Apprentissage Géométrique

Notes de Cours

Master M2 — 2025–2026

Yaë Ulrich Gaba

« La géométrie est l'art de raisonner juste sur des figures fausses. »
— Henri Poincaré

25 mars 2026

Table des matières

Préface

Pourquoi ce cours ?

L'apprentissage profond a révolutionné de nombreux domaines de l'intelligence artificielle : vision par ordinateur, traitement du langage naturel, reconnaissance vocale. Cependant, les architectures classiques — réseaux de neurones entièrement connectés, réseaux convolutifs sur grilles régulières — reposent sur des hypothèses structurelles fortes : les données vivent sur des grilles euclidiennes régulières.

Or, une quantité croissante de problèmes fondamentaux en sciences et en ingénierie fait intervenir des données à structure **non euclidienne** :

- **Graphes** : réseaux sociaux, molécules, réseaux de transport, graphes de connaissances.
- **Variétés** : surfaces 3D, espaces de configurations en robotique, données sur la sphère (météorologie, astrophysique).
- **Nuages de points** : données LiDAR, reconstruction 3D, géométrie moléculaire.
- **Complexes simpliciaux** : modélisation de relations d'ordre supérieur, topologie algébrique appliquée.

L'**apprentissage géométrique profond** (*Geometric Deep Learning*, GDL) est le cadre unificateur qui étend les principes de l'apprentissage profond aux domaines non euclidiens, en s'appuyant sur les **symétries** et les **invariances** intrinsèques des données.

Le principe unificateur : les symétries

Le fil conducteur de ce cours est le **principe d'équivariance**. L'idée fondamentale, héritée de la physique théorique et du programme d'Erlangen de Felix Klein, est que les structures mathématiques intéressantes sont caractérisées par leurs **groupes de symétries**.

Le programme d'Erlangen pour le deep learning

De même que Klein a proposé de classifier les géométries par leurs groupes de transformation, l'apprentissage géométrique profond propose de classifier les architectures de réseaux de neurones par les symétries qu'elles respectent :

Domaine	**Symétrie**	**Architecture**
Grille régulière	Translation	CNN
Ensemble	Permutation	Deep Sets
Graphe	Permutation des nœuds	GNN
Sphère	Rotations SO(3)	Spherical CNN
Espace 3D	SE(3)	SE(3)-Transformer

Public visé

Ce cours s'adresse aux étudiants de **Master 2** et de **doctorat** en mathématiques appliquées, informatique, ou physique, ayant des prérequis en :

- **Algèbre linéaire** : espaces vectoriels, valeurs propres, décomposition spectrale.
- **Apprentissage profond** : réseaux de neurones, rétropropagation, optimisation par descente de gradient.
- **Probabilités et statistiques** : mesures de probabilité, espérance, estimation.
- **Programmation Python** : maîtrise de PyTorch ou d'un framework équivalent.

Des notions de théorie des groupes, de géométrie différentielle et de topologie algébrique sont souhaitables mais seront introduites dans le cours.

Organisation du cours

Le cours est structuré en onze chapitres, progressant des fondations mathématiques vers les applications et la recherche de pointe :

1. **Motivations** : pourquoi la géométrie en apprentissage profond.
2. **Théorie des groupes** : symétries, représentations, équivariance.
3. **Réseaux de neurones sur graphes (GNN)** : passage de messages, agrégation.
4. **Variantes de GNN** : GAT, GIN, GraphSAGE.
5. **Méthodes spectrales** : laplacien de graphe, convolutions spectrales.
6. **Apprentissage sur les variétés** : géométrie différentielle, opérateurs intrinsèques.
7. **Réseaux équivariants et invariants** : formalisme général.
8. **Nuages de points** : PointNet et ses extensions.
9. **Applications** : chimie, physique, biologie, vision 3D.
10. **Connexions avec la TDA** : homologie persistante, représentations topologiques.
11. **Directions de recherche** : tendances actuelles et problèmes ouverts.

Conventions et notations

Notation	Signification
$G = (V, E)$	Graphe avec sommets V et arêtes E
$\mathbf{A}$	Matrice d'adjacence
$\mathbf{D}$	Matrice des degrés
$\mathbf{L} = \mathbf{D} - \mathbf{A}$	Laplacien combinatoire
$\tilde{\mathbf{L}} = \mathbf{D}^{-1/2}\mathbf{L}\mathbf{D}^{-1/2}$	Laplacien normalisé
$\mathbf{X} \in \mathbb{R}^{n \times d}$	Matrice des attributs des nœuds
$\mathbf{h}_v^{(\ell)}$	Représentation du nœud v à la couche ℓ
$\mathcal{N}(v)$	Voisinage du nœud v
$\rho : G \to \mathrm{GL}(V)$	Représentation d'un groupe G
$\mathcal{M}$	Variété différentiable
$T_p\mathcal{M}$	Espace tangent en p
$\|\cdot\|$	Norme
$\langle \cdot, \cdot \rangle$	Produit scalaire
σ	Fonction d'activation (ReLU, etc.)
$\bigoplus$	Agrégation (somme, moyenne, max)

Le cadre 5G de Bronstein et al.

