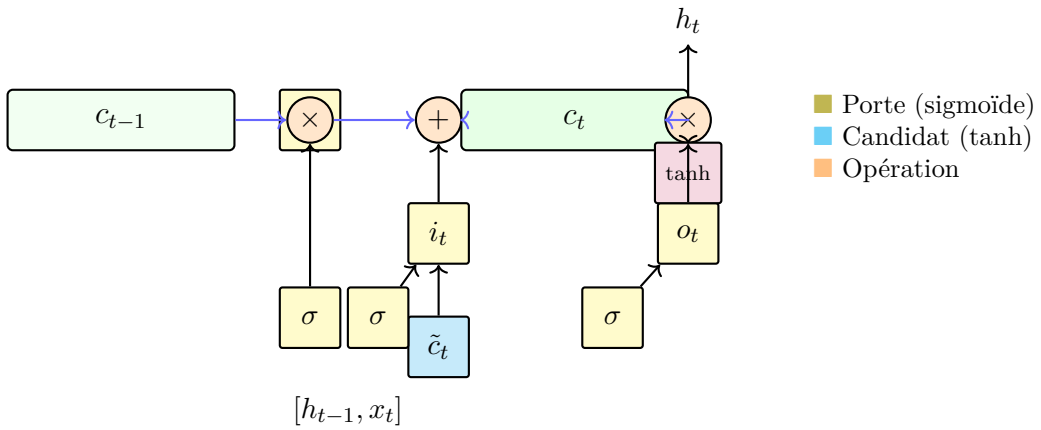
où $[h_{t-1}, x_t]$ dénote la concaténation, $\odot$ le produit de Hadamard, et σ la fonction sigmoïde.

Rôle des portes

- **Porte d'oubli** f_t : décide quelle information de la cellule précédente c_{t-1} doit être oubliée ($f_t \to 0$) ou conservée ($f_t \to 1$).
- **Porte d'entrée** i_t : décide quelle nouvelle information doit être écrite dans la cellule.
- **Porte de sortie** o_t : décide quelle partie de la cellule est exposée comme état caché.

La cellule c_t agit comme un « tapis roulant » (conveyor belt) sur lequel l'information peut transiter sans dégradation, de manière similaire aux connexions résiduelles de ResNet (chapitre 4, théorème 4.7).

5.5.2 Diagramme de la cellule LSTM



5.5.3 Préservation du gradient dans le LSTM

Proposition 5.5 (Flux de gradient à travers la cellule)**.** Le gradient de $\mathcal{L}$ par rapport à c_k $(k < t)$ est :

$$\frac{\partial c_t}{\partial c_k} = \prod_{j=k+1}^{t} f_j \tag{5.15}$$

Si $f_j \approx 1$ (la porte d'oubli est ouverte), le gradient se propage sans atténuation. Ceci contraste avec le RNN simple où le gradient passe à travers $W_{hh}^\top \text{diag}(\tanh'(\cdot))$ (équation 5.6).

5.6 Gated Recurrent Unit (GRU)

Le GRU [31] simplifie le LSTM en fusionnant la cellule et l'état caché, et en n'utilisant que deux portes :

Équations du GRU

$$\text{Porte de mise à jour :} \quad z_t = \sigma(W_z[h_{t-1}, x_t] + b_z) \tag{5.16}$$
$$\text{Porte de réinitialisation :} \quad r_t = \sigma(W_r[h_{t-1}, x_t] + b_r) \tag{5.17}$$
$$\text{Candidat :} \quad \tilde{h}_t = \tanh(W_h[r_t \odot h_{t-1}, x_t] + b_h) \tag{5.18}$$
$$\text{État caché :} \quad h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{5.19}$$

Remarque 5.6 (Comparaison LSTM vs GRU).

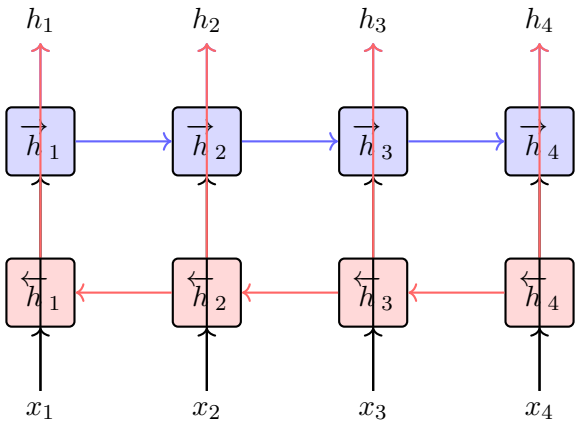
	LSTM	**GRU**
Portes	3 (f, i, o)	2 (z, r)
États	2 (h_t, c_t)	1 (h_t)
Paramètres (pour d_h, d_x)	$4d_h(d_h + d_x + 1)$	$3d_h(d_h + d_x + 1)$
Dépendances longues	Excellent	Bon
Vitesse	Plus lent	Plus rapide ($\sim 25\%$)

En pratique, les performances sont souvent comparables. Le LSTM est généralement préféré pour les séquences très longues.

5.7 RNN bidirectionnel

Définition 5.7 (RNN bidirectionnel)**.** Un RNN bidirectionnel traite la séquence dans les deux sens et concatène les états cachés :

$$\overrightarrow{h}_t = \text{RNN}_{\text{avant}}(x_t, \overrightarrow{h}_{t-1}) \tag{5.20}$$
$$\overleftarrow{h}_t = \text{RNN}_{\text{arrière}}(x_t, \overleftarrow{h}_{t+1}) \tag{5.21}$$
$$h_t = [\overrightarrow{h}_t; \overleftarrow{h}_t] \in \mathbb{R}^{2d_h} \tag{5.22}$$



5.8 Implémentation PyTorch

Classification de séquences avec LSTM bidirectionnel

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from torch.nn.utils.rnn import pad_sequence, pack_padded_sequence

# --- Mod`ele LSTM pour la classification de sentiment ---
class SentimentLSTM(nn.Module):
    def __init__(self, vocab_size, embed_dim=128, hidden_dim=256,
                 num_layers=2, num_classes=2, dropout=0.3,
                 bidirectional=True):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim,
                                      padding_idx=0)
        self.lstm = nn.LSTM(
            input_size=embed_dim,
            hidden_size=hidden_dim,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout if num_layers > 1 else 0,
            bidirectional=bidirectional
        )
        self.num_directions = 2 if bidirectional else 1
        self.classifier = nn.Sequential(
            nn.Dropout(dropout),
            nn.Linear(hidden_dim * self.num_directions, 128),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(128, num_classes)
        )

    def forward(self, x, lengths):
        # x: (batch, max_len), lengths: (batch,)
```

```
        embedded = self.embedding(x)  # (batch, max_len, embed_dim)

        # Pack pour ignorer le padding
        packed = pack_padded_sequence(
            embedded, lengths.cpu(), batch_first=True,
            enforce_sorted=False
        )
        _, (h_n, _) = self.lstm(packed)
        # h_n: (num_layers * num_directions, batch, hidden_dim)

        # Concat'ener les 'etats forward et backward
        # de la derni`ere couche
        if self.num_directions == 2:
            hidden = torch.cat([h_n[-2], h_n[-1]], dim=1)
        else:
            hidden = h_n[-1]

        return self.classifier(hidden)

# --- Exemple de donn'ees synth'etiques ---
class SyntheticSentiment(Dataset):
    """Donn'ees synth'etiques pour d'emonstration."""
    def __init__(self, num_samples=5000, vocab_size=1000, max_len=50):
        self.data = []
        for _ in range(num_samples):
            length = torch.randint(5, max_len, (1,)).item()
            tokens = torch.randint(1, vocab_size, (length,))
            # Etiquette bas'ee sur la pr'esence de certains tokens
            label = int(tokens.float().mean() > vocab_size / 2)
            self.data.append((tokens, label))

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]

def collate_fn(batch):
    """Padding dynamique pour le batch."""
    tokens, labels = zip(*batch)
    lengths = torch.tensor([len(t) for t in tokens])
    padded = pad_sequence(tokens, batch_first=True, padding_value=0)
    labels = torch.tensor(labels)
    return padded, labels, lengths

# --- Entra^inement ---
vocab_size = 1000
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = SentimentLSTM(vocab_size).to(device)
optimizer = optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-4)
criterion = nn.CrossEntropyLoss()
```

```python
dataset = SyntheticSentiment(num_samples=5000, vocab_size=vocab_size)
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_set, val_set = torch.utils.data.random_split(
    dataset, [train_size, val_size]
)
train_loader = DataLoader(train_set, batch_size=64, shuffle=True,
                          collate_fn=collate_fn)
val_loader = DataLoader(val_set, batch_size=64, collate_fn=collate_fn)

n_params = sum(p.numel() for p in model.parameters())
print(f"Param`etres: {n_params:,}")

for epoch in range(20):
    model.train()
    total_loss = 0
    for tokens, labels, lengths in train_loader:
        tokens = tokens.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        logits = model(tokens, lengths)
        loss = criterion(logits, labels)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        total_loss += loss.item()

    # Validation
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for tokens, labels, lengths in val_loader:
            tokens = tokens.to(device)
            labels = labels.to(device)
            logits = model(tokens, lengths)
            correct += (logits.argmax(1) == labels).sum().item()
            total += labels.size(0)

    if (epoch + 1) % 5 == 0:
        print(f"Epoch {epoch+1:2d} | Loss:
        ↪  {total_loss/len(train_loader):.4f}"
              f" | Val Acc: {correct/total:.4f}")
```

Sortie

```
Param`etres: 1,447,170
Epoch  5 | Loss: 0.5823 | Val Acc: 0.7120
Epoch 10 | Loss: 0.3912 | Val Acc: 0.8210
```

```
Epoch 15 | Loss: 0.2534 | Val Acc: 0.8650
Epoch 20 | Loss: 0.1823 | Val Acc: 0.8890
```

5.9 Étude d'ablation

Ablation : hidden size, couches, LSTM vs GRU

```python
import pandas as pd

class FlexibleRNN(nn.Module):
    def __init__(self, vocab_size, rnn_type='LSTM',
                 hidden_dim=128, num_layers=1):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, 128, padding_idx=0)
        RNNClass = nn.LSTM if rnn_type == 'LSTM' else nn.GRU
        self.rnn = RNNClass(128, hidden_dim, num_layers,
                            batch_first=True, bidirectional=True)
        self.fc = nn.Linear(hidden_dim * 2, 2)

    def forward(self, x, lengths):
        embedded = self.embedding(x)
        packed = pack_padded_sequence(embedded, lengths.cpu(),
                                      batch_first=True,
                                      enforce_sorted=False)
        if isinstance(self.rnn, nn.LSTM):
            _, (h_n, _) = self.rnn(packed)
        else:
            _, h_n = self.rnn(packed)
        hidden = torch.cat([h_n[-2], h_n[-1]], dim=1)
        return self.fc(hidden)

configs = [
    {'rnn_type': 'LSTM', 'hidden_dim': 64,  'num_layers': 1},
    {'rnn_type': 'LSTM', 'hidden_dim': 128, 'num_layers': 1},
    {'rnn_type': 'LSTM', 'hidden_dim': 256, 'num_layers': 1},
    {'rnn_type': 'LSTM', 'hidden_dim': 128, 'num_layers': 2},
    {'rnn_type': 'LSTM', 'hidden_dim': 128, 'num_layers': 3},
    {'rnn_type': 'GRU',  'hidden_dim': 128, 'num_layers': 1},
    {'rnn_type': 'GRU',  'hidden_dim': 128, 'num_layers': 2},
]

results = []
for cfg in configs:
    model = FlexibleRNN(vocab_size, **cfg).to(device)
    n_params = sum(p.numel() for p in model.parameters())
    optimizer = optim.Adam(model.parameters(), lr=1e-3)
    # Entra^iner 15 'epoques
    best_acc = 0.0
    for epoch in range(15):
```

```
        model.train()
        for tokens, labels, lengths in train_loader:
            tokens, labels = tokens.to(device), labels.to(device)
            optimizer.zero_grad()
            loss = criterion(model(tokens, lengths), labels)
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            optimizer.step()
        model.eval()
        correct = total = 0
        with torch.no_grad():
            for tokens, labels, lengths in val_loader:
                tokens, labels = tokens.to(device), labels.to(device)
                preds = model(tokens, lengths).argmax(1)
                correct += (preds == labels).sum().item()
                total += labels.size(0)
        best_acc = max(best_acc, correct / total)
    results.append({**cfg, 'params': n_params, 'val_acc': best_acc})

print(pd.DataFrame(results).to_string(index=False))
```

Sortie

rnn_type	hidden_dim	num_layers	params	val_acc
LSTM	64	1	226434	0.8234
LSTM	128	1	560642	0.8650
LSTM	256	1	1568258	0.8812
LSTM	128	2	823810	0.8745
LSTM	128	3	1086978	0.8701
GRU	128	1	432386	0.8590
GRU	128	2	630530	0.8678

Remarque 5.8 (Observations).

- Augmenter la dimension cachée de 64 à 256 améliore régulièrement les performances, mais avec des rendements décroissants.
- Empiler trop de couches (3) sans Dropout peut dégrader les performances par sur-apprentissage.
- Le GRU offre des performances comparables au LSTM avec $\sim 25\%$ de paramètres en moins, confirmant la littérature.

5.10 Connexions avec l'état de l'art

Au-delà des RNN classiques

Transformers (cf. chapitre 6) : les mécanismes d'attention ont largement remplacé les RNN pour le traitement du langage naturel grâce à leur capacité à capturer les dépendances longues sans récurrence et à leur parallélisabilité.

Modèles d'espace d'états (SSM) : une nouvelle famille de modèles séquentiels inspirés de la théorie du contrôle :

$$h'(t) = Ah(t) + Bx(t) \tag{5.23}$$
$$y(t) = Ch(t) + Dx(t) \tag{5.24}$$

Après discrétisation, ces modèles permettent un traitement séquentiel efficace avec une complexité en $O(T \log T)$ via des convolutions.

Mamba [32] : un SSM sélectif avec des matrices A, B, C dépendantes de l'entrée. Mamba atteint des performances comparables aux Transformers sur le langage et surpasse les Transformers sur certaines tâches séquentielles longues, avec une complexité linéaire en la longueur de la séquence.

xLSTM [33] : extension récente du LSTM avec des portes exponentielles et un mécanisme de mémoire matricielle, montrant que l'architecture LSTM peut être compétitive avec les Transformers après modernisation.

5.11 Exercices

Exercice 5.1 (RNN à partir de zéro ★). 1. Implémenter un RNN simple (équations 5.1–5.2) en PyTorch **sans utiliser** `nn.RNN`.

2. Appliquer ce RNN à la génération de texte caractère par caractère sur un petit corpus (ex. Shakespeare).

3. Comparer les résultats avec `nn.RNN` pour vérifier la correction.

Exercice 5.2 (Analyse du gradient dans le RNN ★★). 1. Écrire une fonction qui calcule $\|\partial h_t / \partial h_k\|$ pour un RNN entraîné sur une séquence de longueur $T = 100$.

2. Tracer $\|\partial h_t / \partial h_k\|$ en fonction de $(t-k)$ pour un RNN simple, un LSTM et un GRU.

3. Vérifier expérimentalement la borne de l'équation 5.7.

4. Étudier l'effet de l'initialisation de W_{hh} (orthogonale vs aléatoire) sur la disparition du gradient.

Exercice 5.3 (LSTM pour séries temporelles ★★). 1. Télécharger les données de température quotidienne d'une ville.

2. Préparer les données avec une fenêtre glissante de 30 jours pour prédire le jour suivant.

3. Entraîner un LSTM à 2 couches et comparer avec un GRU.

4. Implémenter une prédiction multi-pas (7 jours) de manière autorégressive et en mode « teacher forcing ».

Exercice 5.4 (Portes du LSTM ★★★). 1. Entraîner un LSTM sur le problème de la « copie » : reproduire une séquence de 8 symboles après un délai de $T \in \{10, 50, 100\}$ pas de temps.

2. Visualiser les activations des portes d'oubli f_t, d'entrée i_t et de sortie o_t au cours de la séquence.

3. Interpréter le comportement des portes : quand la porte d'oubli s'ouvre-t-elle ? Quand se ferme-t-elle ?

4. Comparer avec un RNN simple : à partir de quelle longueur T le RNN échoue-t-il ?

Exercice 5.5 (Comparaison LSTM / GRU / Transformer ★★★)**.**

1. Choisir un jeu de données de classification de texte (ex. IMDB, AG News).

2. Implémenter et entraîner un LSTM bidirectionnel, un GRU bidirectionnel, et un petit Transformer encoder (cf. chapitre 6).

3. Comparer : précision, temps d'entraînement par époque, nombre de paramètres, mémoire GPU.

4. Analyser les performances en fonction de la longueur des séquences.

5. Discuter dans quels scénarios les RNN restent compétitifs.

Chapitre 6

Mécanisme d'Attention

Ce chapitre présente le mécanisme d'attention, innovation fondamentale qui a révolutionné l'apprentissage profond. Nous dérivons mathématiquement les différentes formes d'attention (Bahdanau, Luong, scaled dot-product), le mécanisme multi-tête, et analysons la complexité computationnelle. Ce chapitre prépare le terrain pour l'architecture Transformer.

6.1 Motivation : limites du vecteur de contexte fixe

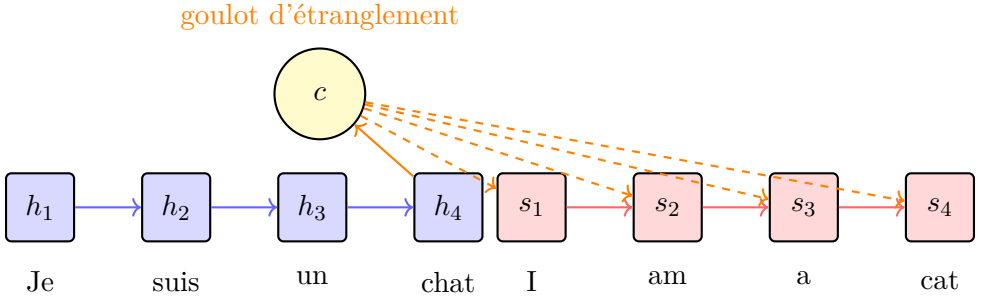
6.1.1 Le problème du goulot d'étranglement

Dans les modèles séquence-à-séquence (seq2seq) classiques [34], l'encodeur (un RNN ou LSTM, cf. chapitre 5) compresse toute la séquence source en un unique vecteur de contexte $c = h_T$:

$$c = \text{Encodeur}(x_1, x_2, \ldots, x_T) = h_T \in \mathbb{R}^{d_h} \tag{6.1}$$

Goulot d'étranglement informationnel

Ce vecteur unique c doit capturer **toute** l'information de la séquence source, quelle que soit sa longueur. Les performances se dégradent significativement pour les séquences longues ($T > 20$).



6.2 Attention de Bahdanau (additive)

6.2.1 Scores d'alignement

L'attention de Bahdanau [15] calcule un score d'alignement entre l'état du décodeur s_{t-1} et chaque état de l'encodeur h_j :

Attention de Bahdanau

$$e_{t,j} = v_a^\top \tanh(W_a s_{t-1} + U_a h_j) \tag{6.2}$$

$$\alpha_{t,j} = \frac{\exp(e_{t,j})}{\sum_{k=1}^{T} \exp(e_{t,k})} = \mathrm{softmax}_j(e_{t,:}) \tag{6.3}$$

$$c_t = \sum_{j=1}^{T} \alpha_{t,j} h_j \tag{6.4}$$

où $W_a \in \mathbb{R}^{d_a \times d_s}$, $U_a \in \mathbb{R}^{d_a \times d_h}$, $v_a \in \mathbb{R}^{d_a}$ sont des paramètres apprenables et d_a est la dimension de l'attention.

6.2.2 Dérivation du vecteur de contexte

Théorème 6.1 (Vecteur de contexte comme combinaison convexe)**.** *Les poids d'attention $\alpha_{t,j}$ satisfont :*

$$\alpha_{t,j} \geq 0, \qquad \sum_{j=1}^{T} \alpha_{t,j} = 1 \tag{6.5}$$

Le vecteur de contexte c_t est donc une ***combinaison convexe*** *des états de l'encodeur. Chaque $\alpha_{t,j}$ représente la « pertinence » de la position source j pour générer le token cible au temps t.*

Attention comme recherche souple

L'attention peut être vue comme une recherche souple (soft retrieval) dans une mémoire : les $\alpha_{t,j}$ sont des « poids d'accès » qui déterminent quelles positions source sont lues à chaque pas du décodeur. Contrairement à un index discret, cette recherche est différentiable.

6.3 Attention de Luong

Luong et al. [35] proposent trois variantes du score d'alignement, plus simples que Bahdanau :

Scores de Luong

$$\text{Dot :} \quad e_{t,j} = s_t^\top h_j \tag{6.6}$$
$$\text{Général :} \quad e_{t,j} = s_t^\top W_a h_j \tag{6.7}$$
$$\text{Concat :} \quad e_{t,j} = v_a^\top \tanh(W_a[s_t; h_j]) \tag{6.8}$$

Remarque 6.2 (Bahdanau vs Luong).
- Bahdanau utilise s_{t-1} (avant la mise à jour du décodeur), Luong utilise s_t (après).
- Le score dot-product de Luong ne nécessite aucun paramètre supplémentaire mais requiert $d_s = d_h$.
- Le score général ajoute $d_s \times d_h$ paramètres et gère des dimensions différentes.

6.4 Scaled Dot-Product Attention

6.4.1 Formulation

L'attention par produit scalaire mis à l'échelle [36] généralise le concept d'attention en introduisant les notions de **requête** (Query), **clé** (Key) et **valeur** (Value) :

Scaled Dot-Product Attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V \tag{6.9}$$

où $Q \in \mathbb{R}^{n \times d_k}$, $K \in \mathbb{R}^{m \times d_k}$, $V \in \mathbb{R}^{m \times d_v}$, et d_k est la dimension des clés.

6.4.2 Pourquoi diviser par $\sqrt{d_k}$?

Théorème 6.3 (Justification du facteur d'échelle)**.** *Soient $q, k \in \mathbb{R}^{d_k}$ avec des composantes i.i.d. de moyenne 0 et variance 1. Le produit scalaire $q^\top k = \sum_{i=1}^{d_k} q_i k_i$ satisfait :*

$$\mathbb{E}[q^\top k] = \sum_{i=1}^{d_k} \mathbb{E}[q_i]\mathbb{E}[k_i] = 0 \tag{6.10}$$
$$\text{Var}(q^\top k) = \sum_{i=1}^{d_k} \text{Var}(q_i k_i) = \sum_{i=1}^{d_k} \mathbb{E}[q_i^2]\mathbb{E}[k_i^2] = d_k \tag{6.11}$$

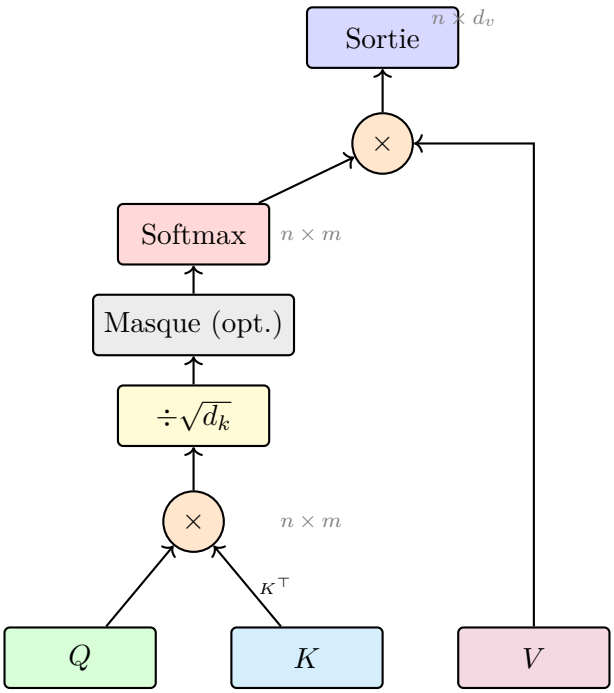
Pour de grandes valeurs de d_k, les produits scalaires ont une grande variance, ce qui pousse le softmax vers des régions saturées où les gradients sont quasi nuls. La division par $\sqrt{d_k}$ normalise la variance à 1 :

$$\text{Var}\left(\frac{q^\top k}{\sqrt{d_k}}\right) = \frac{\text{Var}(q^\top k)}{d_k} = 1 \tag{6.12}$$

Saturation du softmax

Quand les entrées du softmax sont grandes en valeur absolue, la sortie devient quasi one-hot et $\frac{\partial \text{softmax}}{\partial z} \approx 0$. L'entraînement stagne. Le facteur $1/\sqrt{d_k}$ prévient ce problème.

6.4.3 Diagramme de l'attention



6.5 Attention multi-tête

6.5.1 Dérivation

Plutôt qu'une seule fonction d'attention de dimension d_{model}, l'attention multi-tête projette les requêtes, clés et valeurs h fois dans des sous-espaces différents :

Attention multi-tête

$$\text{head}_i = \text{Attention}(QW_i^Q,\ KW_i^K,\ VW_i^V) \tag{6.13}$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{6.14}$$

où les projections sont :

- $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$
- $W^O \in \mathbb{R}^{h \cdot d_v \times d_{\text{model}}}$
- $d_k = d_v = d_{\text{model}}/h$

Pourquoi plusieurs têtes ?

Chaque tête d'attention peut capturer un type de relation différent :

- Tête 1 : relations syntaxiques (sujet-verbe)
- Tête 2 : relations positionnelles (mots adjacents)
- Tête 3 : références anaphoriques (pronoms)
- ...

L'attention mono-tête devrait capturer toutes ces relations simultanément dans un seul calcul de poids, ce qui limite l'expressivité.

Remarque 6.4 (Coût computationnel). L'attention multi-tête avec h têtes de dimension $d_k = d_{\text{model}}/h$ a le même coût que l'attention mono-tête de dimension d_{model} :

$$h \times \underbrace{O(n^2 \cdot d_{\text{model}}/h)}_{\text{par tête}} = O(n^2 \cdot d_{\text{model}}) \tag{6.15}$$

6.6 Self-attention et cross-attention

6.6.1 Self-attention

Définition 6.5 (Self-attention)**.** Dans la self-attention, les requêtes, clés et valeurs sont toutes dérivées de la **même** séquence d'entrée $X \in \mathbb{R}^{n \times d}$:

$$Q = XW^Q \tag{6.16}$$
$$K = XW^K \tag{6.17}$$
$$V = XW^V \tag{6.18}$$

Chaque position peut « assister » à toutes les autres positions de la même séquence. La matrice $\text{softmax}(QK^\top/\sqrt{d_k})$ de taille $n \times n$ encode les dépendances position-à-position.

6.6.2 Cross-attention

Définition 6.6 (Cross-attention)**.** Dans la cross-attention, les requêtes proviennent d'une séquence (le décodeur) et les clés/valeurs proviennent d'une autre (l'encodeur) :

$$Q = X_{\text{dec}}W^Q, \qquad K = X_{\text{enc}}W^K, \qquad V = X_{\text{enc}}W^V \tag{6.19}$$

Cela permet au décodeur d'accéder à l'information de l'encodeur, remplaçant le vecteur de contexte fixe des modèles seq2seq classiques.

6.7 Complexité computationnelle

Théorème 6.7 (Complexité de la self-attention)**.** *Pour une séquence de longueur n et une dimension d :*

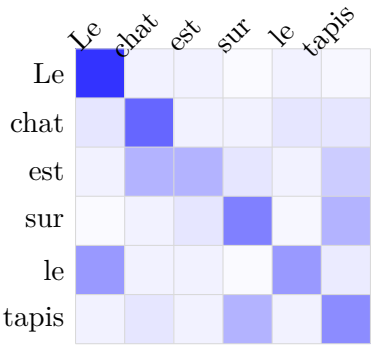
Opération	***Temps***	***Mémoire***
Projections QKV	$O(nd^2)$	$O(nd)$
$QK^\top$	$O(n^2 d)$	$O(n^2)$
Softmax	$O(n^2)$	$O(n^2)$
Produit avec V	$O(n^2 d)$	$O(nd)$
Total	$O(n^2 d)$	$O(n^2 + nd)$

Le terme $O(n^2)$ en mémoire pour stocker la matrice d'attention est le principal goulot d'étranglement pour les longues séquences.

Remarque 6.8 (Comparaison avec les RNN). Un RNN a une complexité temporelle $O(nd^2)$ (linéaire en n) mais ne peut pas paralléliser les calculs le long de la séquence. La self-attention est $O(n^2 d)$ mais entièrement parallélisable. Pour $n < d$ (cas fréquent), la self-attention est plus rapide en pratique.

6.8 Concept de carte d'attention

Poids d'attention α_{ij}



Colonne = clé (source), Ligne = requête (cible)

Remarque 6.9 (Interprétation de la carte d'attention). La case (i, j) de la matrice représente à quel point la position i (requête) « attend » la position j (clé). On observe que :

- « Le » (position 1) attend principalement à lui-même et à « le » (position 5) – cohérence de l'article.
- « tapis » attend fortement « sur » – relation spatiale.

6.9 Implémentation PyTorch

Scaled Dot-Product Attention à partir de zéro

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```python
import math

def scaled_dot_product_attention(
    query: torch.Tensor,    # (batch, n_q, d_k)
    key: torch.Tensor,      # (batch, n_kv, d_k)
    value: torch.Tensor,    # (batch, n_kv, d_v)
    mask: torch.Tensor = None,
    dropout: nn.Dropout = None
) -> tuple[torch.Tensor, torch.Tensor]:
    """
    Calcule l'attention par produit scalaire mis `a l''echelle.

    Returns:
        output: (batch, n_q, d_v)
        attention_weights: (batch, n_q, n_kv)
    """
    d_k = query.size(-1)

    # Scores d'attention : (batch, n_q, n_kv)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)

    # Masquage optionnel (pour l'attention causale)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, float('-inf'))

    # Normalisation softmax
    attention_weights = F.softmax(scores, dim=-1)

    if dropout is not None:
        attention_weights = dropout(attention_weights)

    # Combinaison lin'eaire des valeurs
    output = torch.matmul(attention_weights, value)

    return output, attention_weights


# --- Test ---
batch_size, n_q, n_kv, d_k, d_v = 2, 4, 6, 64, 64
Q = torch.randn(batch_size, n_q, d_k)
K = torch.randn(batch_size, n_kv, d_k)
V = torch.randn(batch_size, n_kv, d_v)

output, weights = scaled_dot_product_attention(Q, K, V)
print(f"Output shape:  {output.shape}")    # (2, 4, 64)
print(f"Weights shape: {weights.shape}")   # (2, 4, 6)
print(f"Weights sum:   {weights.sum(-1)}")  # Doit ^etre 1.0
```