Le cadre unificateur de l'apprentissage géométrique profond repose sur cinq domaines fondamentaux, les « 5G » :

Définition 0.1 (Les 5G)**.** Les cinq domaines de l'apprentissage géométrique profond sont :

1. **Grids** (Grilles) : données sur grilles régulières (images, séries temporelles).
2. **Groups** (Groupes) : symétries globales agissant sur les données.
3. **Graphs** (Graphes) : données relationnelles avec invariance par permutation.
4. **Geodesics** (Géodésiques) : données sur des variétés courbes.
5. **Gauges** (Jauges) : fibrés et connexions pour le transport parallèle.

Remarque 0.2 (Unification). Chacun de ces domaines peut être vu comme un cas particulier d'un cadre général où un **groupe de symétries** $\mathfrak{G}$ agit sur un **domaine** Ω et où l'on cherche des fonctions $f : \mathcal{X}(\Omega) \to \mathcal{Y}$ qui sont **équivariantes** par rapport à l'action de $\mathfrak{G}$:

$$f(\rho_{\mathcal{X}}(g) \cdot x) = \rho_{\mathcal{Y}}(g) \cdot f(x), \quad \forall g \in \mathfrak{G}.$$

Ressources et références principales

Ce cours s'appuie sur plusieurs ouvrages et articles fondateurs :

- M. M. Bronstein, J. Bruna, T. Cohen, P. Veličković, *Geometric Deep Learning : Grids, Groups, Graphs, Geodesics, and Gauges*, 2021.

- T. N. Kipf et M. Welling, *Semi-Supervised Classification with Graph Convolutional Networks*, ICLR 2017.
- P. W. Battaglia et al., *Relational Inductive Biases, Deep Learning, and Graph Networks*, 2018.
- C. R. Qi et al., *PointNet : Deep Learning on Point Sets for 3D Classification and Segmentation*, CVPR 2017.
- N. Thomas et al., *Tensor Field Networks*, 2018.
- V. G. Satorras et al., *E(n) Equivariant Graph Neural Networks*, ICML 2021.

Logiciels utilisés

Les travaux pratiques utilisent principalement :

- **PyTorch** (`torch`) — framework d'apprentissage profond.
- **PyTorch Geometric** (`torch_geometric`) — bibliothèque pour l'apprentissage sur graphes.
- **NetworkX** — manipulation et visualisation de graphes.
- **Open3D** — traitement de nuages de points 3D.
- **GUDHI / Ripser** — calculs d'homologie persistante.

Vérification de l'environnement

```python
import torch
import torch_geometric
import networkx as nx

print(f"PyTorch version: {torch.__version__}")
print(f"PyG version: {torch_geometric.__version__}")
print(f"CUDA disponible: {torch.cuda.is_available()}")

# Test rapide : creation d'un graphe PyG
from torch_geometric.data import Data
edge_index = torch.tensor([[0, 1, 1, 2],
                           [1, 0, 2, 1]], dtype=torch.long)
x = torch.randn(3, 16)  # 3 noeuds, 16 attributs
data = Data(x=x, edge_index=edge_index)
print(f"Graphe: {data.num_nodes} noeuds, {data.num_edges} aretes")
```

Remerciements

Ce cours a bénéficié des contributions de nombreux chercheurs et étudiants. Nous remercions particulièrement les auteurs du *Geometric Deep Learning Blueprint* pour avoir établi le cadre conceptuel unificateur qui structure l'ensemble de cet enseignement.

[Author Name]
Mars 2026

Chapitre 1

Motivations — Géométrie en Deep Learning

Les réseaux de neurones classiques — perceptrons, CNN, RNN — sont conçus pour des données vivant sur des grilles régulières : images sur des pixels, séries temporelles sur des pas de temps uniformément espacés. Mais que faire quand les données vivent sur un graphe (réseaux sociaux, molécules), sur une surface courbe (modèles 3D), ou sur un groupe de symétries (poses, rotations) ? C'est la question à laquelle répond l'*apprentissage géométrique* : une révolution architecturale, portée par les travaux de Bronstein, Bruna, Cohen, et d'autres, qui replace la géométrie au cœur du deep learning.

1.1 Les limites des architectures classiques

Les réseaux de neurones convolutifs (CNN) ont connu un succès spectaculaire en vision par ordinateur, traitement du signal et reconnaissance vocale. Leur puissance provient d'un **biais inductif** fondamental : l'exploitation de la structure de grille régulière des données.

Définition 1.1 (Biais inductif)**.** Un **biais inductif** est un ensemble d'hypothèses a priori incorporées dans l'architecture d'un modèle pour guider l'apprentissage. Pour un CNN :

1. **Localité** : chaque neurone ne regarde qu'un voisinage local.
2. **Partage des poids** : le même filtre est appliqué partout.
3. **Invariance par translation** : la détection de motifs est indépendante de la position.

Cependant, de nombreuses données du monde réel n'ont **pas** de structure de grille :

Exemple 1.2 (Données non euclidiennes)**.**

1. Une **molécule** est un graphe d'atomes reliés par des liaisons chimiques.
2. Un **réseau social** est un graphe de personnes reliées par des relations.
3. La **surface de la Terre** est une sphère, pas un plan.
4. Un **nuage de points** 3D issu d'un capteur LiDAR n'a pas d'ordre canonique.