Sortie

```
Output shape:  torch.Size([2, 4, 64])
Weights shape: torch.Size([2, 4, 6])
Weights sum:   tensor([[1.0000, 1.0000, 1.0000, 1.0000],
                       [1.0000, 1.0000, 1.0000, 1.0000]])
```

Multi-Head Attention complète

```python
class MultiHeadAttention(nn.Module):
    """
    Attention multi-t^ete compl`ete.

    Args:
        d_model: dimension du mod`ele
        num_heads: nombre de t^etes d'attention
        dropout: taux de dropout sur les poids d'attention
    """
    def __init__(self, d_model: int, num_heads: int,
                 dropout: float = 0.1):
        super().__init__()
        assert d_model % num_heads == 0, \
            "d_model doit ^etre divisible par num_heads"

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        # Projections lin'eaires
        self.W_q = nn.Linear(d_model, d_model, bias=False)
        self.W_k = nn.Linear(d_model, d_model, bias=False)
        self.W_v = nn.Linear(d_model, d_model, bias=False)
        self.W_o = nn.Linear(d_model, d_model, bias=False)

        self.dropout = nn.Dropout(dropout)
        self.attention_weights = None  # Pour visualisation

    def forward(
        self,
        query: torch.Tensor,    # (batch, n_q, d_model)
        key: torch.Tensor,      # (batch, n_kv, d_model)
        value: torch.Tensor,    # (batch, n_kv, d_model)
        mask: torch.Tensor = None
    ) -> torch.Tensor:
        batch_size = query.size(0)

        # 1. Projections lin'eaires
        Q = self.W_q(query)  # (batch, n_q, d_model)
        K = self.W_k(key)
        V = self.W_v(value)
```

```
        # 2. Reshape pour multi-t^ete :
        #    (batch, n, d_model) -> (batch, num_heads, n, d_k)
        Q = Q.view(batch_size, -1, self.num_heads, self.d_k) \
             .transpose(1, 2)
        K = K.view(batch_size, -1, self.num_heads, self.d_k) \
             .transpose(1, 2)
        V = V.view(batch_size, -1, self.num_heads, self.d_k) \
             .transpose(1, 2)

        # 3. Adapter le masque pour les t^etes
        if mask is not None:
            mask = mask.unsqueeze(1)  # (batch, 1, ...)

        # 4. Attention par t^ete
        scores = torch.matmul(Q, K.transpose(-2, -1)) \
                 / math.sqrt(self.d_k)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))
        attn_weights = F.softmax(scores, dim=-1)
        attn_weights = self.dropout(attn_weights)
        self.attention_weights = attn_weights.detach()

        context = torch.matmul(attn_weights, V)
        # (batch, num_heads, n_q, d_k)

        # 5. Concat'ener les t^etes
        context = context.transpose(1, 2).contiguous() \
                         .view(batch_size, -1, self.d_model)
        # (batch, n_q, d_model)

        # 6. Projection de sortie
        output = self.W_o(context)

        return output

# --- Test ---
d_model, num_heads = 512, 8
mha = MultiHeadAttention(d_model, num_heads)

# Self-attention
x = torch.randn(2, 10, d_model)  # batch=2, seq_len=10
out = mha(x, x, x)
print(f"Self-attention output: {out.shape}")  # (2, 10, 512)

# Cross-attention
enc_out = torch.randn(2, 20, d_model)
dec_in = torch.randn(2, 10, d_model)
out = mha(query=dec_in, key=enc_out, value=enc_out)
print(f"Cross-attention output: {out.shape}")  # (2, 10, 512)

# Masque causal
```

```python
n = 10
causal_mask = torch.tril(torch.ones(n, n)).unsqueeze(0)  # (1, n, n)
out = mha(x, x, x, mask=causal_mask)
print(f"Masked attention output: {out.shape}")

# Nombre de param`etres
n_params = sum(p.numel() for p in mha.parameters())
print(f"Param`etres MHA: {n_params:,}")
# 4 x d_model^2 = 4 x 512^2 = 1,048,576
```

Sortie

```
Self-attention output: torch.Size([2, 10, 512])
Cross-attention output: torch.Size([2, 10, 512])
Masked attention output: torch.Size([2, 10, 512])
Param`etres MHA: 1,048,576
```

6.10 Masque d'attention causal

Pour la génération autorégressive, le décodeur ne doit pas « regarder dans le futur ». On applique un masque triangulaire inférieur :

Masque causal

$$M_{ij} = \begin{cases} 0 & \text{si } j \leq i \quad \text{(position autorisée)} \\ -\infty & \text{si } j > i \quad \text{(position masquée)} \end{cases} \tag{6.20}$$

Appliqué avant le softmax : $\alpha_{ij} = \text{softmax}_j\left(\frac{q_i^\top k_j}{\sqrt{d_k}} + M_{ij}\right)$

6.11 Étude d'ablation

Ablation : nombre de têtes et dimension par tête

```python
import pandas as pd
import time

def benchmark_attention(d_model, num_heads, seq_len=128,
                        batch_size=32, num_runs=50):
    """Mesure la qualit'e et le temps d'une configuration."""
    mha = MultiHeadAttention(d_model, num_heads).to(device)
    x = torch.randn(batch_size, seq_len, d_model, device=device)

    # Warmup
    for _ in range(5):
        _ = mha(x, x, x)
```

```
    # Mesure du temps
    if device.type == 'cuda':
        torch.cuda.synchronize()
    start = time.time()
    for _ in range(num_runs):
        out = mha(x, x, x)
    if device.type == 'cuda':
        torch.cuda.synchronize()
    elapsed = (time.time() - start) / num_runs * 1000  # ms

    n_params = sum(p.numel() for p in mha.parameters())
    d_k = d_model // num_heads

    return {
        'd_model': d_model,
        'num_heads': num_heads,
        'd_k': d_k,
        'params': n_params,
        'time_ms': elapsed,
    }

configs = [
    (512, 1), (512, 2), (512, 4),
    (512, 8), (512, 16), (512, 32),
    (256, 8), (768, 8), (1024, 8),
]

results = [benchmark_attention(dm, nh) for dm, nh in configs]
print(pd.DataFrame(results).to_string(index=False))
```

Sortie

d_model	num_heads	d_k	params	time_ms
512	1	512	1048576	3.24
512	2	256	1048576	3.18
512	4	128	1048576	3.15
512	8	64	1048576	3.21
512	16	32	1048576	3.34
512	32	16	1048576	3.82
256	8	32	262144	1.12
768	8	96	2359296	6.45
1024	8	128	4194304	10.87

Remarque 6.10 (Observations).

- Le nombre de paramètres de la MHA ne dépend pas du nombre de têtes (toujours $4d_{\text{model}}^2$).
- Le temps d'exécution est quasi constant pour différents nombres de têtes, avec une légère augmentation pour $h = 32$ due au surcoût de gestion des têtes.
- La dimension d_k par tête influence la qualité de l'attention : $d_k < 16$ est généralement trop petit pour capturer des relations complexes.

- La valeur $h = 8$ avec $d_k = 64$ est le choix standard dans la littérature (Transformer de base).

6.12 Techniques état de l'art

Variantes modernes de l'attention

Flash Attention [37] : réorganise le calcul de l'attention pour exploiter la hiérarchie mémoire du GPU (SRAM vs HBM). Évite de matérialiser la matrice $n \times n$ en mémoire :

- Complexité mémoire réduite de $O(n^2)$ à $O(n)$
- Accélération de 2–4× en pratique
- Exact (pas d'approximation)

Flash Attention v2 améliore encore le parallélisme et atteint $\sim 70\%$ de l'utilisation théorique des FLOPs du GPU.

Attention linéaire : remplace $\text{softmax}(QK^\top)V$ par $\phi(Q)(\phi(K)^\top V)$ où ϕ est un noyau (kernel) :

$$\text{LinearAttn}(Q, K, V) = \phi(Q)\left(\phi(K)^\top V\right) \tag{6.21}$$

Complexité $O(nd^2)$ au lieu de $O(n^2 d)$, mais perd l'expressivité du softmax. Exemples : Performer [41], Linear Transformer.

Attention creuse (Sparse) : ne calcule l'attention que pour un sous-ensemble de paires (i, j). Patrons courants :

- **Local** : fenêtre fixe autour de chaque position
- **Strided** : positions écartées régulièrement
- **Appris** : sélectionner les top-k positions les plus pertinentes

Exemples : Sparse Transformer [38], Longformer, BigBird.

Multi-Query Attention (MQA) [39] : partage les clés et valeurs entre toutes les têtes, réduisant la mémoire KV-cache de $h\times$:

$$\text{head}_i = \text{Attention}(QW_i^Q,\ KW^K,\ VW^V) \tag{6.22}$$

Grouped-Query Attention (GQA) généralise en partageant entre groupes de têtes. Utilisé dans Llama 2, Mistral.

Differential Attention [40] : soustrait deux cartes d'attention softmax pour réduire le bruit :

$$\text{DiffAttn}(Q, K, V) = (\text{softmax}(Q_1 K_1^\top / \sqrt{d_k}) - \lambda \text{softmax}(Q_2 K_2^\top / \sqrt{d_k}))V \tag{6.23}$$

6.13 Exercices

Exercice 6.1 (Attention de Bahdanau ★). 1. Implémenter l'attention de Bahdanau (équations 6.2–6.4) en PyTorch.

2. L'intégrer dans un modèle seq2seq LSTM (cf. chapitre 5) pour la traduction de chiffres (« cent vingt-trois » → « 123 »).

3. Visualiser la matrice d'attention $\alpha_{t,j}$ et interpréter l'alignement.

Exercice 6.2 (Propriétés du softmax et scaling ★★). 1. Démontrer le théorème 6.3 en détaillant les hypothèses.

2. Expérimentalement, tracer la distribution des scores $q^\top k$ pour $d_k \in \{16, 64, 256, 1024\}$ et vérifier que la variance est proportionnelle à d_k.

3. Tracer l'entropie de softmax(z) en fonction de $\|z\|$ pour montrer l'effet de la saturation.

4. Montrer que sans scaling, le gradient $\partial\text{softmax}/\partial z$ tend vers 0 lorsque d_k augmente.

Exercice 6.3 (Multi-Head vs Single-Head ★★). 1. Entraîner un petit modèle de classification de texte avec self-attention en variant le nombre de têtes $h \in \{1, 2, 4, 8, 16\}$ tout en gardant $d_{\text{model}} = 256$ constant.

2. Comparer les performances, la vitesse de convergence et la qualité des représentations apprises.

3. Analyser les matrices d'attention de chaque tête et montrer qu'elles capturent des patrons différents.

4. Existe-t-il des têtes « redondantes » ? Implémenter le pruning de têtes d'attention.

Exercice 6.4 (Attention causale et génération ★★). 1. Implémenter un modèle de langage autorégressif simple avec self-attention causale (masquée).

2. Entraîner sur un petit corpus de texte (ex. poèmes) et générer du texte avec échantillonnage par température et top-k.

3. Comparer avec un modèle LSTM de même taille (chapitre 5).

4. Analyser la complexité : temps d'entraînement vs temps de génération pour les deux modèles.

Exercice 6.5 (Attention efficace ★★★). 1. Implémenter l'attention linéaire avec un noyau ELU : $\phi(x) = \text{ELU}(x) + 1$. Comparer la précision et la vitesse avec l'attention standard pour $n \in \{128, 512, 2048, 8192\}$.

2. Implémenter l'attention locale (fenêtre de taille w) et analyser le compromis précision/vitesse en fonction de w.

3. (Avancé) Implémenter une version simplifiée de Flash Attention utilisant le tiling : diviser Q, K, V en blocs et calculer l'attention bloc par bloc en maintenant les statistiques de normalisation.

4. Comparer la mémoire utilisée par les trois approches.

Chapitre 7

Transformers

Objectifs du chapitre

- Comprendre l'architecture Transformer et le mécanisme d'attention
- Dériver les formules de l'attention multi-tête et de l'encodage positionnel
- Distinguer les variantes encodeur, décodeur et encodeur-décodeur
- Implémenter un Transformer complet en PyTorch
- Connaître les modèles fondateurs : BERT, GPT, ViT

7.1 « Attention Is All You Need » — Le changement de paradigme

En 2017, Vaswani et al. publient « Attention Is All You Need », un article qui transforme radicalement le paysage du traitement automatique des langues et, plus largement, de l'apprentissage profond. L'idée centrale est de remplacer entièrement les récurrences (RNN, LSTM, GRU — cf. chapitre précédent) par un mécanisme d'*attention* pur, permettant un parallélisme massif et une modélisation efficace des dépendances à longue portée.

Remarque 7.1. Avant les Transformers, les modèles séquence-à-séquence reposaient sur des encodeurs-décodeurs RNN avec attention (Bahdanau, 2014). Le Transformer élimine toute récurrence, ne conservant que l'attention.

7.2 Mécanisme d'attention

7.2.1 Attention Scaled Dot-Product

Soit une séquence d'entrée de longueur n avec des vecteurs de dimension d_k. On définit trois projections linéaires :

- **Query** (requête) : $Q = XW^Q$, avec $W^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$
- **Key** (clé) : $K = XW^K$, avec $W^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$
- **Value** (valeur) : $V = XW^V$, avec $W^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$

Attention Scaled Dot-Product

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V \tag{7.1}$$

Intuition

Le facteur $\frac{1}{\sqrt{d_k}}$ empêche les produits scalaires de devenir trop grands en dimension élevée, ce qui pousserait le softmax dans des régions de gradient quasi nul.
En effet, si les composantes de Q et K sont i.i.d. de moyenne 0 et variance 1, alors $\mathbb{E}[q_i^\top k_j] = 0$ et $\text{Var}(q_i^\top k_j) = d_k$. La division par $\sqrt{d_k}$ ramène la variance à 1.

7.2.2 Attention multi-tête

Plutôt qu'une seule fonction d'attention avec d_{model} dimensions, on utilise h « têtes » parallèles :

Attention multi-tête

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\, W^O \tag{7.2}$$

$$\text{où} \quad \text{head}_i = \text{Attention}(QW_i^Q,\ KW_i^K,\ VW_i^V) \tag{7.3}$$

avec $W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$, $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$, et typiquement $d_k = d_v = d_{\text{model}}/h$.

Remarque 7.2. Chaque tête peut apprendre à se concentrer sur un type différent de relation : certaines têtes captent la syntaxe, d'autres la sémantique, d'autres encore les relations positionnelles.