Pourquoi ne pas simplement vectoriser ?

On pourrait être tenté de représenter un graphe par sa matrice d'adjacence aplatie en vecteur et d'utiliser un MLP. Cette approche échoue car :

- La représentation dépend de la **numérotation arbitraire** des nœuds.
- Le nombre de paramètres explose : $\mathcal{O}(n^2)$ pour n nœuds.
- Aucune **généralisation** à des graphes de tailles différentes.

1.2 La géométrie comme principe organisateur

1.2.1 Le programme d'Erlangen

En 1872, Felix Klein proposa dans son *Erlanger Programm* de classifier les géométries par leurs groupes de symétries. Cette idée profonde trouve une application directe en apprentissage profond.

Théorème 1.3 (Programme d'Erlangen — version informelle)**.** *Une géométrie est entièrement déterminée par son* ***groupe de transformations*** G *et les propriétés qui sont* ***invariantes*** *sous l'action de* G*.*

Exemple 1.4 (Géométries et leurs groupes)**.**

Géométrie	Groupe	Invariants
Euclidienne	Isométries $\mathrm{E}(n)$	Distances, angles
Affine	Transformations affines	Parallélisme, rapports
Projective	Transformations projectives	Rapport biharmonique

1.2.2 Des CNN aux GNN : le passage de la grille au graphe

Un CNN peut être vu comme un cas particulier de réseau opérant sur un graphe régulier (la grille de pixels), avec le groupe de translation $\mathbb{Z}^2$ comme groupe de symétries.

Convolution discrète sur grille vs. graphe

Convolution sur grille :

$$(f * g)(x) = \sum_{y \in \mathbb{Z}^2} f(y)\, g(x - y)$$

Convolution sur graphe (passage de messages) :

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

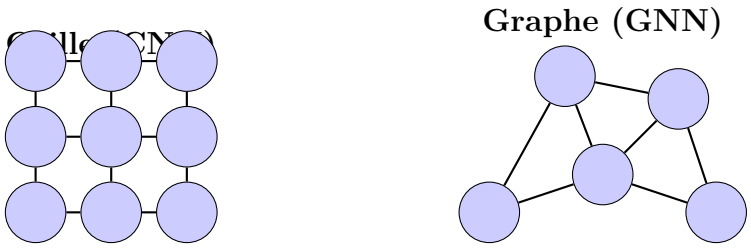

FIG. 1.1 : Passage d'une grille régulière (CNN) à un graphe irrégulier (GNN).

1.3 Le cadre unificateur : Geometric Deep Learning

1.3.1 Les cinq G

Bronstein et al. (2021) ont proposé un cadre unificateur organisé autour de cinq domaines fondamentaux :

Définition 1.5 (Domaines du GDL)**.**

1. **Grilles** ($\Omega = \mathbb{Z}^d$) : symétrie de translation.
2. **Groupes** ($\Omega = G$) : symétrie par le groupe lui-même.
3. **Graphes** ($\Omega = (V, E)$) : symétrie par permutation S_n.
4. **Géodésiques** ($\Omega = \mathcal{M}$) : invariance par isométries.
5. **Jauges** ($\Omega = \mathcal{M}$ avec fibré) : équivariance de jauge.

1.3.2 Le principe général d'équivariance

Théorème 1.6 (Principe d'équivariance)**.** *Soit $\mathfrak{G}$ un groupe agissant sur un espace de signaux $\mathcal{X}(\Omega)$ via la représentation $\rho_{\mathcal{X}}$ et sur un espace cible $\mathcal{Y}$ via $\rho_{\mathcal{Y}}$. Une fonction $f : \mathcal{X}(\Omega) \to \mathcal{Y}$ est **équivariante** si :*

$$f\big(\rho_{\mathcal{X}}(g) \cdot x\big) = \rho_{\mathcal{Y}}(g) \cdot f(x), \quad \forall g \in \mathfrak{G},\ \forall x \in \mathcal{X}(\Omega).$$

*Lorsque $\rho_{\mathcal{Y}}$ est la représentation triviale, f est **invariante**.*

Équivariance en images

Considérons la classification d'images :

- Les couches convolutives sont **équivariantes** par translation : si l'entrée se déplace, les cartes d'activation se déplacent aussi.
- Le pooling global produit une sortie **invariante** : le résultat ne dépend pas de la position de l'objet.

Le même principe s'applique aux GNN : les couches de passage de messages sont équivariantes par permutation des nœuds, et le readout global est invariant.

1.4 L'importance de l'invariance par permutation

1.4.1 Formalisation

Pour un graphe à n nœuds, une **permutation** $\pi \in S_n$ réordonne les nœuds. Cela agit sur :

- La matrice d'adjacence : $\mathbf{A} \mapsto \mathbf{P}_\pi \mathbf{A} \mathbf{P}_\pi^\top$
- Les attributs : $\mathbf{X} \mapsto \mathbf{P}_\pi \mathbf{X}$

où $\mathbf{P}_\pi$ est la matrice de permutation associée à π.