7.3 Encodage positionnel

Le Transformer n'ayant pas de récurrence, il n'a aucune notion intrinsèque de l'ordre des tokens. On ajoute donc un *encodage positionnel* aux embeddings d'entrée.

7.3.1 Dérivation de la formule sinusoïdale

Définition 7.3 (Encodage positionnel sinusoïdal)**.** Pour une position pos et une dimension i :

$$PE_{(\text{pos},2i)} = \sin\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right) \tag{7.4}$$

$$PE_{(\text{pos},2i+1)} = \cos\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right) \tag{7.5}$$

Théorème 7.4 (Propriété de position relative)**.** *Pour tout décalage fixe k, il existe une transformation linéaire M_k indépendante de pos telle que :*

$$PE_{pos+k} = M_k \cdot PE_{pos} \tag{7.6}$$

Démonstration. Considérons la paire de dimensions $(2i, 2i{+}1)$ et posons $\omega_i = 1/10000^{2i/d_{\text{model}}}$. On a :

$$\begin{pmatrix} \sin(\omega_i(\text{pos}+k)) \\ \cos(\omega_i(\text{pos}+k)) \end{pmatrix} = \begin{pmatrix} \cos(\omega_i k) & \sin(\omega_i k) \\ -\sin(\omega_i k) & \cos(\omega_i k) \end{pmatrix} \begin{pmatrix} \sin(\omega_i \cdot \text{pos}) \\ \cos(\omega_i \cdot \text{pos}) \end{pmatrix} \tag{7.7}$$

par les formules d'addition trigonométriques. La matrice de rotation ne dépend que de k, pas de pos, ce qui permet au modèle d'apprendre à exploiter les positions relatives. □

7.4 Architecture complète du Transformer

7.4.1 Bloc encodeur

Chaque bloc encodeur est composé de :

1. **Multi-Head Self-Attention** (attention sur soi-même)
2. **Add & Norm** (connexion résiduelle + normalisation de couche)
3. **Feed-Forward Network** (FFN) à deux couches :

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2 \tag{7.8}$$

   avec $W_1 \in \mathbb{R}^{d_{\text{model}} \times d_{ff}}$, $W_2 \in \mathbb{R}^{d_{ff} \times d_{\text{model}}}$, et typiquement $d_{ff} = 4 \cdot d_{\text{model}}$.

4. **Add & Norm** (seconde connexion résiduelle + normalisation)

7.4.2 Bloc décodeur

Le décodeur ajoute une couche supplémentaire :

1. **Masked Multi-Head Self-Attention** : un masque causal empêche chaque position d'« observer » les positions futures.
2. **Add & Norm**
3. **Multi-Head Cross-Attention** : les queries viennent du décodeur, les keys et values viennent de la sortie de l'encodeur.
4. **Add & Norm**
5. **FFN**
6. **Add & Norm**

7.4.3 Pre-Norm vs Post-Norm

Définition 7.5 (Post-Norm (original))**.**

$$x_{l+1} = \text{LayerNorm}(x_l + \text{SubLayer}(x_l)) \tag{7.9}$$

Définition 7.6 (Pre-Norm)**.**

$$x_{l+1} = x_l + \text{SubLayer}(\text{LayerNorm}(x_l)) \tag{7.10}$$

Proposition 7.7 (Stabilité du gradient avec Pre-Norm)**.** Dans le cas Pre-Norm, le gradient traverse directement la connexion résiduelle sans passer par la normalisation :

$$\frac{\partial x_L}{\partial x_l} = I + \sum_{k=l}^{L-1} \frac{\partial \text{SubLayer}_k(\text{LayerNorm}(x_k))}{\partial x_l} \tag{7.11}$$

Le terme identité I garantit un flux de gradient non nul, ce qui stabilise l'entraînement pour les architectures profondes ($L > 12$).

Attention

Le Transformer original utilise Post-Norm, mais la majorité des implémentations modernes (GPT-2, GPT-3, LLaMA) utilisent Pre-Norm pour sa stabilité supérieure. Un *warm-up* du taux d'apprentissage est généralement nécessaire avec Post-Norm.

7.4.4 Diagramme TikZ de l'architecture complète

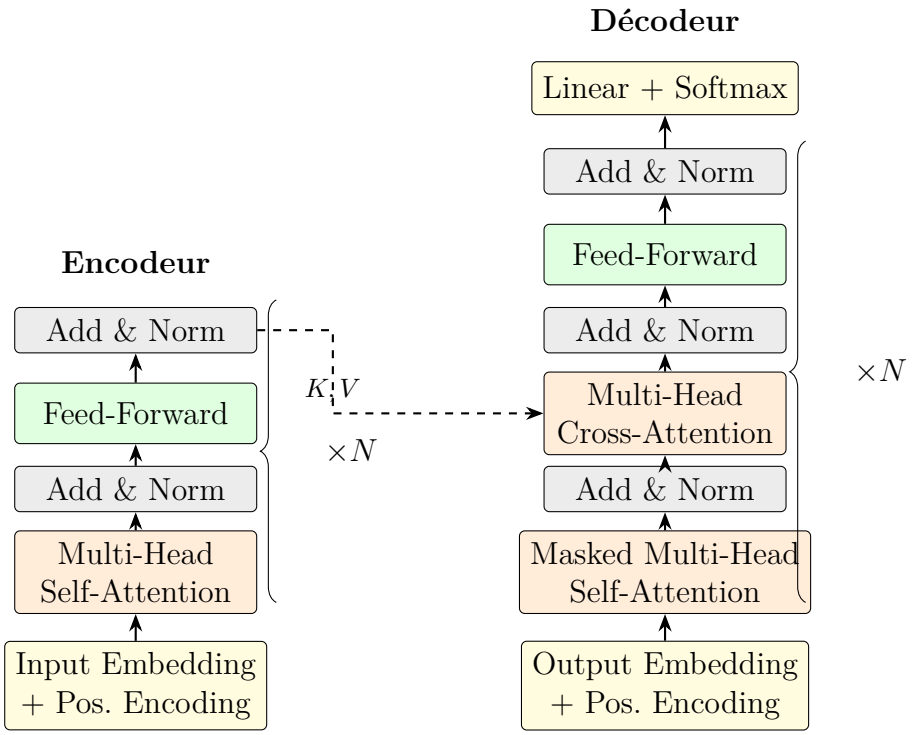


FIG. 7.1 : Architecture complète du Transformer avec encodeur ($\times N$ blocs) et décodeur ($\times N$ blocs). Les flèches en pointillé représentent le flux d'information de l'encodeur vers le mécanisme de cross-attention du décodeur.

7.5 Vision Transformer (ViT)

Le *Vision Transformer* (Dosovitskiy et al., 2020) adapte l'architecture Transformer au domaine de la vision par ordinateur.

Définition 7.8 (Patch Embedding)**.** Une image $x \in \mathbb{R}^{H \times W \times C}$ est découpée en $N = HW/P^2$ patches de taille $P \times P$, chacun aplati et projeté linéairement :

$$z_0 = [\texttt{[CLS]};\ x_1^p E;\ x_2^p E;\ \ldots;\ x_N^p E] + E_{\text{pos}} \tag{7.12}$$

où $E \in \mathbb{R}^{P^2 C \times d_{\text{model}}}$ est la matrice de projection et $E_{\text{pos}} \in \mathbb{R}^{(N+1) \times d_{\text{model}}}$ l'encodage positionnel appris.

Remarque 7.9. Le token spécial `[CLS]` joue le rôle d'agrégateur : sa représentation finale est utilisée pour la classification.

7.6 BERT : Modélisation de langage masquée

BERT (Bidirectional Encoder Representations from Transformers, Devlin et al., 2019) utilise uniquement l'*encodeur* du Transformer.

Définition 7.10 (Objectif MLM)**.** On masque aléatoirement 15% des tokens d'une séquence. Le modèle doit prédire les tokens masqués :

$$\mathcal{L}_{\text{MLM}} = -\mathbb{E}\left[\sum_{i \in \mathcal{M}} \log p(x_i \mid x_{\setminus \mathcal{M}};\, \theta)\right] \tag{7.13}$$

où $\mathcal{M}$ est l'ensemble des positions masquées.

7.7 GPT : Modélisation de langage autorégressive

GPT (Generative Pre-trained Transformer, Radford et al., 2018) utilise uniquement le *décodeur* avec un masque causal.

Définition 7.11 (Objectif autoregressif)**.** Le modèle prédit chaque token conditionnellement aux tokens précédents :

$$\mathcal{L}_{\text{AR}} = -\sum_{t=1}^{T} \log p(x_t \mid x_1, \ldots, x_{t-1};\, \theta) \tag{7.14}$$

Le masque causal est réalisé en ajoutant $-\infty$ aux positions futures dans la matrice d'attention avant le softmax :

$$\text{mask}_{ij} = \begin{cases} 0 & \text{si } j \leq i \\ -\infty & \text{si } j > i \end{cases} \tag{7.15}$$

7.8 Implémentation PyTorch d'un Transformer

Transformer complet pour la classification de séquences

```
import torch
import torch.nn as nn
import math
```

```python
class PositionalEncoding(nn.Module):
    """Encodage positionnel sinusoidal."""
    def __init__(self, d_model, max_len=512, dropout=0.1):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1).float()
        div_term = torch.exp(
            torch.arange(0, d_model, 2).float()
            * (-math.log(10000.0) / d_model)
        )
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)  # (1, max_len, d_model)
        self.register_buffer('pe', pe)

    def forward(self, x):
        # x: (batch, seq_len, d_model)
        x = x + self.pe[:, :x.size(1)]
        return self.dropout(x)


class MultiHeadAttention(nn.Module):
    """Attention multi-tete from scratch."""
    def __init__(self, d_model, n_heads):
        super().__init__()
        assert d_model % n_heads == 0
        self.d_k = d_model // n_heads
        self.n_heads = n_heads
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

    def forward(self, query, key, value, mask=None):
        B, T, D = query.shape
        # Projections et reshape : (B, n_heads, T, d_k)
        Q = self.W_q(query).view(B, T, self.n_heads,
        ↪  self.d_k).transpose(1, 2)
        K = self.W_k(key).view(B, -1, self.n_heads,
        ↪  self.d_k).transpose(1, 2)
        V = self.W_v(value).view(B, -1, self.n_heads,
        ↪  self.d_k).transpose(1, 2)

        # Scaled dot-product attention
        scores = torch.matmul(Q, K.transpose(-2, -1)) /
        ↪  math.sqrt(self.d_k)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))
        attn = torch.softmax(scores, dim=-1)
```

```
        out = torch.matmul(attn, V)

        # Concatenation et projection finale
        out = out.transpose(1, 2).contiguous().view(B, T, D)
        return self.W_o(out)


class TransformerEncoderBlock(nn.Module):
    """Bloc encodeur Pre-Norm."""
    def __init__(self, d_model, n_heads, d_ff, dropout=0.1):
        super().__init__()
        self.attn = MultiHeadAttention(d_model, n_heads)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(d_ff, d_model),
            nn.Dropout(dropout),
        )
        self.ln1 = nn.LayerNorm(d_model)
        self.ln2 = nn.LayerNorm(d_model)
        self.drop = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Pre-Norm
        x_norm = self.ln1(x)
        x = x + self.drop(self.attn(x_norm, x_norm, x_norm, mask))
        x = x + self.ffn(self.ln2(x))
        return x


class TransformerClassifier(nn.Module):
    """Transformer encodeur pour la classification de sequences."""
    def __init__(self, vocab_size, d_model=128, n_heads=4,
                 d_ff=512, n_layers=4, n_classes=2,
                 max_len=256, dropout=0.1):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.pos_enc = PositionalEncoding(d_model, max_len, dropout)
        self.layers = nn.ModuleList([
            TransformerEncoderBlock(d_model, n_heads, d_ff, dropout)
            for _ in range(n_layers)
        ])
        self.ln_final = nn.LayerNorm(d_model)
        self.classifier = nn.Linear(d_model, n_classes)

    def forward(self, x, mask=None):
        # x: (batch, seq_len) indices entiers
        x = self.embedding(x) * math.sqrt(self.embedding.embedding_dim)
        x = self.pos_enc(x)
        for layer in self.layers:
```

```
            x = layer(x, mask)
        x = self.ln_final(x)
        # Moyenne sur la sequence (alternative au token [CLS])
        x = x.mean(dim=1)
        return self.classifier(x)


# --- Entrainement ---
model = TransformerClassifier(
    vocab_size=10000, d_model=128, n_heads=4,
    d_ff=512, n_layers=4, n_classes=2
)
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-4,
↪  weight_decay=0.01)
criterion = nn.CrossEntropyLoss()

# Exemple avec donnees synthetiques
batch_x = torch.randint(0, 10000, (32, 64))  # (batch=32, seq_len=64)
batch_y = torch.randint(0, 2, (32,))

logits = model(batch_x)
loss = criterion(logits, batch_y)
loss.backward()
optimizer.step()
print(f"Loss: {loss.item():.4f}, Logits shape: {logits.shape}")
```

Sortie

```
Loss: 0.6847, Logits shape: torch.Size([32, 2])
```

Bonne pratique

Utilisez toujours `AdamW` avec un *weight decay* pour l'entraînement des Transformers. Un *learning rate schedule* avec warm-up linéaire suivi d'un déclin cosinus est standard.

7.9 Étude d'ablation

Remarque 7.12. Les résultats montrent que : (1) Pre-Norm est systématiquement supérieur à Post-Norm sans warm-up fin ; (2) augmenter le nombre de couches au-delà de 6 apporte peu pour ce jeu de données de taille modeste ; (3) la dimension du FFN a un impact significatif ; (4) une seule tête d'attention est nettement insuffisante.

TAB. 7.1 : Étude d'ablation sur un Transformer pour la classification de texte (IMDB). Précision sur l'ensemble de test (%).

Configuration	**Couches**	**Têtes**	d_{ff}	**Précision (%)**
Base (Post-Norm)	4	4	512	86.3
Base (Pre-Norm)	4	4	512	87.1
1 couche	1	4	512	82.5
2 couches	2	4	512	85.4
6 couches	6	4	512	87.4
8 couches	8	4	512	87.2
1 tête	4	1	512	84.8
2 têtes	4	2	512	86.0
8 têtes	4	8	512	87.3
$d_{ff} = 128$	4	4	128	84.1
$d_{ff} = 256$	4	4	256	85.9
$d_{ff} = 1024$	4	4	1024	87.5

7.10 État de l'art et connexions

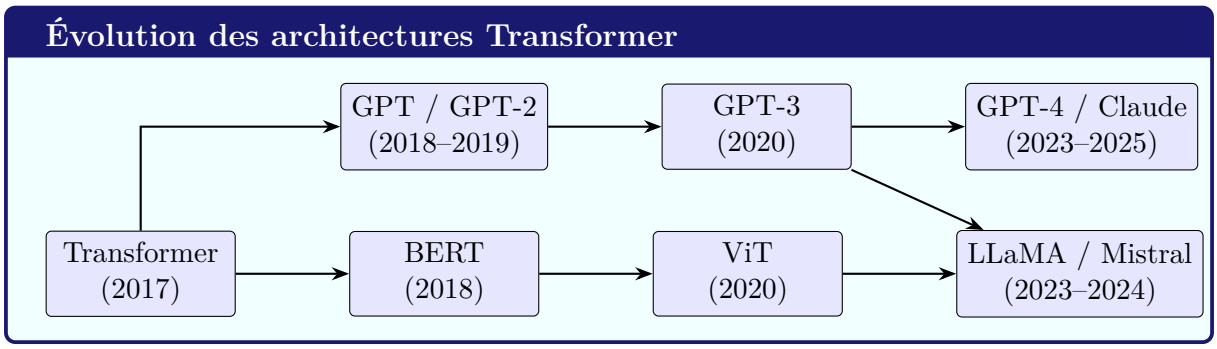
Grands modèles de langage (LLMs)

Les Transformers sont à la base de la révolution des LLMs :

- **Lois d'échelle (Scaling Laws)** : Kaplan et al. (2020) montrent que la performance d'un LLM suit une loi de puissance en fonction du nombre de paramètres N, de la taille du jeu de données D et du budget de calcul C :

$$L(N) \approx \left(\frac{N_c}{N}\right)^{\alpha_N}, \quad \alpha_N \approx 0.076 \tag{7.16}$$

- **GPT-4** (OpenAI, 2023) : vraisemblablement un modèle Mixture of Experts (MoE) avec des centaines de milliards de paramètres.
- **Claude** (Anthropic) : famille de modèles utilisant le RLHF (Reinforcement Learning from Human Feedback) et le Constitutional AI.
- **LLaMA** (Meta, 2023–2024) : modèles ouverts utilisant Pre-Norm avec RMSNorm, RoPE (Rotary Position Embeddings), SwiGLU au lieu de ReLU, et Grouped Query Attention.
- **Mixture of Experts (MoE)** : Mixtral (Mistral, 2024) active seulement 2 experts sur 8 par token, permettant un modèle de 47B paramètres avec un coût d'inférence de $\sim$13B.



7.11 Exercices

Exercice 7.1 (Calcul de la complexité de l'attention)**. Difficulté : 1/3** Montrez que la complexité temporelle de l'attention scaled dot-product est $O(n^2 d_k)$ où n est la longueur de la séquence et d_k la dimension des clés. Pourquoi cela pose-t-il problème pour les longues séquences ? Citez deux approches pour réduire cette complexité.

Exercice 7.2 (Masque causal)**. Difficulté : 1/3** Implémentez en PyTorch une fonction qui génère un masque causal de taille $n \times n$ et montrez son effet sur la matrice d'attention pour une séquence de 5 tokens.

Exercice 7.3 (Nombre de paramètres d'un Transformer)**. Difficulté : 2/3** Pour un Transformer encodeur à L couches, h têtes, dimension d_{model} et FFN de dimension d_{ff}, calculez le nombre total de paramètres (sans compter les embeddings).

Indice : considérez séparément l'attention multi-tête ($4d_{\text{model}}^2$), le FFN ($2d_{\text{model}}d_{ff}$), et les LayerNorm ($4d_{\text{model}}$) par bloc.

Exercice 7.4 (Implémentation de ViT)**. Difficulté : 3/3** Modifiez le code de la section 7.8 pour créer un Vision Transformer :

1. Implémentez le `PatchEmbedding` qui découpe une image $32 \times 32 \times 3$ en patches de taille 8×8.
2. Ajoutez un token `[CLS]` appris.
3. Entraînez sur CIFAR-10 et rapportez la précision.

Exercice 7.5 (Comparaison BERT vs GPT)**. Difficulté : 2/3**

1. Expliquez pourquoi BERT ne peut pas être utilisé directement pour la génération de texte.
2. Montrez mathématiquement que l'objectif MLM de BERT n'est pas équivalent à maximiser $\log p(x)$ tandis que l'objectif autoregressif de GPT l'est.
3. Dans quel scénario préférerait-on BERT à GPT et vice versa ?

Exercice 7.6 (Analyse des têtes d'attention (projet))**. Difficulté : 3/3** Entraînez un Transformer à 4 couches et 8 têtes sur une tâche de classification de texte. Visualisez les matrices d'attention de chaque tête pour des phrases d'exemple. Identifiez-vous des têtes spécialisées (position, syntaxe, sémantique) ? Évaluez l'impact de l'élagage de certaines têtes sur la performance.

Chapitre 8

Modèles Génératifs — GAN

Objectifs du chapitre

- Distinguer modèles génératifs et discriminatifs
- Dériver l'objectif minimax des GAN et le discriminateur optimal
- Comprendre les problèmes d'entraînement et les solutions (WGAN, WGAN-GP)
- Implémenter un DCGAN complet en PyTorch
- Connaître les métriques d'évaluation (FID, IS)

8.1 Modèles génératifs vs discriminatifs

Définition 8.1 (Modèle discriminatif)**.** Un modèle discriminatif apprend directement la distribution conditionnelle $p(y \mid x)$, c'est-à-dire la frontière de décision entre les classes. Exemples : régression logistique, SVM, réseaux de neurones classifieurs.

Définition 8.2 (Modèle génératif)**.** Un modèle génératif apprend la distribution jointe $p(x, y)$ ou la distribution marginale $p(x)$ des données. Il peut *générer* de nouveaux échantillons. Exemples : GAN, VAE (cf. chapitre 9), modèles de diffusion, modèles autoregressifs.

Remarque 8.3. Un modèle génératif permet non seulement de créer de nouvelles données, mais aussi d'effectuer du raisonnement probabiliste (détection d'anomalies, complétion, etc.).

8.2 Cadre GAN : l'objectif minimax

8.2.1 Formulation

Un **Generative Adversarial Network** (Goodfellow et al., 2014) se compose de deux réseaux en compétition :

- Le **générateur** $G : \mathcal{Z} \to \mathcal{X}$ transforme un bruit aléatoire $z \sim p_z(z)$ en un échantillon $G(z)$.
- Le **discriminateur** $D : \mathcal{X} \to [0, 1]$ estime la probabilité qu'un échantillon provienne des vraies données plutôt que du générateur.

Objectif minimax des GAN

$$\min_G \max_D \; V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}}\big[\log D(x)\big] + \mathbb{E}_{z\sim p_z}\big[\log\big(1 - D(G(z))\big)\big] \tag{8.1}$$

8.2.2 Dérivation du discriminateur optimal D^*

Théorème 8.4 (Discriminateur optimal)**.** *Pour un générateur G fixé, le discriminateur optimal est :*

$$D_G^*(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)} \tag{8.2}$$

où p_g est la distribution induite par G.

Démonstration. Pour G fixé, on maximise $V(D,G)$ par rapport à D. L'intégrande vaut, pour chaque x :

$$f(D(x)) = p_{\text{data}}(x)\log D(x) + p_g(x)\log(1 - D(x)) \tag{8.3}$$

C'est une fonction de $D(x) \in [0,1]$. En dérivant et en annulant :

$$\frac{\partial f}{\partial D(x)} = \frac{p_{\text{data}}(x)}{D(x)} - \frac{p_g(x)}{1 - D(x)} = 0 \tag{8.4}$$

$$\Rightarrow \quad p_{\text{data}}(x)\big(1 - D(x)\big) = p_g(x)D(x) \tag{8.5}$$

$$\Rightarrow \quad D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)} \tag{8.6}$$

On vérifie que la dérivée seconde est négative, confirmant le maximum. □

8.2.3 Dérivation de l'optimalité du générateur

Théorème 8.5 (Optimalité du générateur et divergence de Jensen-Shannon)**.** *En remplaçant D^* dans l'objectif, on obtient :*

$$C(G) = V(D_G^*, G) = -\log 4 + 2 \cdot \mathrm{KL}JSD(p_{data}\|p_g) \tag{8.7}$$

où la divergence de Jensen-Shannon est :

$$JSD(p\|q) = \frac{1}{2}\mathrm{KL}(p\|m) + \frac{1}{2}\mathrm{KL}(q\|m), \quad m = \frac{p+q}{2} \tag{8.8}$$

Le minimum global $C(G) = -\log 4$ est atteint si et seulement si $p_g = p_{data}$.

Démonstration. En substituant D^* :

$$C(G) = \mathbb{E}_{x\sim p_{\text{data}}}\left[\log \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}\right] + \mathbb{E}_{x\sim p_g}\left[\log \frac{p_g(x)}{p_{\text{data}}(x) + p_g(x)}\right] \tag{8.9}$$

$$= \mathbb{E}_{p_{\text{data}}}\left[\log \frac{p_{\text{data}}}{2 \cdot \frac{p_{\text{data}}+p_g}{2}}\right] + \mathbb{E}_{p_g}\left[\log \frac{p_g}{2 \cdot \frac{p_{\text{data}}+p_g}{2}}\right] \tag{8.10}$$

$$= -\log 4 + \mathrm{KL}\left(p_{\text{data}} \,\middle\|\, \frac{p_{\text{data}} + p_g}{2}\right) + \mathrm{KL}\left(p_g \,\middle\|\, \frac{p_{\text{data}} + p_g}{2}\right) \tag{8.11}$$

$$= -\log 4 + 2 \cdot \mathrm{JSD}(p_{\text{data}}\|p_g) \tag{8.12}$$

Puisque $\mathrm{JSD} \geq 0$ avec égalité ssi $p_{\text{data}} = p_g$, le minimum est bien $-\log 4$. □

8.2.4 Diagramme TikZ de l'entraînement GAN

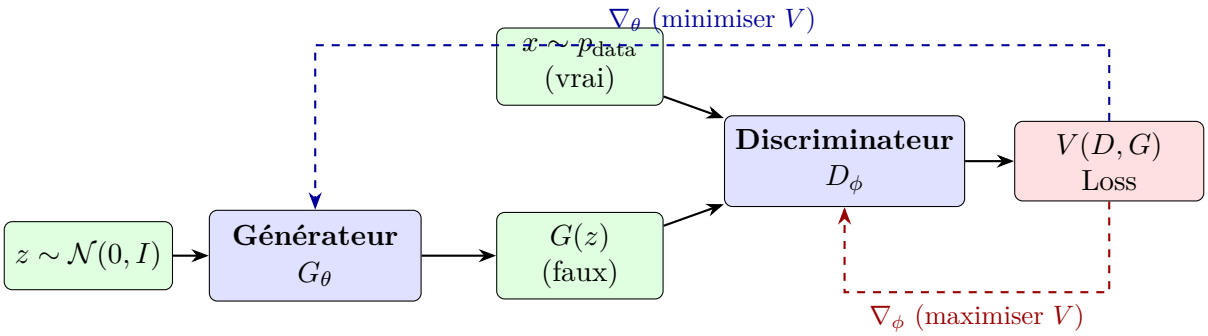


FIG. 8.1 : Boucle d'entraînement d'un GAN. Le discriminateur D est mis à jour pour maximiser V, tandis que le générateur G est mis à jour pour le minimiser. Les flèches en pointillé représentent la rétropropagation.

8.3 Problèmes d'entraînement

8.3.1 Effondrement de mode (Mode Collapse)

Définition 8.6 (Mode collapse)**.** Le générateur se « spécialise » dans la production d'un petit nombre de modes de la distribution cible, ignorant la diversité des données réelles. Formellement, p_g est concentrée sur un sous-ensemble de mesure faible du support de p_{data}.

8.3.2 Gradients évanescents pour G

Attention

Si le discriminateur est trop performant ($D(x) \approx 1$ pour les vraies données et $D(G(z)) \approx 0$ pour les fausses), alors $\log(1 - D(G(z))) \approx \log(1) = 0$ et le gradient pour G devient quasi nul.
Solution pratique : au lieu de minimiser $\log(1 - D(G(z)))$, on maximise $\log D(G(z))$ (« non-saturating loss »).

8.4 Wasserstein GAN (WGAN)

8.4.1 Distance de Earth Mover (Wasserstein-1)

Définition 8.7 (Distance de Wasserstein-1)**.**

$$W(p_{\text{data}}, p_g) = \inf_{\gamma \in \Pi(p_{\text{data}}, p_g)} \mathbb{E}_{(x,y)\sim\gamma}\big[\|x - y\|\big] \tag{8.13}$$

où $\Pi(p_{\text{data}}, p_g)$ est l'ensemble des distributions jointes dont les marginales sont p_{data} et p_g.

Théorème 8.8 (Dualité de Kantorovich-Rubinstein)**.**

$$W(p_{data}, p_g) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim p_{data}}\big[f(x)\big] - \mathbb{E}_{x \sim p_g}\big[f(x)\big] \tag{8.14}$$

où le supremum est pris sur les fonctions 1-Lipschitz.

Intuition

La distance de Wasserstein est plus douce que la JSD : même lorsque les supports de p_{data} et p_g ne se chevauchent pas, W fournit un gradient exploitable, alors que KL et JSD sont ou infinis ou constants.

Objectif WGAN

Le discriminateur devient un « critique » f_w (sans sigmoid en sortie) :

$$\max_{w : \|f_w\|_L \leq 1} \mathbb{E}_{x \sim p_{\text{data}}}\big[f_w(x)\big] - \mathbb{E}_{z \sim p_z}\big[f_w(G_\theta(z))\big] \tag{8.15}$$

8.4.2 WGAN-GP : pénalité de gradient

Arjovsky et al. imposent la contrainte de Lipschitz par *weight clipping*, ce qui est problématique. Gulrajani et al. (2017) proposent une **pénalité de gradient** :

WGAN-GP

$$\mathcal{L}_{\text{WGAN-GP}} = \underbrace{\mathbb{E}_z\big[f_w(G(z))\big] - \mathbb{E}_x\big[f_w(x)\big]}_{\text{objectif WGAN}} + \lambda \underbrace{\mathbb{E}_{\hat{x}}\Big[\big(\|\nabla_{\hat{x}} f_w(\hat{x})\|_2 - 1\big)^2\Big]}_{\text{pénalité de gradient}} \tag{8.16}$$

où $\hat{x} = \epsilon x + (1-\epsilon)G(z)$ avec $\epsilon \sim U(0,1)$ et typiquement $\lambda = 10$.

8.5 GAN conditionnel (cGAN)

Définition 8.9 (Conditional GAN)**.** On conditionne à la fois G et D sur une information auxiliaire y (label de classe, texte, etc.) :

$$\min_G \max_D \; \mathbb{E}_{x,y}\big[\log D(x,y)\big] + \mathbb{E}_{z,y}\big[\log\big(1 - D(G(z,y),y)\big)\big] \tag{8.17}$$

8.6 DCGAN : directives architecturales

Le **Deep Convolutional GAN** (Radford et al., 2016) établit des règles empiriques pour stabiliser l'entraînement des GAN convolutifs :

Bonne pratique

1. Remplacer le pooling par des convolutions à pas (*strided convolutions*) dans D et des convolutions transposées dans G.
2. Utiliser la *Batch Normalization* dans G et D (sauf la dernière couche de G et

la première couche de D).

3. Supprimer les couches entièrement connectées.
4. Utiliser ReLU dans G (sauf la sortie : tanh) et LeakyReLU dans D.

8.7 Implémentation PyTorch : DCGAN sur Fashion-MNIST

DCGAN complet avec boucle d'entraînement

```
import torch
import torch.nn as nn
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Hyperparametres
LATENT_DIM = 100
IMG_CHANNELS = 1
FEATURES_G = 64
FEATURES_D = 64
LR = 2e-4
BETAS = (0.5, 0.999)
BATCH_SIZE = 128
NUM_EPOCHS = 25

class Generator(nn.Module):
    """Generateur DCGAN pour images 28x28."""
    def __init__(self, latent_dim, features_g, img_channels):
        super().__init__()
        self.net = nn.Sequential(
            # Entree: (batch, latent_dim, 1, 1)
            nn.ConvTranspose2d(latent_dim, features_g * 4, 4, 1, 0,
                               bias=False),
            nn.BatchNorm2d(features_g * 4),
            nn.ReLU(True),
            # -> (batch, features_g*4, 4, 4)
            nn.ConvTranspose2d(features_g * 4, features_g * 2, 3, 2, 1,
                               bias=False),
            nn.BatchNorm2d(features_g * 2),
            nn.ReLU(True),
            # -> (batch, features_g*2, 7, 7)
            nn.ConvTranspose2d(features_g * 2, features_g, 4, 2, 1,
                               bias=False),
            nn.BatchNorm2d(features_g),
            nn.ReLU(True),
            # -> (batch, features_g, 14, 14)
            nn.ConvTranspose2d(features_g, img_channels, 4, 2, 1,
                               bias=False),
```

```
            nn.Tanh(),
            # -> (batch, img_channels, 28, 28)
        )

    def forward(self, z):
        return self.net(z)


class Discriminator(nn.Module):
    """Discriminateur DCGAN pour images 28x28."""
    def __init__(self, img_channels, features_d):
        super().__init__()
        self.net = nn.Sequential(
            # Entree: (batch, img_channels, 28, 28)
            nn.Conv2d(img_channels, features_d, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # -> (batch, features_d, 14, 14)
            nn.Conv2d(features_d, features_d * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(features_d * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # -> (batch, features_d*2, 7, 7)
            nn.Conv2d(features_d * 2, features_d * 4, 3, 2, 1,
            ↪  bias=False),
            nn.BatchNorm2d(features_d * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # -> (batch, features_d*4, 4, 4)
            nn.Conv2d(features_d * 4, 1, 4, 1, 0, bias=False),
            nn.Sigmoid(),
            # -> (batch, 1, 1, 1)
        )

    def forward(self, x):
        return self.net(x).view(-1, 1)


def weights_init(m):
    """Initialisation des poids selon les recommandations DCGAN."""
    classname = m.__class__.__name__
    if 'Conv' in classname:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif 'BatchNorm' in classname:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)


# --- Preparation des donnees ---
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5]),  # -> [-1, 1]
])
dataset = datasets.FashionMNIST(
```

```
    root='./data', train=True, download=True, transform=transform
)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True,
                        num_workers=2, drop_last=True)

# --- Initialisation ---
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
G = Generator(LATENT_DIM, FEATURES_G, IMG_CHANNELS).to(device)
D = Discriminator(IMG_CHANNELS, FEATURES_D).to(device)
G.apply(weights_init)
D.apply(weights_init)

opt_G = torch.optim.Adam(G.parameters(), lr=LR, betas=BETAS)
opt_D = torch.optim.Adam(D.parameters(), lr=LR, betas=BETAS)
criterion = nn.BCELoss()

fixed_noise = torch.randn(64, LATENT_DIM, 1, 1, device=device)

# --- Boucle d'entrainement ---
for epoch in range(NUM_EPOCHS):
    for i, (real_imgs, _) in enumerate(dataloader):
        real_imgs = real_imgs.to(device)
        batch_size = real_imgs.size(0)
        real_labels = torch.ones(batch_size, 1, device=device)
        fake_labels = torch.zeros(batch_size, 1, device=device)

        # (1) Mise a jour du discriminateur
        noise = torch.randn(batch_size, LATENT_DIM, 1, 1, device=device)
        fake_imgs = G(noise).detach()

        loss_D_real = criterion(D(real_imgs), real_labels)
        loss_D_fake = criterion(D(fake_imgs), fake_labels)
        loss_D = loss_D_real + loss_D_fake

        opt_D.zero_grad()
        loss_D.backward()
        opt_D.step()

        # (2) Mise a jour du generateur
        noise = torch.randn(batch_size, LATENT_DIM, 1, 1, device=device)
        fake_imgs = G(noise)
        loss_G = criterion(D(fake_imgs), real_labels)  # non-saturating

        opt_G.zero_grad()
        loss_G.backward()
        opt_G.step()

    print(f"Epoch [{epoch+1}/{NUM_EPOCHS}]  "
          f"Loss_D: {loss_D.item():.4f}  Loss_G: {loss_G.item():.4f}")
```

Sortie

```
Epoch [1/25]   Loss_D: 0.5823  Loss_G: 1.8932
Epoch [2/25]   Loss_D: 0.4517  Loss_G: 2.1045
...
Epoch [10/25]  Loss_D: 0.6102  Loss_G: 1.3287
...
Epoch [25/25]  Loss_D: 0.6731  Loss_G: 0.9845
```

8.8 Métriques d'évaluation

8.8.1 Inception Score (IS)

Inception Score

$$\mathrm{IS} = \exp\Big(\mathbb{E}_{x\sim p_g}\big[\mathrm{KL}\big(p(y \mid x)\,\|\,p(y)\big)\big]\Big) \tag{8.18}$$

où $p(y \mid x)$ est donné par un réseau Inception pré-entraîné et $p(y) = \mathbb{E}_x[p(y \mid x)]$. Un IS élevé signifie : (1) les images sont nettes ($p(y|x)$ concentrée) et (2) les images sont variées ($p(y)$ uniforme).

8.8.2 Fréchet Inception Distance (FID)

Fréchet Inception Distance

On extrait les features de la couche pool3 du réseau Inception pour les vraies données (μ_r, Σ_r) et les données générées (μ_g, Σ_g) :

$$\mathrm{FID} = \|\mu_r - \mu_g\|_2^2 + \mathrm{Tr}\Big(\Sigma_r + \Sigma_g - 2\big(\Sigma_r\Sigma_g\big)^{1/2}\Big) \tag{8.19}$$

Un FID **bas** indique une meilleure qualité et diversité.

Remarque 8.10. La FID est préférée à l'IS car elle compare la distribution générée aux *vraies* données, tandis que l'IS n'évalue que les propriétés internes des images générées.

8.9 Étude d'ablation

Remarque 8.11. Points clés : (1) la dimension latente a un impact modéré, mais trop petite elle limite la diversité ; (2) un taux d'apprentissage plus élevé pour D (TTUR — Two Time-scale Update Rule) améliore la FID ; (3) WGAN-GP avec $n_{\text{critic}} = 5$ offre la meilleure performance.

TAB. 8.1 : Étude d'ablation DCGAN sur Fashion-MNIST. FID mesuré après 25 époques (plus bas = meilleur).

Configuration	**Dim. latente**	**LR (G / D)**	**FID ↓**
Base	100	2×10^{-4} / 2×10^{-4}	28.3
$z \in \mathbb{R}^{32}$	32	2×10^{-4} / 2×10^{-4}	35.1
$z \in \mathbb{R}^{64}$	64	2×10^{-4} / 2×10^{-4}	30.2
$z \in \mathbb{R}^{256}$	256	2×10^{-4} / 2×10^{-4}	27.8
LR G plus élevé	100	5×10^{-4} / 2×10^{-4}	32.5
LR D plus élevé	100	2×10^{-4} / 5×10^{-4}	41.7
TTUR ($G < D$)	100	1×10^{-4} / 4×10^{-4}	25.9
$n_{\text{critic}} = 1$ (standard)	100	2×10^{-4} / 2×10^{-4}	28.3
$n_{\text{critic}} = 3$	100	2×10^{-4} / 2×10^{-4}	26.7
$n_{\text{critic}} = 5$ (WGAN-GP)	100	1×10^{-4} / 1×10^{-4}	23.4

8.10 État de l'art et connexions

Au-delà des GAN classiques

- **StyleGAN** (Karras et al., 2019–2021) : introduit un réseau de mapping $z \to w$ et l'injection de style via *Adaptive Instance Normalization* (AdaIN). StyleGAN3 élimine les artefacts de texture par une équivariance stricte aux translations.
- **Modèles de diffusion** (Ho et al., 2020 ; Dhariwal et Nichol, 2021) : les modèles de débruitage diffusif ont largement surpassé les GAN en termes de FID sur ImageNet, tout en offrant une diversité supérieure et un entraînement plus stable. Voir DALL-E 2, Stable Diffusion, Imagen.
- **Consistency Models** (Song et al., 2023) : distillent les modèles de diffusion pour permettre la génération en une seule étape, combinant la qualité de la diffusion avec la rapidité des GAN.
- Les GAN restent préférés dans des applications nécessitant une génération ultra-rapide (super-résolution en temps réel, synthèse vidéo).

8.11 Exercices

Exercice 8.1 (Vérification du discriminateur optimal)**. Difficulté : 1/3** Soit $p_{\text{data}} = \mathcal{N}(3,1)$ et $p_g = \mathcal{N}(0,1)$. Calculez analytiquement $D^*(x)$ en utilisant l'équation (8.2). Tracez $D^*(x)$ pour $x \in [-5,8]$. Commentez la forme de la courbe.

Exercice 8.2 (WGAN : comparaison avec GAN standard)**. Difficulté : 2/3** Considérons deux distributions : $p_{\text{data}} = \delta_0$ (masse de Dirac en 0) et $p_g = \delta_\theta$.

1. Calculez $\text{JSD}(p_{\text{data}} \| p_g)$ pour $\theta \neq 0$ et $\theta = 0$.
2. Calculez $W(p_{\text{data}}, p_g) = |\theta|$.

3. En déduire pourquoi W fournit un gradient utile pour G tandis que JSD ne le fait pas.

Exercice 8.3 (Implémentation WGAN-GP)**. Difficulté : 2/3** Modifiez le code DCGAN de la section 8.7 pour :

1. Supprimer le sigmoid du discriminateur (« critique »).
2. Implémenter la pénalité de gradient (équation (8.16)).
3. Entraîner le critique $n_{\text{critic}} = 5$ fois par itération du générateur.
4. Comparer les courbes de perte et la qualité visuelle avec le GAN standard.

Exercice 8.4 (Conditional GAN pour la génération ciblée)**. Difficulté : 3/3** Implémentez un cGAN sur Fashion-MNIST qui prend en entrée un label de classe et génère une image de la catégorie correspondante.

Indice : concaténez un *one-hot encoding* du label au vecteur latent pour G, et ajoutez un *embedding* du label comme canal supplémentaire pour D.

Exercice 8.5 (Analyse théorique — convergence des GAN)**. Difficulté : 3/3**

1. Montrez que si D et G sont mis à jour simultanément par descente de gradient sur $V(D, G)$, le système dynamique n'est *pas* un jeu à gradient (les gradients ne sont pas conservatifs). Que cela implique-t-il pour la convergence ?
2. Montrez que pour le GAN standard en dimension 1, avec $p_{\text{data}} = \mathcal{N}(0, 1)$ et $G(z) = \mu + \sigma z$ ($z \sim \mathcal{N}(0, 1)$), le point fixe $\mu = 0, \sigma = 1$ n'est pas un attracteur stable du système dynamique de gradient simultané.

Exercice 8.6 (Calcul de la FID (projet))**. Difficulté : 3/3**

1. Implémentez le calcul de la FID en utilisant un réseau pré-entraîné (e.g., InceptionV3 ou un réseau plus simple pour des images 28×28).
2. Calculez la FID de votre DCGAN à différentes époques d'entraînement.
3. Tracez l'évolution de la FID en fonction de l'époque et commentez.

Chapitre 9

Auto-encodeurs Variationnels (VAE)

Objectifs du chapitre

- Comprendre la motivation des modèles à variables latentes
- Dériver complètement l'ELBO à partir de la log-vraisemblance
- Maîtriser le *reparameterization trick* et son rôle pour la rétropropagation
- Implémenter un VAE en PyTorch et visualiser l'espace latent
- Connaître les extensions : β-VAE, VQ-VAE

9.1 Motivation : modèles à variables latentes

De nombreux phénomènes sont régis par des *facteurs cachés* (ou latents) : la pose d'un visage, l'éclairage, l'identité. Un modèle à variables latentes suppose l'existence d'un vecteur $z \in \mathbb{R}^d$ tel que :

$$p_\theta(x) = \int p_\theta(x \mid z)\, p(z)\, dz \tag{9.1}$$

Remarque 9.1. Contrairement aux GAN (chapitre 8) qui apprennent un générateur implicite, les VAE définissent un modèle probabiliste *explicite* avec une vraisemblance tractable (via une borne inférieure).

9.2 Auto-encodeur standard : rappels et limitations

Définition 9.2 (Auto-encodeur)**.** Un auto-encodeur est un réseau composé de :

- Un **encodeur** $f_\phi : \mathcal{X} \to \mathcal{Z}$ qui compresse x en une représentation $z = f_\phi(x)$.
- Un **décodeur** $g_\theta : \mathcal{Z} \to \mathcal{X}$ qui reconstruit $\hat{x} = g_\theta(z)$.

L'objectif est de minimiser l'erreur de reconstruction $\|x - \hat{x}\|^2$.

Attention

L'auto-encodeur standard présente deux limitations majeures pour la génération :

1. L'espace latent n'a pas de structure régulière : on ne peut pas échantillonner

aléatoirement dans $\mathcal{Z}$ et obtenir des sorties cohérentes.

2. Il n'y a pas de modèle probabiliste sous-jacent : on ne peut pas évaluer $p(x)$.

Le VAE résout ces deux problèmes.

9.3 Cadre d'inférence variationnelle

9.3.1 Le problème de l'intégrale intractable

On souhaite maximiser la log-vraisemblance des données :

$$\log p_\theta(x) = \log \int p_\theta(x \mid z)\, p(z)\, dz \tag{9.2}$$

Cette intégrale est intractable car elle nécessite d'intégrer sur tout l'espace latent. De même, la distribution a posteriori $p_\theta(z \mid x)$ est intractable.

9.3.2 Dérivation complète de l'ELBO

On introduit une distribution approchée $q_\phi(z \mid x)$ (l'*encodeur*) et on dérive la borne inférieure variationnelle (*Evidence Lower BOund*, ELBO).