Définition 1.7 (Fonction invariante / équivariante sur graphes)**.** Une fonction au niveau du graphe $f : (\mathbf{A}, \mathbf{X}) \mapsto y$ est **invariante par permutation** si :

$$f(\mathbf{P}_\pi \mathbf{A} \mathbf{P}_\pi^\top, \mathbf{P}_\pi \mathbf{X}) = f(\mathbf{A}, \mathbf{X}), \quad \forall \pi \in S_n.$$

Une fonction au niveau des nœuds $f : (\mathbf{A}, \mathbf{X}) \mapsto \mathbf{H}$ est **équivariante par permutation** si :

$$f(\mathbf{P}_\pi \mathbf{A} \mathbf{P}_\pi^\top, \mathbf{P}_\pi \mathbf{X}) = \mathbf{P}_\pi f(\mathbf{A}, \mathbf{X}), \quad \forall \pi \in S_n.$$

1.4.2 Deep Sets : le cas le plus simple

Théorème 1.8 (Zaheer et al., 2017)**.** *Une fonction $f : 2^{\mathcal{X}} \to \mathbb{R}$ est invariante par permutation et continue si et seulement si elle peut s'écrire sous la forme :*

$$f(\{x_1, \ldots, x_n\}) = \rho\left(\sum_{i=1}^{n} \phi(x_i)\right)$$

pour des fonctions continues $\phi : \mathcal{X} \to \mathbb{R}^d$ et $\rho : \mathbb{R}^d \to \mathbb{R}$.

Implémentation Deep Sets en PyTorch

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

class DeepSets(nn.Module):
    """Modele invariant par permutation pour ensembles."""
    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        # phi : encodeur par element
        self.phi = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
        )
        # rho : decodeur apres aggregation
        self.rho = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, output_dim),
```

```
        )

    def forward(self, x):
        # x: (batch, n_elements, input_dim)
        h = self.phi(x)            # (batch, n_elements, hidden_dim)
        h = h.sum(dim=1)           # (batch, hidden_dim) -- aggregation
        return self.rho(h)         # (batch, output_dim)

# Test
model = DeepSets(input_dim=3, hidden_dim=64, output_dim=10)
x = torch.randn(8, 20, 3)  # batch=8, 20 points, dim=3
out = model(x)
print(f"Sortie: {out.shape}")  # (8, 10)

# Verification de l'invariance par permutation
perm = torch.randperm(20)
x_perm = x[:, perm, :]
out_perm = model(x_perm)
print(f"Invariance: {torch.allclose(out, out_perm, atol=1e-5)}")
```

1.5 Exemples motivants

1.5.1 Prédiction de propriétés moléculaires

La prédiction de propriétés moléculaires est l'une des applications phares du GDL. Une molécule est naturellement représentée comme un graphe :

- **Nœuds** : atomes, avec attributs (type atomique, charge, etc.).
- **Arêtes** : liaisons chimiques, avec attributs (ordre de liaison, etc.).

Proposition 1.9 (Propriétés souhaitées)**.** Un modèle pour la prédiction moléculaire doit être :

1. **Invariant par permutation** des atomes (pas de numérotation canonique).
2. **Invariant par rotation et translation** (énergie indépendante de l'orientation).
3. **Extensif / intensif** selon la propriété (énergie vs. gap HOMO-LUMO).

1.5.2 Météorologie sur la sphère

Les données météorologiques vivent naturellement sur la sphère S^2. Les CNN classiques ne peuvent pas être directement appliqués car :

- Les projections planes introduisent des **distorsions** (Mercator, etc.).
- La grille de pixels n'est pas uniforme sur la sphère.
- Les symétries physiques (rotations) ne sont pas respectées.

Remarque 1.10 (Spherical CNNs). Les *Spherical CNNs* (Cohen et al., 2018) définissent des convolutions directement sur S^2 en utilisant les harmoniques sphériques comme base spectrale, respectant ainsi l'équivariance par SO(3).

1.5.3 Dynamique des particules

La simulation de systèmes de particules (fluides, matériaux granulaires) bénéficie du GDL car :

- Les particules forment un ensemble non ordonné.
- Les lois physiques sont équivariantes par SE(3).
- Les interactions sont locales (graphe de voisinage).

1.6 Panorama des architectures géométriques

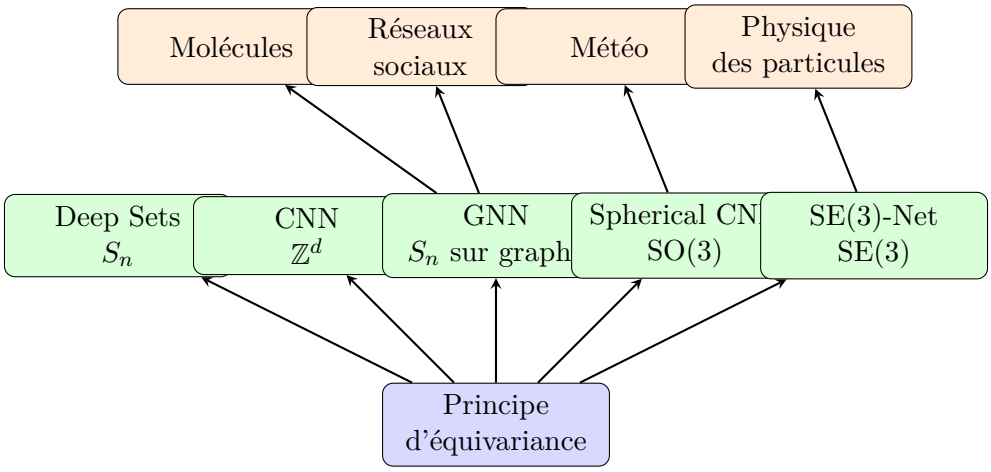

FIG. 1.2 : Panorama des architectures géométriques et de leurs applications.

1.7 Pourquoi les symétries améliorent l'apprentissage

L'exploitation des symétries améliore l'apprentissage de trois manières complémentaires :

Théorème 1.11 (Avantages de l'équivariance)**.** *Soit $\mathfrak{G}$ le groupe de symétries d'un problème. Un modèle équivariant par $\mathfrak{G}$ bénéficie de :*

1. ***Réduction de la complexité d'échantillonnage** : le nombre d'exemples nécessaires est réduit d'un facteur $|\mathfrak{G}|$ (ou de la dimension du groupe pour les groupes continus).*
2. ***Amélioration de la généralisation** : la garantie de généralisation à toutes les transformations de $\mathfrak{G}$ est « gratuite ».*
3. ***Partage des poids** : le nombre de paramètres est réduit, évitant le surapprentissage.*

Proposition 1.12 (Borne de généralisation)**.** Pour un modèle paramétré par $\theta \in \Theta$, la complexité de Rademacher de la classe de fonctions équivariantes $\mathcal{F}_{\text{equiv}}$ satisfait :

$$\mathfrak{R}_n(\mathcal{F}_{\text{equiv}}) \leq \frac{\mathfrak{R}_n(\mathcal{F})}{\sqrt{|\mathfrak{G}|}}$$

où $\mathcal{F}$ est la classe non contrainte.

1.8 De la théorie à la pratique

Premier GNN avec PyTorch Geometric