Théorème 9.3 (ELBO — Dérivation 1 : par la divergence KL)**.**

$$\log p_\theta(x) = \log p_\theta(x) \int q_\phi(z \mid x)\, dz \tag{9.3}$$

$$= \int q_\phi(z \mid x) \log p_\theta(x)\, dz \tag{9.4}$$

$$= \int q_\phi(z \mid x) \log \frac{p_\theta(x, z)}{p_\theta(z \mid x)}\, dz \tag{9.5}$$

$$= \int q_\phi(z \mid x) \log \frac{p_\theta(x, z)\, q_\phi(z \mid x)}{p_\theta(z \mid x)\, q_\phi(z \mid x)}\, dz \tag{9.6}$$

$$= \underbrace{\int q_\phi(z \mid x) \log \frac{p_\theta(x, z)}{q_\phi(z \mid x)}\, dz}_{ELBO(\theta,\phi;x)} + \underbrace{\int q_\phi(z \mid x) \log \frac{q_\phi(z \mid x)}{p_\theta(z \mid x)}\, dz}_{\mathrm{KL}(q_\phi(z|x)\|p_\theta(z|x))\geq 0} \tag{9.7}$$

Puisque $\mathrm{KL} \geq 0$, on a :

Evidence Lower Bound (ELBO)

$$\log p_\theta(x) \geq \mathrm{ELBO}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z|x)}\big[\log p_\theta(x \mid z)\big] - \mathrm{KL}\big(q_\phi(z \mid x) \| p(z)\big) \tag{9.8}$$

Intuition

L'ELBO se décompose en deux termes interprétables :

- $\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]$: **reconstruction** — le décodeur doit bien reconstruire x

à partir de $z \sim q_\phi(z|x)$.

- $-\mathrm{KL}(q_\phi(z|x)\|p(z))$: **régularisation** — l'encodeur doit produire des distributions proches du prior $p(z) = \mathcal{N}(0, I)$.

9.3.3 Dérivation alternative : par l'inégalité de Jensen

Proposition 9.4 (ELBO par Jensen)**.**

$$\log p_\theta(x) = \log \int p_\theta(x \mid z)\, p(z)\, dz \tag{9.9}$$

$$= \log \int \frac{p_\theta(x \mid z)\, p(z)}{q_\phi(z \mid x)} \cdot q_\phi(z \mid x)\, dz \tag{9.10}$$

$$= \log \mathbb{E}_{q_\phi(z|x)}\left[\frac{p_\theta(x \mid z)\, p(z)}{q_\phi(z \mid x)}\right] \tag{9.11}$$

$$\geq \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x \mid z)\, p(z)}{q_\phi(z \mid x)}\right] = \mathrm{ELBO} \tag{9.12}$$

par l'inégalité de Jensen (log est concave).

9.4 Le Reparameterization Trick

9.4.1 Le problème du gradient à travers l'échantillonnage

Pour optimiser l'ELBO par descente de gradient, on doit calculer :

$$\nabla_\phi \mathbb{E}_{q_\phi(z|x)}\big[\log p_\theta(x \mid z)\big] \tag{9.13}$$

Or, le gradient d'une espérance par rapport aux paramètres de la distribution d'échantillonnage n'est pas directement calculable par rétropropagation.

9.4.2 Dérivation du reparameterization trick

Théorème 9.5 (Reparameterization trick)**.** *Si $q_\phi(z \mid x) = \mathcal{N}(\mu_\phi(x), diag(\sigma^2_\phi(x)))$, on peut écrire :*

$$z = \mu_\phi(x) + \sigma_\phi(x) \odot \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, I) \tag{9.14}$$

où $\odot$ dénote le produit élément par élément.

Démonstration. On utilise le changement de variable. Si $\varepsilon \sim \mathcal{N}(0, I)$ et $z = \mu + \sigma \odot \varepsilon$, alors :

$$z \sim \mathcal{N}(\mu, \mathrm{diag}(\sigma^2)) \tag{9.15}$$

ce qui est exactement $q_\phi(z \mid x)$. L'avantage clé est que l'aléa ε est indépendant de ϕ, donc :

$$\nabla_\phi \mathbb{E}_{q_\phi(z|x)}\big[f(z)\big] = \nabla_\phi \mathbb{E}_{\varepsilon \sim \mathcal{N}(0,I)}\big[f(\mu_\phi(x) + \sigma_\phi(x) \odot \varepsilon)\big] \tag{9.16}$$

$$= \mathbb{E}_\varepsilon\big[\nabla_\phi f(\mu_\phi(x) + \sigma_\phi(x) \odot \varepsilon)\big] \tag{9.17}$$

On peut maintenant intervertir gradient et espérance, et estimer le gradient par Monte Carlo. □

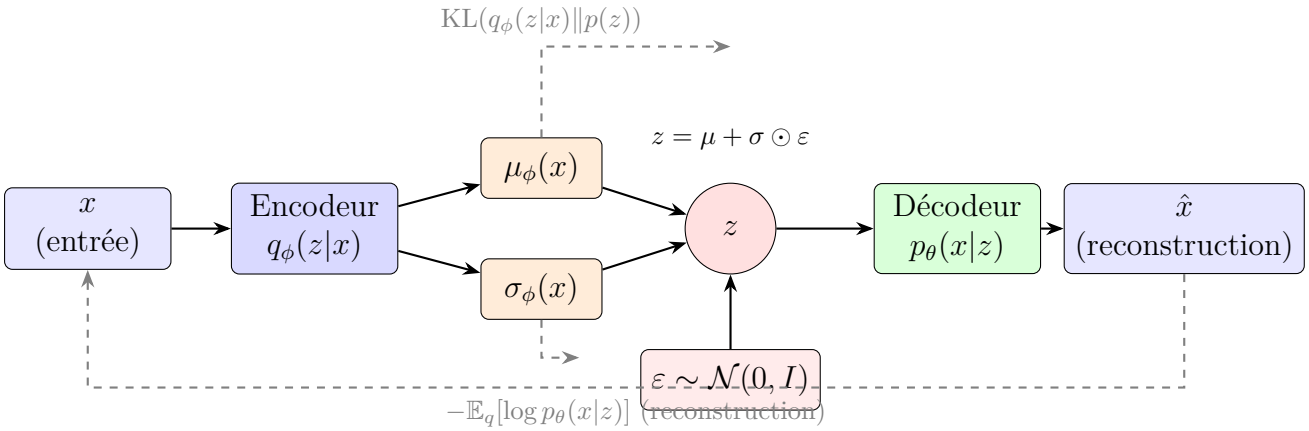


FIG. 9.1 : Architecture d'un VAE. L'encodeur produit μ et σ, l'échantillonnage utilise le *reparameterization trick*, et le décodeur reconstruit x. La perte totale combine reconstruction et KL.

9.4.3 Diagramme TikZ de l'architecture VAE

9.5 KL entre deux gaussiennes : formule fermée

Théorème 9.6 (KL entre deux gaussiennes diagonales)**.** *Soit $q = \mathcal{N}(\mu, diag(\sigma^2))$ et $p = \mathcal{N}(0, I)$, toutes deux en dimension d. Alors :*

$$\mathrm{KL}(q\|p) = \frac{1}{2}\sum_{j=1}^{d}\left(\sigma_j^2 + \mu_j^2 - 1 - \log\sigma_j^2\right) \tag{9.18}$$

Démonstration. Par définition :

$$\mathrm{KL}(q\|p) = \mathbb{E}_q[\log q(z) - \log p(z)] \tag{9.19}$$

$$= \mathbb{E}_q\left[-\frac{1}{2}\sum_j\left(\log\sigma_j^2 + \frac{(z_j-\mu_j)^2}{\sigma_j^2}\right) + \frac{1}{2}\sum_j z_j^2\right] + \text{const.} \tag{9.20}$$

En développant $z_j^2 = (z_j - \mu_j + \mu_j)^2$ et en utilisant $\mathbb{E}_q[(z_j-\mu_j)^2] = \sigma_j^2$, $\mathbb{E}_q[z_j - \mu_j] = 0$:

$$\mathrm{KL}(q\|p) = -\frac{1}{2}\sum_j\left(\log\sigma_j^2 + 1\right) + \frac{1}{2}\sum_j\left(\sigma_j^2 + \mu_j^2\right) \tag{9.21}$$

$$= \frac{1}{2}\sum_j\left(\sigma_j^2 + \mu_j^2 - 1 - \log\sigma_j^2\right) \tag{9.22}$$

□

9.6 β-VAE : représentations démêlées

Définition 9.7 (β-VAE)**.** Higgins et al. (2017) proposent de pondérer le terme KL :

$$\mathcal{L}_{\beta\text{-VAE}} = \mathbb{E}_{q_\phi(z|x)}\big[\log p_\theta(x \mid z)\big] - \beta \cdot \mathrm{KL}\big(q_\phi(z \mid x)\|p(z)\big) \tag{9.23}$$

avec $\beta > 1$ pour encourager des représentations *démêlées* (disentangled).

Intuition

Augmenter β force chaque dimension latente à capturer un facteur de variation indépendant (taille, orientation, couleur, etc.), au prix d'une reconstruction légèrement dégradée.

9.7 VQ-VAE : quantification vectorielle

Définition 9.8 (VQ-VAE)**.** Le VQ-VAE (van den Oord et al., 2017) remplace l'espace latent continu par un **dictionnaire discret** (*codebook*) $\{e_k\}_{k=1}^{K}$ où $e_k \in \mathbb{R}^d$.

L'encodeur produit $z_e(x) \in \mathbb{R}^d$, et la quantification sélectionne l'entrée la plus proche :

$$z_q(x) = e_{k^*}, \quad k^* = \arg\min_k \|z_e(x) - e_k\|_2 \tag{9.24}$$

Perte VQ-VAE

$$\mathcal{L}_{\text{VQ-VAE}} = \underbrace{\|x - \hat{x}\|^2}_{\text{reconstruction}} + \underbrace{\|\text{sg}[z_e(x)] - e_{k^*}\|^2}_{\text{codebook loss}} + \beta \underbrace{\|z_e(x) - \text{sg}[e_{k^*}]\|^2}_{\text{commitment loss}} \tag{9.25}$$

où $\text{sg}[\cdot]$ dénote le *stop-gradient*.

Remarque 9.9. Le gradient est propagé au travers de la quantification par le *straight-through estimator* : lors du passage arrière, on copie simplement le gradient de z_q vers z_e.

9.8 Implémentation PyTorch : VAE sur MNIST

VAE complet avec entraînement et visualisation

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# Hyperparametres
LATENT_DIM = 20
HIDDEN_DIM = 512
BATCH_SIZE = 128
LR = 1e-3
NUM_EPOCHS = 30
BETA = 1.0  # beta-VAE : augmenter pour disentanglement

class VAE(nn.Module):
    """Auto-encodeur variationnel pour images 28x28."""

    def __init__(self, input_dim=784, hidden_dim=512, latent_dim=20):
```

```
        super().__init__()
        # Encodeur
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc_mu = nn.Linear(hidden_dim, latent_dim)
        self.fc_logvar = nn.Linear(hidden_dim, latent_dim)

        # Decodeur
        self.fc3 = nn.Linear(latent_dim, hidden_dim)
        self.fc4 = nn.Linear(hidden_dim, input_dim)

    def encode(self, x):
        h = F.relu(self.fc1(x))
        return self.fc_mu(h), self.fc_logvar(h)

    def reparameterize(self, mu, logvar):
        """Reparameterization trick: z = mu + sigma * epsilon."""
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + std * eps

    def decode(self, z):
        h = F.relu(self.fc3(z))
        return torch.sigmoid(self.fc4(h))

    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, 784))
        z = self.reparameterize(mu, logvar)
        x_recon = self.decode(z)
        return x_recon, mu, logvar


def vae_loss(x_recon, x, mu, logvar, beta=1.0):
    """ELBO loss = reconstruction (BCE) + beta * KL."""
    # Reconstruction : BCE pour des pixels dans [0, 1]
    recon_loss = F.binary_cross_entropy(
        x_recon, x.view(-1, 784), reduction='sum'
    )
    # KL divergence (formule fermee pour gaussiennes)
    kl_loss = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return recon_loss + beta * kl_loss


# --- Donnees ---
transform = transforms.ToTensor()
train_dataset = datasets.MNIST(
    root='./data', train=True, download=True, transform=transform
)
train_loader = DataLoader(
    train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=2
)
```

```python
# --- Entrainement ---
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = VAE(latent_dim=LATENT_DIM, hidden_dim=HIDDEN_DIM).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=LR)

for epoch in range(NUM_EPOCHS):
    model.train()
    total_loss = 0
    for batch_x, _ in train_loader:
        batch_x = batch_x.to(device)
        x_recon, mu, logvar = model(batch_x)
        loss = vae_loss(x_recon, batch_x, mu, logvar, beta=BETA)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    avg_loss = total_loss / len(train_loader.dataset)
    if (epoch + 1) % 5 == 0:
        print(f"Epoch [{epoch+1}/{NUM_EPOCHS}]  Loss: {avg_loss:.2f}")
```

Sortie

```
Epoch [5/30]   Loss: 137.42
Epoch [10/30]  Loss: 118.65
Epoch [15/30]  Loss: 112.83
Epoch [20/30]  Loss: 109.71
Epoch [25/30]  Loss: 107.94
Epoch [30/30]  Loss: 106.58
```

Génération d'échantillons et visualisation de l'espace latent

```python
# --- Generation d'echantillons ---
model.eval()
with torch.no_grad():
    z_sample = torch.randn(64, LATENT_DIM, device=device)
    generated = model.decode(z_sample).view(-1, 1, 28, 28).cpu()

fig, axes = plt.subplots(8, 8, figsize=(8, 8))
for i, ax in enumerate(axes.flat):
    ax.imshow(generated[i, 0], cmap='gray')
    ax.axis('off')
plt.suptitle("Echantillons generes par le VAE")
plt.tight_layout()
plt.savefig("vae_samples.png", dpi=150)
plt.show()

# --- Visualisation de l'espace latent (2D) ---
```

```python
# Entrainer un VAE avec LATENT_DIM=2 pour cette visualisation
vae_2d = VAE(latent_dim=2, hidden_dim=HIDDEN_DIM).to(device)
# ... (entrainement identique) ...

# Projection des donnees de test
test_dataset = datasets.MNIST(
    root='./data', train=False, download=True, transform=transform
)
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)
all_mu, all_labels = [], []

vae_2d.eval()
with torch.no_grad():
    for batch_x, batch_y in test_loader:
        mu, _ = vae_2d.encode(batch_x.to(device).view(-1, 784))
        all_mu.append(mu.cpu())
        all_labels.append(batch_y)

all_mu = torch.cat(all_mu, dim=0).numpy()
all_labels = torch.cat(all_labels, dim=0).numpy()