```python
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv, global_mean_pool
from torch_geometric.datasets import TUDataset
from torch_geometric.loader import DataLoader

class SimpleGNN(torch.nn.Module):
    def __init__(self, num_features, num_classes):
        super().__init__()
        self.conv1 = GCNConv(num_features, 64)
        self.conv2 = GCNConv(64, 64)
        self.lin = torch.nn.Linear(64, num_classes)

    def forward(self, data):
        x, edge_index, batch = data.x, data.edge_index, data.batch
        # Couches equivariantes par permutation
        x = F.relu(self.conv1(x, edge_index))
        x = F.relu(self.conv2(x, edge_index))
        # Readout invariant par permutation
        x = global_mean_pool(x, batch)
        return self.lin(x)

# Charger un dataset de classification de graphes
dataset = TUDataset(root='/tmp/MUTAG', name='MUTAG')
loader = DataLoader(dataset, batch_size=32, shuffle=True)

model = SimpleGNN(dataset.num_features, dataset.num_classes)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# Boucle d'entrainement
for epoch in range(50):
    total_loss = 0
    for data in loader:
        optimizer.zero_grad()
        out = model(data)
        loss = F.cross_entropy(out, data.y)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    if (epoch + 1) % 10 == 0:
        print(f"Epoch {epoch+1}, Loss: {total_loss/len(loader):.4f}")
```

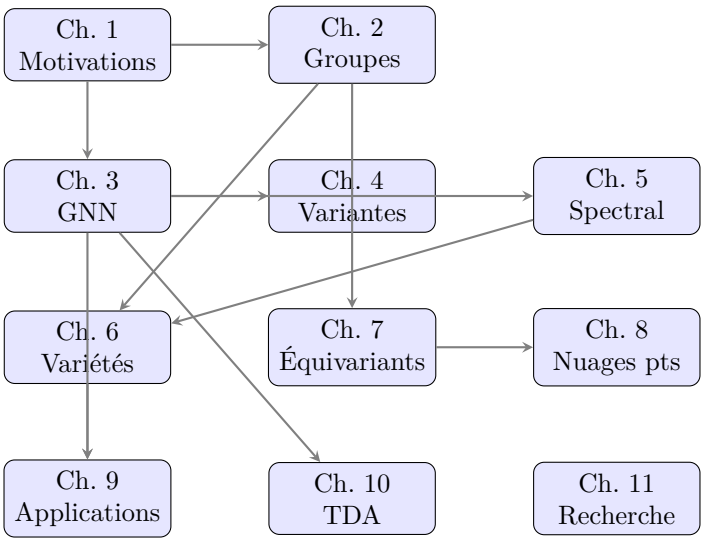

FIG. 1.3 : Graphe de dépendances entre les chapitres.

1.9 Plan du cours et dépendances

Exercices

Exercice 1.1 (Invariance vs. équivariance)**.** Soit $f : \mathbb{R}^{n\times d} \to \mathbb{R}^{n\times d'}$ définie par $f(\mathbf{X}) = \sigma(\mathbf{X}\mathbf{W})$ où $\mathbf{W} \in \mathbb{R}^{d\times d'}$.

1. Montrer que f est équivariante par permutation des lignes de $\mathbf{X}$.
2. Proposer une modification de f pour obtenir une fonction invariante $g : \mathbb{R}^{n\times d} \to \mathbb{R}^{d'}$.
3. Implémenter les deux fonctions en PyTorch et vérifier expérimentalement.

Exercice 1.2 (Limites de la vectorisation)**.** Considérons un graphe aléatoire $G \sim$ Erdős-Rényi(n, p).

1. Combien de représentations matricielles différentes $\mathbf{A}$ correspondent au même graphe (isomorphe) ?
2. En déduire pourquoi un MLP appliqué à vec$(\mathbf{A})$ a une complexité d'échantillonnage exponentielle.
3. Montrer qu'un GNN évite ce problème par construction.

Exercice 1.3 (Deep Sets pour l'approximation de fonctions symétriques)**.** Implémenter un modèle Deep Sets pour approximer la fonction $f(\{x_1, \ldots, x_n\}) = \max_i x_i - \min_i x_i$ (l'étendue d'un ensemble).

1. Entraîner le modèle sur des ensembles de taille variable $n \in [5, 50]$.
2. Évaluer la généralisation à $n = 100$.
3. Comparer avec une agrégation par max au lieu de la somme.

Chapitre 2

Théorie des Groupes pour le Deep Learning

Les symétries sont partout : un visage est (approximativement) symétrique par réflexion, un cristal l'est par rotation, les lois de la physique sont invariantes par translation dans l'espace et le temps. La théorie des groupes, née des travaux de Galois et d'Abel au XIX^{e} siècle, est le langage mathématique des symétries. Et en deep learning, exploiter les symétries d'un problème — les intégrer comme *biais inductif* dans l'architecture du réseau — est devenu l'une des idées les plus puissantes de la dernière décennie.

2.1 Introduction à la théorie des groupes