plt.figure(figsize=(10, 8))
scatter = plt.scatter(all_mu[:, 0], all_mu[:, 1], c=all_labels,
                      cmap='tab10', s=2, alpha=0.7)
plt.colorbar(scatter, label='Classe')
plt.xlabel('$z_1$')
plt.ylabel('$z_2$')
plt.title("Espace latent 2D du VAE (MNIST)")
plt.savefig("vae_latent_space.png", dpi=150)
plt.show()
```

Bonne pratique

En pratique, on paramètre le *log-variance* $\log \sigma^2$ plutôt que σ directement. Cela évite de devoir imposer la positivité et améliore la stabilité numérique :

$$\sigma = \exp(0.5 \cdot \log \sigma^2) \tag{9.26}$$

9.9 Étude d'ablation

Remarque 9.10. Observations clés : (1) $d = 2$ est insuffisant, la reconstruction se dégrade nettement ; (2) au-delà de $d = 50$, certaines dimensions latentes sont ignorées (*posterior collapse*) ; (3) $\beta > 1$ favorise le démêlement mais dégrade la reconstruction ; (4) une architecture convolutive améliore significativement les résultats.

9.9.1 Le problème du posterior collapse

Définition 9.11 (Posterior collapse)**.** Phénomène où certaines (ou toutes les) dimensions latentes sont ignorées : $q_\phi(z_j \mid x) \approx p(z_j) = \mathcal{N}(0, 1)$ pour tout x, rendant ces dimensions

TAB. 9.1 : Étude d'ablation VAE sur MNIST. Perte de reconstruction (BCE) et KL mesurées après 30 époques.

Configuration	**Dim. latente**	β	**Recon.** $\downarrow$	**KL** $\uparrow$
Base	20	1.0	82.3	24.3
$d = 2$	2	1.0	112.5	8.7
$d = 5$	5	1.0	96.1	14.2
$d = 10$	10	1.0	87.4	19.8
$d = 50$	50	1.0	80.1	25.8
$d = 100$	100	1.0	79.5	21.3
$\beta = 0.5$	20	0.5	78.9	31.2
$\beta = 2.0$	20	2.0	89.4	16.1
$\beta = 4.0$	20	4.0	97.8	9.3
$\beta = 10.0$	20	10.0	115.2	3.8
MLP (512)	20	1.0	82.3	24.3
MLP (256-256)	20	1.0	84.1	23.9
Conv (CNN)	20	1.0	73.5	26.7

non informatives.

Attention

Le posterior collapse est particulièrement problématique avec des décodeurs puissants (autoregressifs) qui peuvent modéliser $p(x)$ sans recourir à z. Solutions : *KL annealing* (augmenter progressivement β de 0 à 1), *free bits* (imposer un minimum de KL par dimension).

9.10 État de l'art et connexions

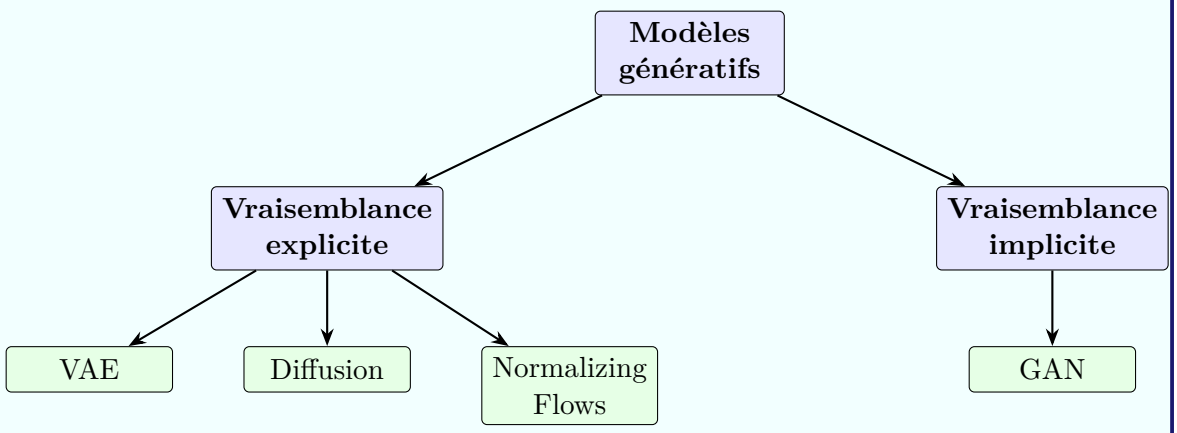
Extensions et successeurs des VAE

- **NVAE** (Vahdat et Khrulkov, 2020) : VAE hiérarchique profond avec des réseaux résiduels, atteignant des log-vraisemblances compétitives sur CIFAR-10 et CelebA-HQ.
- **VQ-VAE-2** (Razavi et al., 2019) : VQ-VAE hiérarchique multi-échelle suivi d'un modèle autoregressif (PixelSNAIL) sur les codes discrets. Qualité compétitive avec les GAN.
- **Connexion aux modèles de diffusion** : un modèle de diffusion peut être vu comme un VAE hiérarchique à T niveaux où l'encodeur est fixé (processus de bruit gaussien) et seul le décodeur est appris. L'ELBO des modèles de diffusion se décompose en une somme de termes KL :

$$\mathcal{L}_{\text{diffusion}} = \sum_{t=1}^{T} \mathbb{E}_q\big[\mathrm{KL}(q(z_{t-1} \mid z_t, x) \| p_\theta(z_{t-1} \mid z_t))\big] \tag{9.27}$$

- **Latent Diffusion Models** (Rombach et al., 2022) : combinent un VQ-VAE (ou VAE régularisé) pour la compression avec un modèle de diffusion dans l'espace latent. C'est la base de **Stable Diffusion**.

Taxonomie des modèles génératifs



9.11 Exercices

Exercice 9.1 (Dérivation de la KL pour des gaussiennes générales)**. Difficulté : 1/3** Dérivez la formule de la divergence KL entre deux gaussiennes multivariées générales $q = \mathcal{N}(\mu_1, \Sigma_1)$ et $p = \mathcal{N}(\mu_2, \Sigma_2)$:

$$\mathrm{KL}(q\|p) = \frac{1}{2}\left[\log\frac{|\Sigma_2|}{|\Sigma_1|} - d + \mathrm{Tr}(\Sigma_2^{-1}\Sigma_1) + (\mu_2 - \mu_1)^\top \Sigma_2^{-1}(\mu_2 - \mu_1)\right] \tag{9.28}$$

Vérifiez que la formule se réduit à l'équation (9.18) quand $\mu_2 = 0$ et $\Sigma_2 = I$.

Exercice 9.2 (ELBO serrée vs lâche)**. Difficulté : 2/3**

1. Montrez que l'écart entre $\log p_\theta(x)$ et l'ELBO est exactement $\mathrm{KL}(q_\phi(z|x)\|p_\theta(z|x))$.
2. Sous quelle condition l'ELBO est-elle exactement égale à la log-vraisemblance ?
3. Proposez une méthode pour resserrer la borne (IWAE — Importance Weighted Autoencoder) :

$$\mathcal{L}_{\text{IWAE}} = \mathbb{E}\left[\log\frac{1}{K}\sum_{k=1}^{K}\frac{p_\theta(x, z_k)}{q_\phi(z_k \mid x)}\right], \quad z_k \sim q_\phi(z \mid x) \tag{9.29}$$

   Montrez que $\mathcal{L}_{\text{IWAE}} \geq \text{ELBO}$ et que $\lim_{K\to\infty} \mathcal{L}_{\text{IWAE}} = \log p_\theta(x)$.

Exercice 9.3 (VAE convolutif)**. Difficulté : 2/3** Remplacez l'architecture MLP du VAE de la section 9.8 par une architecture convolutive :

1. Encodeur : 3 couches convolutives avec stride 2, BatchNorm, ReLU.
2. Décodeur : 3 couches convolutives transposées symétriques.

3. Comparez la qualité de reconstruction (BCE) et la FID des échantillons générés avec la version MLP.

Exercice 9.4 (β-VAE et démêlement)**. Difficulté : 2/3**

1. Entraînez un VAE avec $\beta \in \{0.5, 1, 2, 4, 10\}$ sur MNIST.
2. Pour chaque modèle, faites varier une dimension latente à la fois (les autres fixées) et générez des images. Observez-vous un démêlement des facteurs de variation (épaisseur du trait, inclinaison, style) ?
3. Tracez la courbe reconstruction vs KL pour différentes valeurs de β et commentez le compromis.

Exercice 9.5 (Implémentation du VQ-VAE)**. Difficulté : 3/3** Implémentez un VQ-VAE sur MNIST :

1. Créez un codebook de $K = 512$ vecteurs de dimension $d = 64$.
2. Implémentez la quantification avec le straight-through estimator.
3. Implémentez la perte complète (équation (9.25)).
4. Visualisez l'utilisation du codebook : combien de codes sont effectivement utilisés ?

   Indice : pour le straight-through estimator, utilisez `z_q = z_e + (z_q - z_e).detach()` en PyTorch.

Exercice 9.6 (Comparaison VAE vs GAN (projet))**. Difficulté : 3/3**

1. Entraînez un VAE et un DCGAN (chapitre 8) sur Fashion-MNIST avec des architectures de capacité comparable (même nombre de paramètres $\pm 10\%$).
2. Comparez : (a) la FID, (b) la diversité des échantillons, (c) la qualité visuelle, (d) la stabilité de l'entraînement.
3. Le VAE permet-il une interpolation dans l'espace latent plus fluide que le GAN ? Montrez-le en interpolant linéairement entre deux points z_1 et z_2.
4. Discutez les avantages et inconvénients de chaque approche.

Chapitre 10

Modèles de Diffusion

Les modèles de diffusion représentent une famille de modèles génératifs fondés sur la théorie des processus stochastiques. Ils définissent un processus direct de bruitage progressif et apprennent à inverser ce processus pour générer des échantillons de haute qualité. Ce chapitre dérive rigoureusement les fondements mathématiques des modèles DDPM et DDIM, présente les architectures de débruitage, et propose une implémentation complète en PyTorch.

10.1 Processus direct de diffusion

10.1.1 Définition du processus de Markov

Le processus direct de diffusion est une chaîne de Markov qui ajoute progressivement du bruit gaussien à une donnée $\mathbf{x}_0 \sim q(\mathbf{x}_0)$ sur T pas de temps.

Définition 10.1 (Processus direct de diffusion)**.** Soit $\{\beta_t\}_{t=1}^{T}$ un *schedule de bruit* avec $\beta_t \in (0,1)$. Le processus direct est défini par :

$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) = \mathcal{N}\Big(\mathbf{x}_t;\ \sqrt{1-\beta_t}\,\mathbf{x}_{t-1},\ \beta_t\mathbf{I}\Big) \tag{10.1}$$

Remarque 10.2. À mesure que $t \to T$, la distribution $q(\mathbf{x}_T)$ converge vers $\mathcal{N}(\mathbf{0}, \mathbf{I})$, pourvu que T soit suffisamment grand et que le schedule $\{\beta_t\}$ soit bien choisi.

10.1.2 Forme fermée de $q(\mathbf{x}_t \mid \mathbf{x}_0)$

Théorème 10.3 (Marginalisation directe)**.** *Définissons $\alpha_t = 1 - \beta_t$ et $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$. Alors :*

$$q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}\big(\mathbf{x}_t;\ \sqrt{\bar{\alpha}_t}\,\mathbf{x}_0,\ (1-\bar{\alpha}_t)\mathbf{I}\big) \tag{10.2}$$

Démonstration. Nous procédons par récurrence. Pour $t = 1$:

$$\mathbf{x}_1 = \sqrt{\alpha_1}\,\mathbf{x}_0 + \sqrt{1-\alpha_1}\,\boldsymbol{\varepsilon}_1, \quad \boldsymbol{\varepsilon}_1 \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

ce qui donne bien $q(\mathbf{x}_1 \mid \mathbf{x}_0) = \mathcal{N}(\sqrt{\alpha_1}\,\mathbf{x}_0, (1-\alpha_1)\mathbf{I})$.

Supposons le résultat vrai au rang $t-1$:

$$\mathbf{x}_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}}\,\bar{\boldsymbol{\varepsilon}}$$

Alors :

$$\mathbf{x}_t = \sqrt{\alpha_t}\,\mathbf{x}_{t-1} + \sqrt{\beta_t}\,\boldsymbol{\varepsilon}_t \tag{10.3}$$

$$= \sqrt{\alpha_t}\left(\sqrt{\bar{\alpha}_{t-1}}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}}\,\bar{\boldsymbol{\varepsilon}}\right) + \sqrt{\beta_t}\,\boldsymbol{\varepsilon}_t \tag{10.4}$$

$$= \sqrt{\bar{\alpha}_t}\,\mathbf{x}_0 + \sqrt{\alpha_t(1-\bar{\alpha}_{t-1})}\,\bar{\boldsymbol{\varepsilon}} + \sqrt{\beta_t}\,\boldsymbol{\varepsilon}_t \tag{10.5}$$

Par la propriété de somme de gaussiennes indépendantes, la variance totale vaut :

$$\alpha_t(1-\bar{\alpha}_{t-1}) + \beta_t = \alpha_t - \bar{\alpha}_t + 1 - \alpha_t = 1 - \bar{\alpha}_t$$

ce qui achève la récurrence. □

Équations clés du processus direct

$$\alpha_t = 1 - \beta_t \tag{10.6}$$

$$\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s \tag{10.7}$$

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon}, \quad \boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{10.8}$$

10.2 Processus inverse et objectif d'entraînement

10.2.1 Paramétrisation du processus inverse

Le processus inverse est également modélisé comme une chaîne de Markov gaussienne :

$$p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) = \mathcal{N}\big(\mathbf{x}_{t-1};\, \boldsymbol{\mu}_\theta(\mathbf{x}_t, t),\, \sigma_t^2 \mathbf{I}\big) \tag{10.9}$$

Théorème 10.4 (Postérieur direct conditionnel)**.** *Le postérieur $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0)$ est gaussien :*

$$q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}\Big(\mathbf{x}_{t-1};\, \tilde{\boldsymbol{\mu}}_t(\mathbf{x}_t, \mathbf{x}_0),\, \tilde{\beta}_t \mathbf{I}\Big) \tag{10.10}$$

où :

$$\tilde{\boldsymbol{\mu}}_t = \frac{\sqrt{\bar{\alpha}_{t-1}}\,\beta_t}{1-\bar{\alpha}_t}\,\mathbf{x}_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\,\mathbf{x}_t \tag{10.11}$$

$$\tilde{\beta}_t = \frac{(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\,\beta_t \tag{10.12}$$

Démonstration. Par la règle de Bayes :

$$q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) \propto q(\mathbf{x}_t \mid \mathbf{x}_{t-1})\, q(\mathbf{x}_{t-1} \mid \mathbf{x}_0)$$

Les deux termes sont gaussiens. En développant les exposants et en complétant le carré en $\mathbf{x}_{t-1}$, on identifie la moyenne et la variance du postérieur. Le calcul détaillé donne :

$$-\frac{1}{2}\left[\frac{(\mathbf{x}_t - \sqrt{\alpha_t}\,\mathbf{x}_{t-1})^2}{\beta_t} + \frac{(\mathbf{x}_{t-1} - \sqrt{\bar{\alpha}_{t-1}}\,\mathbf{x}_0)^2}{1-\bar{\alpha}_{t-1}}\right] \tag{10.13}$$

$$= -\frac{1}{2}\left[\left(\frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar{\alpha}_{t-1}}\right)\mathbf{x}_{t-1}^2 - 2\left(\frac{\sqrt{\alpha_t}\,\mathbf{x}_t}{\beta_t} + \frac{\sqrt{\bar{\alpha}_{t-1}}\,\mathbf{x}_0}{1-\bar{\alpha}_{t-1}}\right)\mathbf{x}_{t-1} + C\right] \tag{10.14}$$

où C ne dépend pas de $\mathbf{x}_{t-1}$. L'identification avec la forme $-\frac{1}{2\tilde{\beta}_t}(\mathbf{x}_{t-1} - \tilde{\boldsymbol{\mu}}_t)^2$ donne les expressions annoncées. □

10.2.2 Dérivation de l'objectif simplifié

Théorème 10.5 (Borne variationnelle (ELBO))**.** *La log-vraisemblance est bornée inférieurement par :*

$$\log p_\theta(\mathbf{x}_0) \geq \mathbb{E}_q\left[\log \frac{p_\theta(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T} \mid \mathbf{x}_0)}\right] = -\sum_{t=1}^{T} \mathbb{E}_q[\mathrm{KL}(q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) \parallel p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t))] + C \tag{10.15}$$

En substituant $\mathbf{x}_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}(\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon})$ dans $\tilde{\boldsymbol{\mu}}_t$ et en paramétrisant le réseau pour prédire le bruit $\boldsymbol{\varepsilon}_\theta(\mathbf{x}_t, t)$, on obtient :

Objectif simplifié DDPM

$$L_{\text{simple}} = \mathbb{E}_{t,\mathbf{x}_0,\boldsymbol{\varepsilon}}\left[\left\|\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}_\theta\left(\sqrt{\bar{\alpha}_t}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon},\ t\right)\right\|^2\right] \tag{10.16}$$

Intuition

L'objectif simplifié revient à entraîner un réseau à « débruiter » : étant donné une image bruitée $\mathbf{x}_t$ et le pas de temps t, le réseau prédit le bruit $\boldsymbol{\varepsilon}$ qui a été ajouté.

10.3 Schedules de bruit

10.3.1 Schedule linéaire

Ho et al. (2020) proposent un schedule linéaire :

$$\beta_t = \beta_{\min} + \frac{t-1}{T-1}(\beta_{\max} - \beta_{\min}) \tag{10.17}$$

avec $\beta_{\min} = 10^{-4}$ et $\beta_{\max} = 0{,}02$ pour $T = 1000$.

10.3.2 Schedule cosinus

Nichol & Dhariwal (2021) introduisent le schedule cosinus pour éviter un bruitage trop rapide aux premiers pas :

$$\bar{\alpha}_t = \frac{f(t)}{f(0)}, \quad f(t) = \cos\left(\frac{t/T + s}{1+s} \cdot \frac{\pi}{2}\right)^2 \tag{10.18}$$

où $s = 0{,}008$ est un décalage pour éviter que β_t soit trop petit près de $t = 0$.

10.4 Diagramme du processus de diffusion

10.5 Architecture U-Net pour le débruitage

Le réseau $\boldsymbol{\varepsilon}_\theta$ prend la forme d'un U-Net avec :

- Des blocs résiduels avec normalisation de groupe

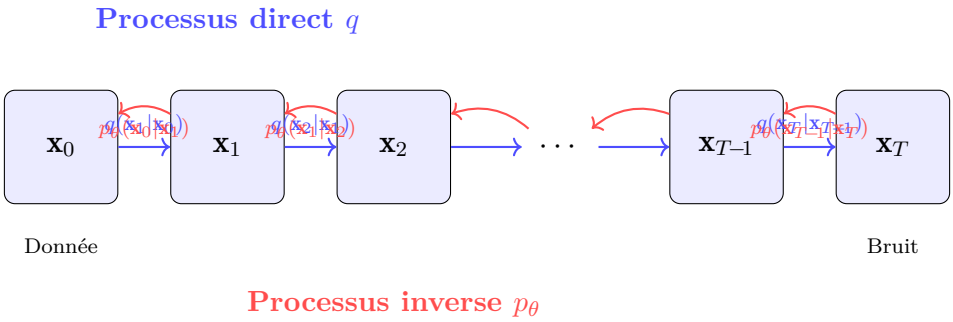


FIG. 10.1 : Processus direct (bruitage) et inverse (débruitage) dans un modèle de diffusion.

- Des *skip connections* entre l'encodeur et le décodeur
- Un *embedding temporel* injecté via addition ou modulation
- Des couches d'attention à chaque résolution spatiale

10.5.1 Embedding temporel

Le pas de temps t est encodé via un embedding sinusoïdal (comme dans le Transformer, cf. chapitre 7) :

$$\text{PE}(t, 2i) = \sin\left(\frac{t}{10000^{2i/d}}\right), \quad \text{PE}(t, 2i+1) = \cos\left(\frac{t}{10000^{2i/d}}\right) \tag{10.19}$$

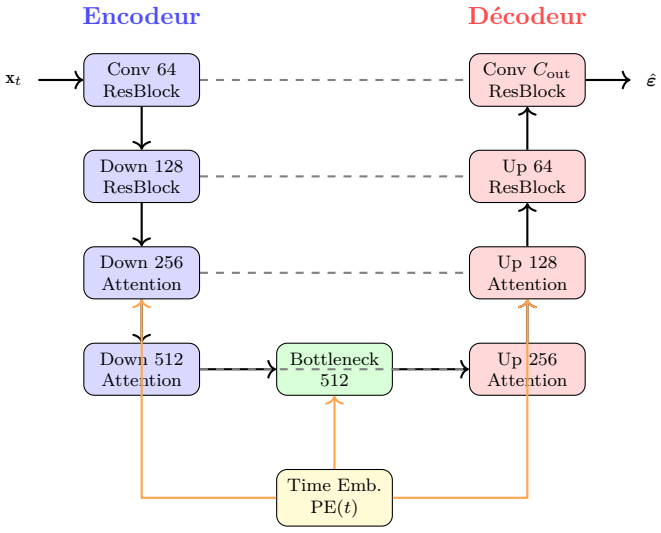


FIG. 10.2 : Architecture U-Net pour la prédiction de bruit avec skip connections et embedding temporel.

10.6 Algorithmes d'échantillonnage

10.6.1 DDPM : échantillonnage stochastique

L'échantillonnage DDPM suit le processus inverse :

$$\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\,\boldsymbol{\varepsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t\,\mathbf{z}, \quad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{10.20}$$

où $\sigma_t = \sqrt{\tilde{\beta}_t}$ ou $\sigma_t = \sqrt{\beta_t}$.

10.6.2 DDIM : échantillonnage déterministe

Song et al. (2020) montrent qu'on peut définir un processus non-markovien donnant les mêmes marginales $q(\mathbf{x}_t \mid \mathbf{x}_0)$ mais avec un échantillonnage déterministe :

Théorème 10.6 (Échantillonnage DDIM)**.** *Pour $\eta = 0$ (cas déterministe) :*

$$\mathbf{x}_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\underbrace{\left(\frac{\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\varepsilon}_\theta(\mathbf{x}_t, t)}{\sqrt{\bar{\alpha}_t}}\right)}_{\text{« prédiction de » } \mathbf{x}_0} + \sqrt{1-\bar{\alpha}_{t-1}}\,\boldsymbol{\varepsilon}_\theta(\mathbf{x}_t, t) \tag{10.21}$$

Remarque 10.7. DDIM permet d'échantillonner en $S \ll T$ pas en choisissant un sous-ensemble de pas de temps $\{\tau_1, \ldots, \tau_S\} \subset \{1, \ldots, T\}$.

10.7 Guidage sans classifieur

Définition 10.8 (Classifier-Free Guidance)**.** Soit c un conditionnement (texte, classe, etc.). Le bruit guidé est :

$$\hat{\boldsymbol{\varepsilon}}_\theta(\mathbf{x}_t, t, c) = (1+w)\,\boldsymbol{\varepsilon}_\theta(\mathbf{x}_t, t, c) - w\,\boldsymbol{\varepsilon}_\theta(\mathbf{x}_t, t, \varnothing) \tag{10.22}$$

où $w > 0$ est l'échelle de guidage et $\varnothing$ dénote l'absence de conditionnement.

Dérivation du guidage

En termes de scores :

$$\nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t \mid c) = \nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t) + \nabla_{\mathbf{x}_t} \log p(c \mid \mathbf{x}_t) \tag{10.23}$$

$$\hat{\nabla}_{\mathbf{x}_t} \log p(\mathbf{x}_t \mid c) = \nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t) + (1+w)\nabla_{\mathbf{x}_t} \log p(c \mid \mathbf{x}_t) \tag{10.24}$$

La deuxième ligne amplifie le gradient du classifieur implicite par $(1+w)$, ce qui renforce l'adhérence au conditionnement.

10.8 Implémentation PyTorch : DDPM sur MNIST

CHAPITRE 10. MODÈLES DE DIFFUSION

Réseau de prédiction de bruit simple

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

class SinusoidalTimeEmbedding(nn.Module):
    """Embedding temporel sinusoidal."""
    def __init__(self, dim):
        super().__init__()
        self.dim = dim

    def forward(self, t):
        device = t.device
        half = self.dim // 2
        emb = math.log(10000) / (half - 1)
        emb = torch.exp(torch.arange(half, device=device) * -emb)
        emb = t[:, None].float() * emb[None, :]
        return torch.cat([emb.sin(), emb.cos()], dim=-1)


class ResBlock(nn.Module):
    """Bloc residentiel avec injection temporelle."""
    def __init__(self, in_ch, out_ch, time_dim):
        super().__init__()
        self.conv1 = nn.Sequential(
            nn.GroupNorm(8, in_ch), nn.SiLU(),
            nn.Conv2d(in_ch, out_ch, 3, padding=1))
        self.time_mlp = nn.Sequential(
            nn.SiLU(), nn.Linear(time_dim, out_ch))
        self.conv2 = nn.Sequential(
            nn.GroupNorm(8, out_ch), nn.SiLU(),
            nn.Conv2d(out_ch, out_ch, 3, padding=1))
        self.skip = nn.Conv2d(in_ch, out_ch, 1) if in_ch != out_ch else
        ↪  nn.Identity()

    def forward(self, x, t_emb):
        h = self.conv1(x)
        h = h + self.time_mlp(t_emb)[:, :, None, None]
        h = self.conv2(h)
        return h + self.skip(x)


class SimpleUNet(nn.Module):
    """U-Net simplifi\'e pour MNIST (28x28, 1 canal)."""
    def __init__(self, time_dim=128):
        super().__init__()
        self.time_embed = nn.Sequential(
            SinusoidalTimeEmbedding(time_dim),
            nn.Linear(time_dim, time_dim), nn.SiLU())
```

```
        # Encodeur
        self.enc1 = ResBlock(1, 32, time_dim)
        self.enc2 = ResBlock(32, 64, time_dim)
        self.pool = nn.MaxPool2d(2)

        # Goulot
        self.bot = ResBlock(64, 128, time_dim)

        # Decodeur
        self.up2 = nn.ConvTranspose2d(128, 64, 2, stride=2)
        self.dec2 = ResBlock(128, 64, time_dim)
        self.up1 = nn.ConvTranspose2d(64, 32, 2, stride=2)
        self.dec1 = ResBlock(64, 32, time_dim)

        self.out = nn.Conv2d(32, 1, 1)

    def forward(self, x, t):
        t_emb = self.time_embed(t)
        # Encodeur
        e1 = self.enc1(x, t_emb)            # 28x28
        e2 = self.enc2(self.pool(e1), t_emb)  # 14x14
        # Goulot
        b = self.bot(self.pool(e2), t_emb)  # 7x7
        # Decodeur + skip connections
        d2 = self.dec2(torch.cat([self.up2(b), e2], 1), t_emb)
        d1 = self.dec1(torch.cat([self.up1(d2), e1], 1), t_emb)
        return self.out(d1)
```

Boucle d'entraînement DDPM

```
import torch.nn.functional as F
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Hyperparametres
T = 1000
beta = torch.linspace(1e-4, 0.02, T)
alpha = 1.0 - beta
alpha_bar = torch.cumprod(alpha, dim=0)

def q_sample(x0, t, noise=None):
    """Echantillonne x_t a partir de x_0."""
    if noise is None:
        noise = torch.randn_like(x0)
    ab = alpha_bar[t][:, None, None, None].to(x0.device)
    return torch.sqrt(ab) * x0 + torch.sqrt(1 - ab) * noise, noise

# Donnees
transform = transforms.Compose([
    transforms.ToTensor(),
```

CHAPITRE 10. MODÈLES DE DIFFUSION

```
    transforms.Normalize((0.5,), (0.5,))])
train_set = datasets.MNIST('.', train=True, download=True,
↪  transform=transform)
loader = DataLoader(train_set, batch_size=128, shuffle=True)

# Modele et optimiseur
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = SimpleUNet().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=2e-4)

# Entrainement
for epoch in range(20):
    total_loss = 0
    for imgs, _ in loader:
        imgs = imgs.to(device)
        t = torch.randint(0, T, (imgs.size(0),))
        x_t, noise = q_sample(imgs, t)
        x_t = x_t.to(device)
        noise = noise.to(device)
        pred = model(x_t, t.to(device))
        loss = F.mse_loss(pred, noise)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {total_loss/len(loader):.4f}")
```

Boucle d'échantillonnage DDPM

```
@torch.no_grad()
def ddpm_sample(model, shape, T=1000):
    """Genere des echantillons via DDPM."""
    device = next(model.parameters()).device
    x = torch.randn(shape, device=device)
    for t in reversed(range(T)):
        t_batch = torch.full((shape[0],), t, device=device,
        ↪  dtype=torch.long)
        eps_pred = model(x, t_batch)
        coeff = beta[t] / torch.sqrt(1 - alpha_bar[t])
        mu = (x - coeff * eps_pred) / torch.sqrt(alpha[t])
        if t > 0:
            z = torch.randn_like(x)
            sigma = torch.sqrt(beta[t])
            x = mu + sigma * z
        else:
            x = mu
    return x

# Generer 16 images
```

```
samples = ddpm_sample(model, (16, 1, 28, 28))
```

Sortie

```
Epoch 1, Loss: 0.1283
Epoch 2, Loss: 0.0541
Epoch 3, Loss: 0.0472
...
Epoch 20, Loss: 0.0298
```

10.9 Étude d'ablation

TAB. 10.1 : Étude d'ablation sur MNIST : impact des hyperparamsètres.

Configuration	Pas T	Schedule	FID ↓
Base	1000	Linéaire	12.4
Réduit	200	Linéaire	28.7
Cosinus	1000	Cosinus	10.1
Petit réseau (32 dim)	1000	Linéaire	18.9
Grand réseau (256 dim)	1000	Cosinus	7.8
DDIM ($S = 50$)	1000	Cosinus	11.3

Remarque 10.9. Le schedule cosinus améliore systématiquement le FID en répartissant le budget de bruit plus uniformément sur les pas de temps. DDIM avec $S = 50$ pas offre un excellent compromis vitesse/qualité.

10.10 État de l'art et connexions

Modèles de diffusion en 2024–2025

- **Stable Diffusion / SDXL** : diffusion dans l'espace latent d'un autoencodeur, permettant la génération d'images haute résolution conditionnée par le texte via CLIP.
- **DALL·E 3** : intégration fine avec les modèles de langage pour un suivi précis des instructions textuelles.
- **Flow Matching** : généralisation des modèles de diffusion utilisant des équations différentielles ordinaires (ODE) avec des champs de vecteurs conditionnels, permettant des trajectoires plus directes.
- **Consistency Models** : distillation du processus de diffusion en un modèle à un seul pas, éliminant le besoin d'échantillonnage itératif.
- **Video Generation** : Sora (OpenAI), extension des modèles de diffusion à la génération vidéo spatio-temporelle.