La théorie des groupes fournit le langage mathématique pour décrire les symétries. Ce chapitre introduit les concepts fondamentaux nécessaires à la compréhension de l'apprentissage géométrique profond.

Définition 2.1 (Groupe)**.** Un **groupe** est un ensemble G muni d'une opération binaire $\cdot : G \times G \to G$ satisfaisant :

1. **Associativité** : $(g \cdot h) \cdot k = g \cdot (h \cdot k)$ pour tous $g, h, k \in G$.
2. **Élément neutre** : il existe $e \in G$ tel que $e \cdot g = g \cdot e = g$ pour tout $g \in G$.
3. **Inverse** : pour tout $g \in G$, il existe $g^{-1} \in G$ tel que $g \cdot g^{-1} = g^{-1} \cdot g = e$.

Exemple 2.2 (Groupes fondamentaux en GDL)**.**

1. **Groupe symétrique** S_n : permutations de n éléments. $|S_n| = n!$.
2. **Groupe cyclique** $\mathbb{Z}_n = \mathbb{Z}/n\mathbb{Z}$: rotations discrètes.
3. **Groupe orthogonal** $\mathrm{O}(n)$: matrices $\mathbf{Q} \in \mathbb{R}^{n \times n}$ telles que $\mathbf{Q}^\top \mathbf{Q} = \mathbf{I}$.
4. **Groupe spécial orthogonal** $\mathrm{SO}(n)$: rotations, $\det(\mathbf{Q}) = +1$.
5. **Groupe euclidien** $\mathrm{E}(n) = \mathrm{O}(n) \ltimes \mathbb{R}^n$: rotations, réflexions et translations.
6. **Groupe spécial euclidien** $\mathrm{SE}(n) = \mathrm{SO}(n) \ltimes \mathbb{R}^n$: rotations et translations (mouvements rigides).

2.1.1 Sous-groupes et homomorphismes

Définition 2.3 (Sous-groupe)**.** Un sous-ensemble $H \subseteq G$ est un **sous-groupe** de G (noté $H \leq G$) si H est lui-même un groupe pour l'opération de G.

Définition 2.4 (Homomorphisme de groupes)**.** Un **homomorphisme** de groupes est une application $\varphi : G \to H$ telle que $\varphi(g_1 \cdot g_2) = \varphi(g_1) \cdot \varphi(g_2)$ pour tous $g_1, g_2 \in G$. Si φ est bijectif, c'est un **isomorphisme**.

Proposition 2.5 (Propriétés des homomorphismes)**.** Soit $\varphi : G \to H$ un homomorphisme. Alors :

1. $\varphi(e_G) = e_H$.
2. $\varphi(g^{-1}) = \varphi(g)^{-1}$ pour tout $g \in G$.
3. $\ker(\varphi) = \{g \in G : \varphi(g) = e_H\}$ est un sous-groupe normal de G.

2.2 Actions de groupe

Définition 2.6 (Action de groupe)**.** Une **action** (gauche) d'un groupe G sur un ensemble X est une application $\cdot : G \times X \to X$ telle que :

1. $e \cdot x = x$ pour tout $x \in X$.
2. $(g \cdot h) \cdot x = g \cdot (h \cdot x)$ pour tous $g, h \in G$ et $x \in X$.

Exemple 2.7 (Actions en deep learning)**.**
1. S_n agit sur $\mathbb{R}^n$ par permutation des coordonnées : $\pi \cdot (x_1, \ldots, x_n) = (x_{\pi^{-1}(1)}, \ldots, x_{\pi^{-1}(n)})$.
2. $\mathrm{SO}(3)$ agit sur $\mathbb{R}^3$ par rotation : $R \cdot \mathbf{x} = R\mathbf{x}$.
3. $\mathbb{Z}^2$ agit sur les images par translation : $\mathbf{t} \cdot f(\mathbf{x}) = f(\mathbf{x} - \mathbf{t})$.

Définition 2.8 (Orbite et stabilisateur)**.** Soit G agissant sur X. Pour $x \in X$:

- L'**orbite** de x est $G \cdot x = \{g \cdot x : g \in G\}$.
- Le **stabilisateur** de x est $G_x = \{g \in G : g \cdot x = x\}$.

Théorème 2.9 (Théorème orbite-stabilisateur)**.** *Pour un groupe fini G agissant sur X et $x \in X$:*

$$|G| = |G \cdot x| \cdot |G_x|$$

2.3 Représentations de groupes

Définition 2.10 (Représentation)**.** Une **représentation** d'un groupe G sur un espace vectoriel V est un homomorphisme $\rho : G \to \mathrm{GL}(V)$, c'est-à-dire une application qui associe à chaque $g \in G$ une transformation linéaire inversible $\rho(g) : V \to V$ telle que $\rho(g_1 g_2) = \rho(g_1)\rho(g_2)$.