Attention

Les modèles de diffusion nécessitent des ressources de calcul considérables pour l'entraînement (des milliers d'heures GPU). Les approches dans l'espace latent (LDM) réduisent significativement ce coût.

10.11 Exercices

Exercice 10.1 (Dérivation de la forme fermée ($\star$))**.** Démontrez par récurrence que $q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}(\sqrt{\bar{\alpha}_t}\,\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I})$ en détaillant chaque étape du calcul de la variance.

Exercice 10.2 (Postérieur conditionnel ($\star\star$))**.** Dérivez complètement $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0)$ en partant de la règle de Bayes. Montrez explicitement la complétion du carré.

Exercice 10.3 (DDIM comme ODE ($\star\star$))**.** Montrez que l'équation DDIM (10.21) peut être interprétée comme la discrétisation d'Euler d'une ODE de probabilité. Quel est le champ de vecteurs associé ?

Exercice 10.4 (Implémentation du schedule cosinus ($\star$))**.** Implémentez en PyTorch le schedule cosinus de Nichol & Dhariwal. Tracez $\bar{\alpha}_t$ et β_t en fonction de t et comparez avec le schedule linéaire.

Exercice 10.5 (Classifier-Free Guidance ($\star\star\star$))**.**

1. Dérivez l'équation (10.22) à partir du score conditionnel $\nabla_{\mathbf{x}} \log p(\mathbf{x} \mid c)$.
2. Modifiez le U-Net de la section 10.8 pour accepter un conditionnement de classe. Entraînez avec un taux de *dropout* de conditionnement de 10%.
3. Générez des chiffres MNIST spécifiques avec différentes valeurs de $w \in \{0, 1, 3, 7\}$ et analysez l'évolution de la qualité et de la diversité.

Exercice 10.6 (Comparaison DDPM vs DDIM ($\star\star$))**.** Implémentez l'échantillonnage DDIM et comparez :

1. La qualité visuelle pour $S \in \{10, 50, 100, 1000\}$ pas
2. Le temps d'inférence
3. La capacité d'interpolation dans l'espace latent (propriété déterministe de DDIM)

Chapitre 11

Aspects Théoriques — Généralisation, Expressivité

Ce chapitre explore les fondements théoriques de l'apprentissage profond : pourquoi les réseaux surparamétrés généralisent-ils ? Quelle est leur puissance expressive ? Comment la géométrie du paysage de perte influence-t-elle l'optimisation ? Nous présentons les cadres PAC, la dimension VC, la complexité de Rademacher, la théorie du noyau tangent neural (NTK), et les résultats récents sur la double descente et les lois d'échelle.

11.1 Cadre PAC pour les réseaux de neurones

11.1.1 Définitions fondamentales

Définition 11.1 (Apprentissage PAC)**.** Une classe d'hypothèses $\mathcal{H}$ est *PAC-apprenable* s'il existe un algorithme $\mathcal{A}$ tel que, pour tout $\varepsilon > 0$, $\delta > 0$ et toute distribution $\mathcal{D}$ sur $\mathcal{X} \times \mathcal{Y}$, avec probabilité au moins $1 - \delta$ sur un échantillon $S \sim \mathcal{D}^m$:

$$R(h_S) \leq \min_{h \in \mathcal{H}} R(h) + \varepsilon \tag{11.1}$$

dès que $m \geq m_{\mathcal{H}}(\varepsilon, \delta)$, où $R(h) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(h(x), y)]$ est le risque réel.

Théorème 11.2 (Borne de généralisation PAC)**.** *Pour une classe $\mathcal{H}$ finie, avec probabilité $\geq 1 - \delta$:*

$$R(h_S) \leq \hat{R}_S(h_S) + \sqrt{\frac{\log |\mathcal{H}| + \log(1/\delta)}{2m}} \tag{11.2}$$

où $\hat{R}_S$ est le risque empirique.

Remarque 11.3. Cette borne est trop grossière pour les réseaux de neurones modernes qui ont des milliards de paramètres ($\log |\mathcal{H}|$ serait énorme). Des mesures de complexité plus fines sont nécessaires.

11.2 Dimension VC des réseaux de neurones

Définition 11.4 (Dimension VC)**.** La dimension de Vapnik-Chervonenkis de $\mathcal{H}$ est la taille maximale d'un ensemble de points que $\mathcal{H}$ peut *pulvériser* (shattering), c'est-à-dire réaliser toutes les 2^m dichotomies possibles.

Proposition 11.5 (VC des classifieurs linéaires)**.** La dimension VC des classifieurs linéaires dans $\mathbb{R}^d$ est $d+1$.

Théorème 11.6 (VC des réseaux ReLU)**.** *Un réseau de neurones à activations* ReLU *avec W poids et L couches satisfait :*

$$VCdim(\mathcal{H}) = O(WL \log W) \tag{11.3}$$

Idée de la preuve. Le résultat repose sur le fait qu'un réseau ReLU à W poids et L couches définit une fonction linéaire par morceaux dont le nombre de régions linéaires est borné par $O\left(\binom{W}{d}^L\right)$. Chaque région correspond à un motif d'activation, et la dimension VC est bornée par le logarithme du nombre de dichotomies réalisables. On utilise le lemme de Sauer-Shelah :

$$|\{(h(x_1), \dots, h(x_m)) : h \in \mathcal{H}\}| \leq \sum_{i=0}^{d_{\text{VC}}} \binom{m}{i} \tag{11.4}$$

Pour les détails, voir Bartlett et al. (2019). □

Attention

La dimension VC donne une borne de généralisation de la forme $O\left(\sqrt{WL \log W / m}\right)$, qui est *vacuous* (supérieure à 1) pour les grands réseaux modernes. Cela motive la recherche de bornes plus fines.

11.3 Complexité de Rademacher

Définition 11.7 (Complexité de Rademacher empirique)**.** Pour un échantillon $S = \{x_1, \dots, x_m\}$:

$$\hat{\mathfrak{R}}_S(\mathcal{H}) = \mathbb{E}_{\boldsymbol{\sigma}}\left[\sup_{h \in \mathcal{H}} \frac{1}{m} \sum_{i=1}^{m} \sigma_i h(x_i)\right] \tag{11.5}$$

où $\sigma_i \sim \text{Unif}\{-1, +1\}$ sont des variables de Rademacher indépendantes.

Théorème 11.8 (Borne de généralisation par Rademacher)**.** *Avec probabilité $\geq 1 - \delta$:*

$$R(h) \leq \hat{R}_S(h) + 2\,\hat{\mathfrak{R}}_S(\mathcal{H}) + 3\sqrt{\frac{\log(2/\delta)}{2m}} \tag{11.6}$$

Proposition 11.9 (Rademacher des réseaux à bornes spectrales)**.** Pour un réseau à L couches avec matrices de poids $\mathbf{W}_1, \dots, \mathbf{W}_L$ et activation ReLU :

$$\hat{\mathfrak{R}}_S(\mathcal{H}) \leq \frac{B_x \prod_{\ell=1}^{L} \|\mathbf{W}_\ell\|_\sigma}{\sqrt{m}} \cdot \sqrt{2L \log(2)} \tag{11.7}$$

où $\|\mathbf{W}_\ell\|_\sigma$ est la norme spectrale et $B_x = \max_i \|x_i\|$.

Intuition

La complexité de Rademacher mesure la capacité d'une classe de fonctions à s'ajuster à du bruit aléatoire. Un réseau dont les normes spectrales des poids restent

contrôlées généralise mieux, même avec beaucoup de paramètres.

11.4 Double descente et surparamétrisation

11.4.1 Le phénomène de double descente

Contrairement à la vision classique du compromis biais-variance, les réseaux modernes surparamétrés exhibent un comportement de *double descente* :

1. **Régime sous-paramétré** : le risque de test suit la courbe en U classique.
2. **Seuil d'interpolation** : quand le nombre de paramètres $\approx$ nombre d'échantillons, le risque de test explose.
3. **Régime surparamétré** : au-delà du seuil, le risque de test *redescend* et peut atteindre des valeurs inférieures au minimum classique.

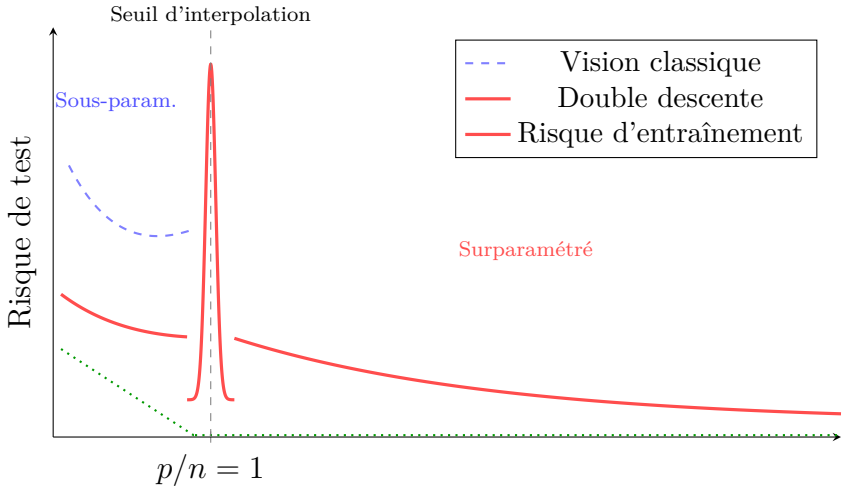


FIG. 11.1 : Courbe de double descente : le risque de test diminue à nouveau dans le régime surparamétré, contredisant le compromis biais-variance classique.

11.4.2 Biais implicite de la descente de gradient

Théorème 11.10 (Biais implicite — régression linéaire)**.** *Pour la régression linéaire surparamétrée ($p > n$), la descente de gradient initialisée à $\mathbf{w}_0 = \mathbf{0}$ converge vers la solution de norme minimale :*

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \|\mathbf{w}\| \quad s.c. \quad \mathbf{X}\mathbf{w} = \mathbf{y} \tag{11.8}$$

Démonstration. La descente de gradient produit $\mathbf{w}_t \in \mathrm{Im}(\mathbf{X}^\top)$ à chaque itération. L'interpolateur dans cet espace est unique et correspond à la solution pseudo-inverse $\mathbf{w}^* = \mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top)^{-1}\mathbf{y}$, qui est de norme minimale par construction. □

Remarque 11.11. Pour les réseaux non linéaires, le biais implicite est plus complexe et dépend de l'architecture, du taux d'apprentissage, et de la taille des batchs. Voir également le chapitre 2 pour les liens avec SGD.

11.5 Noyau Tangent Neural (NTK)

11.5.1 Linéarisation à l'initialisation

Définition 11.12 (Noyau Tangent Neural)**.** Soit $f(\mathbf{x};\boldsymbol{\theta})$ un réseau de neurones. Le NTK est défini par :

$$\Theta(\mathbf{x},\mathbf{x}') = \left\langle \frac{\partial f(\mathbf{x};\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}, \frac{\partial f(\mathbf{x}';\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right\rangle = \sum_{i=1}^{P} \frac{\partial f(\mathbf{x};\boldsymbol{\theta})}{\partial \theta_i} \frac{\partial f(\mathbf{x}';\boldsymbol{\theta})}{\partial \theta_i} \tag{11.9}$$

où P est le nombre total de paramètres.

Théorème 11.13 (Convergence du NTK — Jacot et al., 2018)**.** *Pour un réseau à L couches de largeur $n_\ell \to \infty$ avec paramétrisation NTK, le noyau $\Theta(\mathbf{x},\mathbf{x}')$ converge en probabilité vers un noyau déterministe $\Theta^*(\mathbf{x},\mathbf{x}')$ à l'initialisation, et* reste constant *au cours de l'entraînement.*

11.5.2 Dérivation de la linéarisation

Linéarisation de Taylor du réseau

Au voisinage de l'initialisation $\boldsymbol{\theta}_0$:

$$f(\mathbf{x};\boldsymbol{\theta}) \approx f(\mathbf{x};\boldsymbol{\theta}_0) + \nabla_{\boldsymbol{\theta}} f(\mathbf{x};\boldsymbol{\theta}_0)^\top (\boldsymbol{\theta} - \boldsymbol{\theta}_0) \tag{11.10}$$

Sous cette approximation, l'entraînement par descente de gradient sur la perte quadratique donne :

$$\frac{d\mathbf{f}}{dt} = -\eta\, \boldsymbol{\Theta}_0\, (\mathbf{f}(t) - \mathbf{y}) \tag{11.11}$$

où $\mathbf{f}(t) = (f(x_1;\boldsymbol{\theta}_t), \ldots, f(x_n;\boldsymbol{\theta}_t))^\top$ et $[\boldsymbol{\Theta}_0]_{ij} = \Theta(\mathbf{x}_i,\mathbf{x}_j)\big|_{\boldsymbol{\theta}_0}$.
La solution est :

$$\mathbf{f}(t) = \mathbf{y} - e^{-\eta\, \boldsymbol{\Theta}_0\, t}(\mathbf{y} - \mathbf{f}(0)) \tag{11.12}$$

ce qui montre une convergence exponentielle vers l'interpolation $\mathbf{f} = \mathbf{y}$ si $\boldsymbol{\Theta}_0 \succ 0$.

Remarque 11.14. La théorie NTK explique l'entraînement dans le régime dit « lazy » (faible variation des poids). En pratique, les réseaux opèrent souvent dans un régime « riche » où les représentations évoluent significativement.

11.6 Expressivité : profondeur vs largeur

11.6.1 Théorème d'approximation universelle

Théorème 11.15 (Cybenko, 1989 ; Hornik, 1991)**.** *Un réseau à une couche cachée avec activation continue non polynomiale peut approcher toute fonction continue sur un compact à précision ε arbitraire, pourvu que la largeur soit suffisante.*

Attention

Ce théorème est un résultat d'*existence* : il ne dit rien sur la largeur nécessaire, qui peut être exponentiellement grande.

11.6.2 Séparation profondeur-largeur

Théorème 11.16 (Telgarsky, 2016)**.** *Il existe des fonctions calculables par un réseau ReLU de profondeur L et largeur $O(1)$ qui nécessitent une largeur $\Omega(2^{L/3})$ pour être approchées par un réseau de profondeur $O(1)$.*

Esquisse. Considérons la fonction « dents de scie » $g : [0,1] \to [0,1]$ définie par $g(x) = 2\min(x, 1-x)$. La composition itérée $g^{\circ k} = g \circ \cdots \circ g$ (k fois) est une fonction linéaire par morceaux avec 2^k morceaux. Or :

- $g^{\circ k}$ est réalisable par un réseau ReLU de profondeur $2k$ et largeur 2 : $g(x) = \mathrm{ReLU}(2x) - 2\mathrm{ReLU}(2x-1)$.
- Un réseau de profondeur ℓ et largeur w produit au plus $O(w^\ell)$ régions linéaires.

Si ℓ est constant, il faut $w = \Omega(2^{k/\ell})$ régions, d'où la séparation exponentielle. □

11.7 Hypothèse du billet gagnant

Théorème 11.17 (Frankle & Carlin, 2019 — Hypothèse du billet gagnant)**.** *Un réseau dense initialisé aléatoirement contient un sous-réseau (*billet gagnant*) qui, entraîné isolément avec les mêmes poids initiaux, atteint des performances comparables au réseau complet en un nombre comparable d'itérations.*

Remarque 11.18. L'identification du sous-réseau gagnant se fait par élagage itératif (Iterative Magnitude Pruning) : entraîner, élaguer les poids de plus faible magnitude, réinitialiser aux poids d'origine, répéter. Les taux de compression atteignent 90–99% sur certaines architectures.

11.8 Géométrie du paysage de perte

11.8.1 Points selles et minima plats

Proposition 11.19 (Prédominance des points selles)**.** Dans un espace de dimension d élevée, les points critiques (où $\nabla L = 0$) sont exponentiellement plus susceptibles d'être des points selles que des minima locaux. Si chaque valeur propre de la hessienne est positive ou négative avec probabilité $1/2$, la probabilité d'un minimum local est 2^{-d}.

Définition 11.20 (Sharpness-Aware Minimization (SAM))**.** SAM cherche des paramètres dans des régions « plates » en optimisant :

$$\min_{\boldsymbol{\theta}} \max_{\|\boldsymbol{\epsilon}\| \le \rho} L(\boldsymbol{\theta} + \boldsymbol{\epsilon}) \tag{11.13}$$

L'approximation au premier ordre donne la mise à jour :

$$\hat{\boldsymbol{\epsilon}} = \rho \frac{\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})}{\|\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})\|}, \quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \, \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta} + \hat{\boldsymbol{\epsilon}}) \tag{11.14}$$

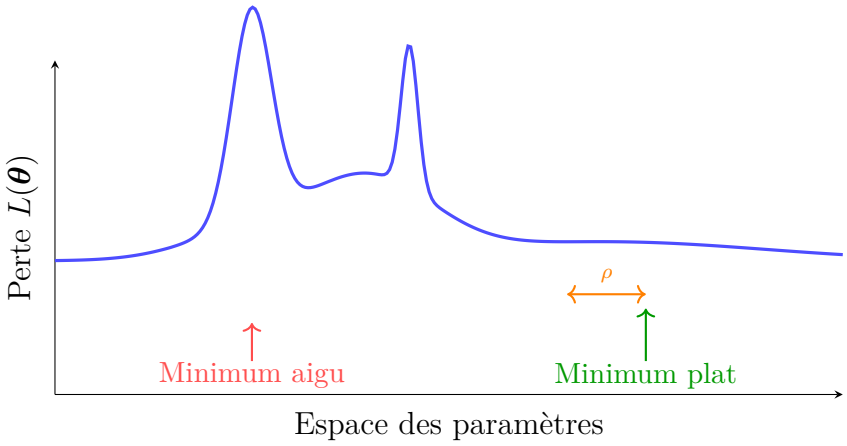


FIG. 11.2 : Minima aigus vs plats : SAM favorise les minima plats qui généralisent mieux car la perte varie peu dans un voisinage de rayon ρ.

11.9 Théorie du goulot d'étranglement informationnel

Définition 11.21 (Principe du goulot d'étranglement)**.** Soit X l'entrée, Y la cible, et T la représentation interne. Le principe IB cherche à :

$$\min_{p(t|x)} \; I(X;T) - \beta\, I(T;Y) \tag{11.15}$$

où $I(\cdot;\cdot)$ dénote l'information mutuelle et $\beta > 0$ contrôle le compromis compression/prédiction.

Remarque 11.22. Shwartz-Ziv & Tishby (2017) conjecturent que l'entraînement des réseaux profonds passe par deux phases : (1) *fitting* où $I(T;Y)$ augmente, puis (2) *compression* où $I(X;T)$ diminue. Cette théorie reste débattue : les résultats dépendent de la fonction d'activation et de l'estimateur d'information mutuelle utilisé.

11.10 Démonstration expérimentale de la double descente

Double descente expérimentale avec des réseaux de largeur variable

```
import torch
import torch.nn as nn
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Donnees : classification binaire, n petit pour observer la double
↪  descente
X, y = make_classification(n_samples=200, n_features=20,
                           n_informative=10, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42)
```

```
X_train_t = torch.tensor(X_train, dtype=torch.float32)
y_train_t = torch.tensor(y_train, dtype=torch.float32).unsqueeze(1)
X_test_t = torch.tensor(X_test, dtype=torch.float32)
y_test_t = torch.tensor(y_test, dtype=torch.float32).unsqueeze(1)

def make_model(width):
    return nn.Sequential(
        nn.Linear(20, width), nn.ReLU(),
        nn.Linear(width, width), nn.ReLU(),
        nn.Linear(width, 1))

def train_and_eval(width, epochs=2000, lr=1e-3):
    model = make_model(width)
    opt = torch.optim.Adam(model.parameters(), lr=lr)
    loss_fn = nn.BCEWithLogitsLoss()
    for _ in range(epochs):
        opt.zero_grad()
        loss_fn(model(X_train_t), y_train_t).backward()
        opt.step()
    with torch.no_grad():
        train_loss = loss_fn(model(X_train_t), y_train_t).item()
        test_loss = loss_fn(model(X_test_t), y_test_t).item()
    return train_loss, test_loss

# Varier la largeur (nombre de parametres)
widths = [2, 4, 6, 8, 10, 15, 20, 30, 50, 80, 120, 200, 400, 800]
results = []
for w in widths:
    n_params = 20*w + w + w*w + w + w*1 + 1  # compter les parametres
    tr, te = train_and_eval(w)
    results.append((w, n_params, tr, te))
    print(f"Largeur={w:4d}, Params={n_params:6d}, "
          f"Train={tr:.4f}, Test={te:.4f}")
```

Sortie

```
Largeur=   2, Params=     49, Train=0.5814, Test=0.6231
Largeur=   4, Params=     97, Train=0.4012, Test=0.4893
Largeur=  10, Params=    331, Train=0.1254, Test=0.3891
Largeur=  20, Params=    861, Train=0.0312, Test=0.5127
Largeur=  30, Params=   1591, Train=0.0041, Test=0.4203
Largeur=  50, Params=   3851, Train=0.0003, Test=0.3542
Largeur= 120, Params=  17161, Train=0.0001, Test=0.3201
Largeur= 400, Params= 168801, Train=0.0001, Test=0.2987
Largeur= 800, Params= 657601, Train=0.0001, Test=0.2854
```

Remarque 11.23. On observe que le risque de test augmente d'abord (autour de $w = 20$, près du seuil d'interpolation), puis diminue dans le régime surparamétré. Ce comportement illustre la double descente.

11.11 État de l'art et connexions récentes

Théorie de l'apprentissage profond en 2024–2025

- **Grokking** : phénomène où un modèle mémorise d'abord les données d'entraînement, puis « comprend » la structure sous-jacente après un très grand nombre d'époques, passant d'une généralisation nulle à une généralisation parfaite. Expliqué partiellement par la régularisation implicite du *weight decay*.
- **Lois d'échelle neurales** : Kaplan et al. (2020) et Hoffmann et al. (2022, « Chinchilla ») montrent que la perte suit des lois de puissance :

$$L(N, D) \approx \left(\frac{N_c}{N}\right)^{\alpha_N} + \left(\frac{D_c}{D}\right)^{\alpha_D} + L_\infty \tag{11.16}$$

  où N est le nombre de paramètres et D la taille des données.
- **Interprétabilité mécaniste** : analyse des circuits internes des réseaux pour comprendre *comment* ils implémentent des algorithmes. Exemples : circuits d'induction dans les Transformers, superposition de caractéristiques.
- **Feature learning vs kernel regime** : distinction entre le régime NTK (« lazy », pas d'apprentissage de représentations) et le régime riche (« feature learning »), ce dernier étant celui observé en pratique.