Exemple 2.11 (Représentations courantes)**.**
1. **Représentation triviale** : $\rho(g) = \mathrm{Id}$ pour tout g.

2. **Représentation par permutation** : $\rho(\pi) = \mathbf{P}_\pi$ (matrice de permutation).
3. **Représentation standard de** SO(3) : $\rho(R) = R$ (la matrice de rotation elle-même).
4. **Représentation régulière** : $[\rho(g)f](h) = f(g^{-1}h)$.

2.3.1 Représentations irréductibles

Définition 2.12 (Représentation irréductible)**.** Une représentation $\rho : G \to \mathrm{GL}(V)$ est **irréductible** si les seuls sous-espaces de V invariants sous l'action de G sont $\{0\}$ et V lui-même.

Théorème 2.13 (Décomposition de Maschke)**.** *Toute représentation de dimension finie d'un groupe fini (sur $\mathbb{R}$ ou $\mathbb{C}$) peut être décomposée en somme directe de représentations irréductibles :*

$$V = V_1 \oplus V_2 \oplus \cdots \oplus V_k$$

où chaque V_i est un sous-espace irréductible.

Représentations irréductibles de SO(3)

Les représentations irréductibles de SO(3) sont indexées par $\ell \in \{0, 1, 2, \ldots\}$ et ont dimension $2\ell + 1$:

- $\ell = 0$: scalaires (dimension 1) — représentation triviale.
- $\ell = 1$: vecteurs (dimension 3) — représentation standard.
- $\ell = 2$: tenseurs symétriques sans trace (dimension 5).

Les matrices de représentation sont les **matrices de Wigner** $D^\ell_{mm'}(R)$, reliées aux harmoniques sphériques :

$$Y_\ell^m(R^{-1}\hat{\mathbf{r}}) = \sum_{m'=-\ell}^{\ell} D^\ell_{mm'}(R)\, Y_\ell^{m'}(\hat{\mathbf{r}})$$

2.4 Équivariance et invariance

Définition 2.14 (Application équivariante)**.** Soient $\rho_1 : G \to \mathrm{GL}(V_1)$ et $\rho_2 : G \to \mathrm{GL}(V_2)$ deux représentations. Une application linéaire $T : V_1 \to V_2$ est **équivariante** (ou **entrelacement**) si :

$$T \circ \rho_1(g) = \rho_2(g) \circ T, \quad \forall g \in G.$$

Lorsque ρ_2 est la représentation triviale, T est **invariante**.

Théorème 2.15 (Lemme de Schur)**.** *Soient ρ_1 et ρ_2 deux représentations irréductibles de G et $T : V_1 \to V_2$ un entrelacement :*

1. *Si $\rho_1 \not\cong \rho_2$, alors $T = 0$.*
2. *Si $\rho_1 \cong \rho_2$ (sur $\mathbb{C}$), alors $T = \lambda\,\mathrm{Id}$ pour un certain $\lambda \in \mathbb{C}$.*

Importance du lemme de Schur

Le lemme de Schur est fondamental car il **contraint fortement** la forme des couches équivariantes dans un réseau de neurones. Si l'on impose l'équivariance, l'espace des applications linéaires autorisées est drastiquement réduit, ce qui justifie le partage des poids.

2.5 Produit tensoriel de représentations

Définition 2.16 (Produit tensoriel)**.** Le **produit tensoriel** de deux représentations $\rho_1 : G \to \mathrm{GL}(V_1)$ et $\rho_2 : G \to \mathrm{GL}(V_2)$ est la représentation $\rho_1 \otimes \rho_2 : G \to \mathrm{GL}(V_1 \otimes V_2)$ définie par :

$$(\rho_1 \otimes \rho_2)(g)(v_1 \otimes v_2) = \rho_1(g)v_1 \otimes \rho_2(g)v_2.$$

Théorème 2.17 (Décomposition de Clebsch–Gordan pour SO(3))**.** *Le produit tensoriel de deux représentations irréductibles de* SO(3) *se décompose selon :*

$$D^{\ell_1} \otimes D^{\ell_2} = \bigoplus_{\ell=|\ell_1-\ell_2|}^{\ell_1+\ell_2} D^{\ell}$$

Les coefficients de changement de base sont les ***coefficients de Clebsch–Gordan*** $C^{\ell m}_{\ell_1 m_1, \ell_2 m_2}$.

Produit tensoriel avec e3nn