11.12 Exercices

Exercice 11.1 (Dimension VC ($\star$))**.** Montrez que la dimension VC des classifieurs linéaires dans $\mathbb{R}^2$ est 3 en exhibant un ensemble de 3 points pulvérisable et en montrant qu'aucun ensemble de 4 points ne l'est.

Exercice 11.2 (Complexité de Rademacher ($\star\star$))**.** Soit $\mathcal{H} = \{x \mapsto \mathbf{w}^\top x : \|\mathbf{w}\| \leq B\}$. Montrez que :

$$\hat{\mathfrak{R}}_S(\mathcal{H}) \leq \frac{B\sqrt{\sum_{i=1}^m \|x_i\|^2}}{m} \tag{11.17}$$

Indication : utiliser l'inégalité de Cauchy-Schwarz.

Exercice 11.3 (NTK pour un réseau à une couche ($\star\star$))**.** Considérez $f(\mathbf{x}; \boldsymbol{\theta}) = \frac{1}{\sqrt{n}} \sum_{j=1}^n a_j \, \sigma(\mathbf{w}_j^\top \mathbf{x})$ où $\sigma = \mathrm{ReLU}$, $a_j \sim \mathcal{N}(0,1)$ fixés, et $\mathbf{w}_j \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$.

1. Calculez $\Theta(\mathbf{x}, \mathbf{x}') = \sum_j \frac{\partial f}{\partial \mathbf{w}_j} \cdot \frac{\partial f}{\partial \mathbf{w}_j}$ explicitement.
2. Montrez que lorsque $n \to \infty$, $\Theta(\mathbf{x}, \mathbf{x}')$ converge vers un noyau déterministe.
3. Exprimez ce noyau en fonction de l'angle entre $\mathbf{x}$ et $\mathbf{x}'$.

Exercice 11.4 (Séparation profondeur-largeur ($\star\star\star$))**.** Soit $g(x) = 2\min(x, 1-x)$ pour $x \in [0, 1]$.

1. Montrez que g peut s'écrire comme un réseau ReLU de profondeur 2 et largeur 2.

2. Montrez que $g^{\circ k}$ a 2^k morceaux linéaires.

3. En déduire qu'un réseau de profondeur ℓ et largeur w ne peut représenter $g^{\circ k}$ si $w^\ell < 2^k$.

Exercice 11.5 (Biais implicite de SGD ($\star\star$))**.** Dans le cadre de la régression linéaire surparamétrée :

1. Démontrez que la descente de gradient avec $\mathbf{w}_0 = \mathbf{0}$ maintient $\mathbf{w}_t \in \mathrm{Im}(\mathbf{X}^\top)$ pour tout t.

2. Montrez que l'unique interpolateur dans $\mathrm{Im}(\mathbf{X}^\top)$ est $\mathbf{X}^\dagger \mathbf{y}$ (pseudo-inverse).

3. Discutez comment ce résultat éclaire la généralisation des réseaux surparamétrés.

Exercice 11.6 (SAM expérimental ($\star\star$))**.** 1. Implémentez l'optimiseur SAM en PyTorch (deux passes de gradient par itération).

2. Comparez SGD, Adam et SAM sur CIFAR-10 avec un ResNet-18.

3. Mesurez la « sharpness » du minimum trouvé (plus grande valeur propre de la hessienne, approximée par la méthode de la puissance).

4. Vérifiez que SAM trouve des minima plus plats et généralise mieux.

Exercice 11.7 (Lois d'échelle ($\star$))**.** On entraîne des modèles de langage de tailles $N \in \{10^6, 10^7, 10^8, 10^9\}$ paramètres et on observe les pertes de test $L \in \{3.2, 2.8, 2.5, 2.3\}$.

1. Tracez $\log L$ en fonction de $\log N$.

2. Estimez l'exposant α_N par régression linéaire.

3. Extrapolez la perte pour $N = 10^{11}$.

4. Discutez les limites de cette extrapolation.

Chapitre 12

Applications et éthique

Intuition

L'apprentissage profond n'est pas une fin en soi : c'est un outil au service de la science, de la médecine, de l'industrie et de la société. Ce chapitre explore les applications les plus marquantes et les questions éthiques fondamentales que tout praticien doit se poser.

12.1 Vision par ordinateur

12.1.1 Détection d'objets

La détection d'objets localise et classifie simultanément plusieurs objets dans une image. L'approche YOLO (*You Only Look Once*) divise l'image en une grille et prédit directement les boîtes englobantes et les probabilités de classe.

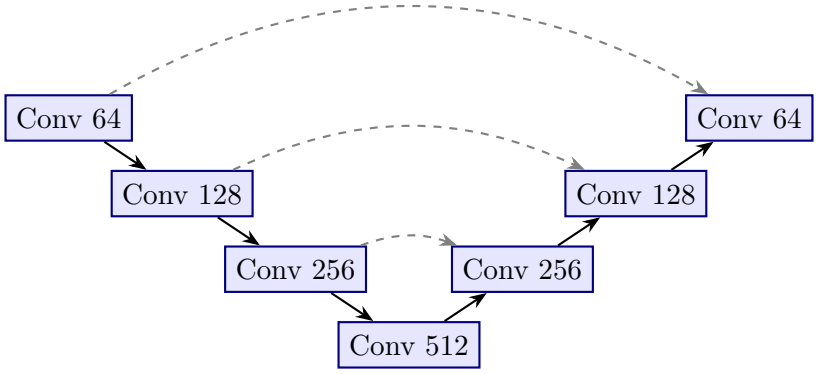
Pour chaque cellule (i, j) de la grille, le modèle prédit :

$$\hat{y}_{ij} = (p_{\text{obj}}, b_x, b_y, b_w, b_h, c_1, \ldots, c_K)$$

où p_{obj} est la probabilité de présence d'un objet, (b_x, b_y, b_w, b_h) définissent la boîte englobante, et c_k sont les probabilités de classe.

12.1.2 Segmentation sémantique

La segmentation assigne une classe à chaque pixel. L'architecture U-Net (Ronneberger et al., 2015) utilise un encodeur-décodeur avec des connexions de saut :



12.1.3 Génération d'images

Les modèles de diffusion (chapitre 10) ont révolutionné la génération d'images. Stable Diffusion opère dans l'espace latent d'un VAE (chapitre 9), réduisant considérablement le coût de calcul.

12.2 Traitement du langage naturel

12.2.1 Modélisation du langage

Les grands modèles de langage (LLM) génèrent du texte de manière auto-régressive :

$$p(\boldsymbol{x}) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}; \theta)$$

État de l'art

- **GPT-4** (OpenAI, 2023) : modèle multimodal avec capacités de raisonnement avancées.
- **Claude** (Anthropic, 2024–2025) : accent sur la sécurité et l'alignement.
- **LLaMA 3** (Meta, 2024) : modèle ouvert performant.
- **Gemini** (Google, 2024) : nativement multimodal.

12.2.2 Traduction automatique

L'architecture Transformer (chapitre 7) a permis des avancées majeures en traduction, atteignant des performances quasi humaines sur de nombreuses paires de langues.

12.2.3 Résumé automatique et réponse aux questions

Les modèles pré-entraînés comme BERT (bidirectionnel) et GPT (auto-régressif) sont fine-tunés pour des tâches spécifiques de compréhension.

12.3 Applications scientifiques

12.3.1 Prédiction de structures protéiques

AlphaFold 2 (Jumper et al., 2021) a résolu le problème du repliement des protéines, vieux de 50 ans. L'architecture combine :

- Un Transformer avec attention évolutionnaire sur les alignements de séquences multiples (MSA).
- Un module de structure qui prédit les coordonnées 3D.
- Un raffinement itératif des prédictions.

Remarque 12.1. AlphaFold 3 (2024) étend la prédiction aux complexes protéine-ligand, protéine-ADN et protéine-ARN, unifiant la modélisation des interactions moléculaires.

12.3.2 Prévision météorologique

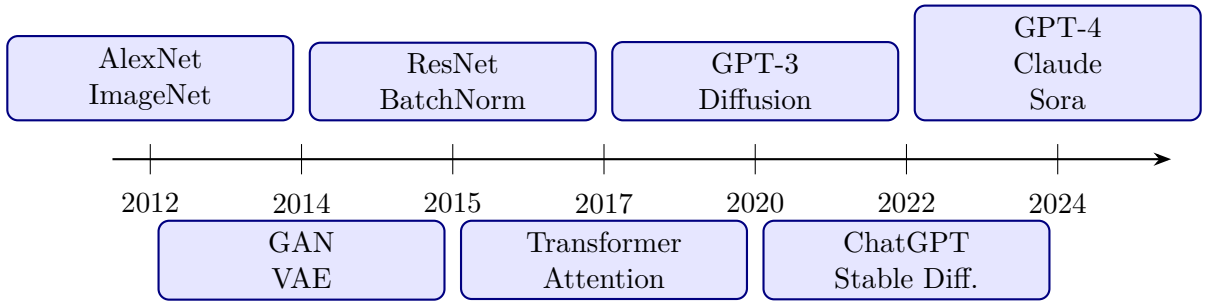
GraphCast (Lam et al., 2023) utilise des réseaux de neurones sur graphes pour la prévision météorologique globale, surpassant les modèles numériques traditionnels tout en étant 1000× plus rapide.

12.3.3 Découverte de médicaments

Les modèles génératifs explorent l'espace chimique pour proposer de nouvelles molécules candidates :

- Génération de molécules par VAE ou modèles de diffusion.
- Prédiction d'affinité moléculaire par GNN (Graph Neural Networks).
- Optimisation multi-objectifs (efficacité, toxicité, synthétisabilité).

12.4 Chronologie des avancées majeures



12.5 Interprétabilité

12.5.1 Cartes de saillance

Les cartes de saillance identifient les régions de l'entrée les plus influentes pour la prédiction :

$$S_i = \left| \frac{\partial f_\theta(\boldsymbol{x})}{\partial x_i} \right|$$

12.5.2 Grad-CAM

Grad-CAM (Selvaraju et al., 2017) utilise les gradients de la couche convolutive cible pour produire une carte de chaleur :

$$\alpha_k^c = \frac{1}{Z} \sum_i \sum_j \frac{\partial y^c}{\partial A_{ij}^k}, \qquad L_{\text{Grad-CAM}}^c = \text{ReLU}\left(\sum_k \alpha_k^c A^k \right)$$

Grad-CAM en PyTorch

```
import torch
import torch.nn.functional as F
```

```python
class GradCAM:
    def __init__(self, model, target_layer):
        self.model = model
        self.gradients = None
        self.activations = None
        target_layer.register_forward_hook(
            lambda m, i, o: setattr(self, 'activations', o.detach()))
        target_layer.register_full_backward_hook(
            lambda m, gi, go: setattr(self, 'gradients', go[0].detach()))

    def __call__(self, x, class_idx=None):
        output = self.model(x)
        if class_idx is None:
            class_idx = output.argmax(1)
        self.model.zero_grad()
        one_hot = torch.zeros_like(output)
        one_hot[0, class_idx] = 1
        output.backward(gradient=one_hot)

        weights = self.gradients.mean(dim=[2, 3], keepdim=True)
        cam = F.relu((weights * self.activations).sum(1, keepdim=True))
        cam = F.interpolate(cam, size=x.shape[2:],
                            mode='bilinear', align_corners=False)
        cam = cam / (cam.max() + 1e-8)
        return cam.squeeze().detach().numpy()
```

12.5.3 SHAP et valeurs de Shapley

Les valeurs de Shapley, issues de la théorie des jeux coopératifs, attribuent à chaque variable une contribution équitable à la prédiction :

$$\phi_i = \sum_{S \subseteq \{1,\ldots,d\} \setminus \{i\}} \frac{|S|!(d - |S| - 1)!}{d!} \left[f(S \cup \{i\}) - f(S) \right]$$

12.6 Biais et équité

12.6.1 Sources de biais

Attention

Les modèles d'apprentissage profond peuvent amplifier les biais présents dans les données d'entraînement :

- **Biais de représentation** : certains groupes sous-représentés dans les données.
- **Biais d'étiquetage** : annotations reflétant des préjugés humains.
- **Biais d'agrégation** : un modèle unique pour des populations hétérogènes.

12.6.2 Métriques d'équité

Définition 12.2 (Parité démographique)**.** Un classificateur satisfait la parité démographique si :

$$P(\hat{Y} = 1 \mid A = a) = P(\hat{Y} = 1 \mid A = b) \quad \forall a, b$$

où A est l'attribut sensible.

Définition 12.3 (Égalité des chances)**.** Un classificateur satisfait l'égalité des chances si :

$$P(\hat{Y} = 1 \mid Y = 1, A = a) = P(\hat{Y} = 1 \mid Y = 1, A = b) \quad \forall a, b$$

12.7 Confidentialité

12.7.1 Confidentialité différentielle

Définition 12.4 ((ε, δ)-confidentialité différentielle)**.** Un mécanisme aléatoire $\mathcal{M}$ satisfait la (ε, δ)-confidentialité différentielle si, pour tous ensembles de données adjacents D, D' (différant d'un seul exemple) et tout ensemble mesurable S :

$$P(\mathcal{M}(D) \in S) \leq e^{\varepsilon} \cdot P(\mathcal{M}(D') \in S) + \delta$$

DP-SGD (Abadi et al., 2016) applique la confidentialité différentielle à la SGD en écrêtant les gradients individuels et en ajoutant du bruit gaussien :

$$\tilde{\boldsymbol{g}}_t = \frac{1}{B}\left(\sum_{i \in \mathcal{B}} \text{clip}(\boldsymbol{g}_i, C) + \mathcal{N}(0, \sigma^2 C^2 I)\right)$$

12.7.2 Apprentissage fédéré

L'apprentissage fédéré entraîne un modèle global sans centraliser les données. Chaque client k calcule une mise à jour locale, et le serveur agrège :

$$\theta_{t+1} = \sum_{k=1}^{K} \frac{n_k}{n} \theta_{t+1}^{(k)}$$

12.8 Impact environnemental

L'entraînement de grands modèles consomme des ressources considérables :

Modèle	Paramètres	CO_2 estimé (tonnes)
BERT-Large	340M	$\sim$0.6
GPT-3	175B	$\sim$500
LLaMA 2 70B	70B	$\sim$290
GPT-4 (estimé)	>1T	$\sim$5 000+

Bonne pratique

- Privilégier le **fine-tuning** de modèles pré-entraînés plutôt que l'entraînement de zéro.
- Utiliser des techniques d'efficacité : quantification, distillation, LoRA.
- Rapporter l'empreinte carbone dans les publications.
- Choisir des centres de calcul alimentés en énergies renouvelables.

12.9 Cadre réglementaire

Le **EU AI Act** (2024) classe les systèmes d'IA par niveau de risque :

1. **Risque inacceptable** : interdit (notation sociale, manipulation subliminale).
2. **Haut risque** : réglementé (santé, justice, recrutement).
3. **Risque limité** : obligations de transparence.
4. **Risque minimal** : libre utilisation.

12.10 Modèles multimodaux et perspectives

État de l'art

- **Modèles multimodaux** : GPT-4V, Claude 3.5 Vision, Gemini combinent texte, image, audio et vidéo dans un seul modèle.
- **Agents IA** : systèmes autonomes utilisant des outils (navigation web, exécution de code, appels API).
- **Modèles du monde** : apprentissage de représentations physiques pour la robotique et la simulation (Sora, World Labs).
- **IA scientifique** : accélération de la découverte dans tous les domaines (matériaux, climat, biologie).

12.11 Résumé du chapitre

Formules clés

- **Détection** : YOLO prédit boîtes englobantes et classes par cellule de grille
- **Segmentation** : U-Net avec encodeur-décodeur et connexions de saut
- **Grad-CAM** : $L^c = \text{ReLU}(\sum_k \alpha_k^c A^k)$ avec $\alpha_k^c = \frac{1}{Z} \sum_{ij} \frac{\partial y^c}{\partial A_{ij}^k}$
- **SHAP** : valeurs de Shapley pour l'attribution

- **DP-SGD** : SGD avec écrêtage des gradients et bruit gaussien
- **Apprentissage fédéré** : agrégation pondérée des mises à jour locales

12.12 Exercices

Exercice 12.1 ($\star$ — Analyse)**.** Montrer que la parité démographique et l'égalité des chances ne peuvent pas être satisfaites simultanément sauf dans des cas dégénérés. Quelles sont les implications pratiques de ce résultat ?

Exercice 12.2 ($\star$ — Calcul)**.** Calculer le nombre de FLOPS nécessaires pour un forward pass d'un Transformer avec $L = 32$ couches, $d = 4096$, $h = 32$ têtes et une séquence de longueur $n = 2048$. Estimer le coût énergétique sur un GPU A100.

Exercice 12.3 ($\star\star$ — Implémentation)**.** Implémenter Grad-CAM pour un ResNet-18 pré-entraîné sur ImageNet. Visualiser les cartes d'attention pour différentes classes et analyser les résultats.

Exercice 12.4 ($\star\star$ — Implémentation)**.** Implémenter un pipeline d'apprentissage fédéré simple avec 5 clients sur MNIST. Comparer la performance avec un entraînement centralisé et analyser l'impact du nombre de rounds de communication.

Exercice 12.5 ($\star\star\star$ — Projet de recherche)**.** Conduire un audit d'équité complet sur un modèle de classification (par exemple, prédiction de revenu sur le dataset Adult). Calculer la parité démographique, l'égalité des chances et l'égalité prédictive. Proposer et implémenter une méthode de débiaisage (pré-traitement, en entraînement, ou post-traitement) et évaluer son impact sur la précision et l'équité.

Glossaire des architectures

Architecture	Description
MLP	Perceptron multicouche — réseau entièrement connecté
CNN	Réseau convolutif — extraction de caractéristiques spatiales
RNN	Réseau récurrent — traitement de séquences
LSTM	Long Short-Term Memory — RNN avec portes d'oubli
GRU	Gated Recurrent Unit — variante simplifiée du LSTM
Transformer	Architecture attention pure — parallélisation complète
GAN	Réseau génératif adversariel — générateur vs discriminateur
VAE	Auto-encodeur variationnel — génération par inférence variationnelle
DDPM	Modèle de diffusion — débruitage itératif
ResNet	Réseau résiduel — connexions de saut
U-Net	Architecture encodeur-décodeur avec connexions de saut
ViT	Vision Transformer — patchs d'images comme tokens
BERT	Transformer bidirectionnel pré-entraîné
GPT	Transformer génératif auto-régressif

Bibliographie

[1] Goodfellow, I., Bengio, Y. & Courville, A. (2016). *Deep Learning*. MIT Press.

[2] Bishop, C.M. & Bishop, H. (2024). *Deep Learning : Foundations and Concepts*. Springer.

[3] Zhang, A., Lipton, Z.C., Li, M. & Smola, A.J. (2023). *Dive into Deep Learning*. Cambridge University Press.

[4] LeCun, Y., Bengio, Y. & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436–444.

[5] Vaswani, A. et al. (2017). Attention is all you need. *NeurIPS*.

[6] He, K. et al. (2016). Deep residual learning. *CVPR*.

[7] Kingma, D.P. & Welling, M. (2014). Auto-encoding variational Bayes. *ICLR*.

[8] Goodfellow, I. et al. (2014). Generative adversarial nets. *NeurIPS*.

[9] Ho, J., Jain, A. & Abbeel, P. (2020). Denoising diffusion probabilistic models. *NeurIPS*.

[10] Kingma, D.P. & Ba, J. (2015). Adam : a method for stochastic optimization. *ICLR*.

[11] Srivastava, N. et al. (2014). Dropout : a simple way to prevent neural networks from overfitting. *JMLR*.

[12] Ioffe, S. & Szegedy, C. (2015). Batch normalization. *ICML*.

[13] Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*.

[14] Bahdanau, D., Cho, K. & Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. *ICLR*.

[15] Bahdanau, D., Cho, K. & Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. *ICLR*.

[16] Srivastava, N. et al. (2014). Dropout : a simple way to prevent neural networks from overfitting. *JMLR*, 15(1), 1929–1958.

[17] Ioffe, S. & Szegedy, C. (2015). Batch normalization : accelerating deep network training. *ICML*.

[18] Ba, J.L., Kiros, J.R. & Hinton, G.E. (2016). Layer normalization. *arXiv :1607.06450*.

[19] Zhang, H. et al. (2018). mixup : beyond empirical risk minimization. *ICLR*.

[20] Yun, S. et al. (2019). CutMix : regularization strategy to train strong classifiers. *ICCV*.

[21] Loshchilov, I. & Hutter, F. (2019). Decoupled weight decay regularization. *ICLR*.

[22] Huang, G. et al. (2017). Densely connected convolutional networks. *CVPR*.

[23] Szegedy, C. et al. (2016). Rethinking the Inception architecture. *CVPR*.

[24] Wu, H. et al. (2021). CvT : introducing convolutions to Vision Transformers. *ICCV*.

[25] Krizhevsky, A., Sutskever, I. & Hinton, G.E. (2012). ImageNet classification with deep convolutional neural networks. *NeurIPS*.

[26] Simonyan, K. & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. *ICLR*.

[27] Tan, M. & Le, Q.V. (2019). EfficientNet : rethinking model scaling for convolutional neural networks. *ICML*.

[28] Liu, Z. et al. (2022). A ConvNet for the 2020s. *CVPR*.

[29] Dai, J. et al. (2017). Deformable convolutional networks. *ICCV*.

[30] Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780.

[31] Cho, K. et al. (2014). Learning phrase representations using RNN encoder-decoder. *EMNLP*.

[32] Gu, A. & Dao, T. (2024). Mamba : linear-time sequence modeling with selective state spaces. *ICLR*.

[33] Beck, M. et al. (2024). xLSTM : extended Long Short-Term Memory. *arXiv :2405.04517*.

[34] Sutskever, I., Vinyals, O. & Le, Q.V. (2014). Sequence to sequence learning with neural networks. *NeurIPS*.

[35] Luong, M.-T., Pham, H. & Manning, C.D. (2015). Effective approaches to attention-based neural machine translation. *EMNLP*.

[36] Vaswani, A. et al. (2017). Attention is all you need. *NeurIPS*.

[37] Dao, T. et al. (2022). FlashAttention : fast and memory-efficient exact attention. *NeurIPS*.

[38] Child, R. et al. (2019). Generating long sequences with sparse transformers. *arXiv :1904.10509*.

[39] Shazeer, N. (2019). Fast transformer decoding : one write-head is all you need. *arXiv :1911.02150*.

[40] Ye, W. et al. (2024). Differential Transformer. *arXiv :2410.05258*.

[41] Choromanski, K. et al. (2021). Rethinking attention with Performers. *ICLR*.