```python
import torch
from e3nn import o3

# Representations irreductibles de SO(3)
irreps_1 = o3.Irreps("1x0e + 1x1o")  # scalaire + vecteur
irreps_2 = o3.Irreps("1x1o")          # vecteur

# Produit tensoriel
tp = o3.FullTensorProduct(irreps_1, irreps_2)
print(f"Entree 1: {irreps_1}")
print(f"Entree 2: {irreps_2}")
print(f"Sortie: {tp.irreps_out}")
# 0e x 1o -> 1o, 1o x 1o -> 0e + 1e + 2e

x1 = irreps_1.randn(5, -1)  # batch de 5
x2 = irreps_2.randn(5, -1)
out = tp(x1, x2)
print(f"Shape sortie: {out.shape}")
```

2.6 Groupes de Lie

Définition 2.18 (Groupe de Lie)**.** Un **groupe de Lie** est un groupe G qui est aussi une variété différentiable, de sorte que les opérations de multiplication $(g, h) \mapsto gh$ et d'inversion $g \mapsto g^{-1}$ sont lisses.

Exemple 2.19 (Groupes de Lie en GDL)**.**

Groupe	Dimension	Compact ?	Usage
SO(2)	1	Oui	Rotations 2D
SO(3)	3	Oui	Rotations 3D
SE(3)	6	Non	Mouvements rigides
O(3)	3	Oui	Rotations + réflexions
$\mathrm{GL}(n, \mathbb{R})$	n^2	Non	Transformations linéaires

2.6.1 Algèbre de Lie

Définition 2.20 (Algèbre de Lie)**.** L'**algèbre de Lie** $\mathfrak{g}$ d'un groupe de Lie G est l'espace tangent à G en l'élément neutre e, muni du **crochet de Lie** $[\cdot, \cdot] : \mathfrak{g} \times \mathfrak{g} \to \mathfrak{g}$.

Application exponentielle

L'**application exponentielle** $\exp : \mathfrak{g} \to G$ relie l'algèbre de Lie au groupe :

$$\exp(X) = \sum_{k=0}^{\infty} \frac{X^k}{k!}$$

Pour SO(3), l'algèbre de Lie $\mathfrak{so}(3)$ est l'espace des matrices antisymétriques 3×3. La formule de Rodrigues donne :

$$\exp(\theta\, [\hat{\mathbf{n}}]_\times) = \mathbf{I} + \sin\theta\, [\hat{\mathbf{n}}]_\times + (1 - \cos\theta)\, [\hat{\mathbf{n}}]_\times^2$$

où $[\hat{\mathbf{n}}]_\times$ est la matrice antisymétrique associée au vecteur unitaire $\hat{\mathbf{n}}$.

2.7 Convolution sur les groupes

Définition 2.21 (Convolution de groupe)**.** Soit G un groupe localement compact muni d'une mesure de Haar μ. La **convolution de groupe** de deux fonctions $f, g : G \to \mathbb{R}$ est :

$$(f * g)(x) = \int_G f(y)\, g(y^{-1}x)\, d\mu(y).$$

Proposition 2.22 (Équivariance de la convolution)**.** La convolution de groupe est équivariante par translation à gauche : si $L_a f(x) = f(a^{-1}x)$, alors :

$$L_a(f * g) = (L_a f) * g.$$

C'est cette propriété qui justifie l'utilisation de la convolution comme couche fondamentale dans les réseaux équivariants.

Théorème 2.23 (CNN comme convolution sur $\mathbb{Z}^2$)**.** *Un réseau convolutif classique réalise une convolution de groupe sur $G = \mathbb{Z}^2$ (le groupe de translation discret). La première couche effectue une corrélation croisée :*

$$[f \star \psi](x) = \sum_{y \in \mathbb{Z}^2} f(y)\, \psi(y - x) = \sum_{y \in \mathbb{Z}^2} f(x + y)\, \psi(y)$$

qui est équivariante par translation de f.

2.8 Group Equivariant CNNs (G-CNNs)

Cohen et Welling (2016) ont généralisé les CNN en remplaçant le groupe de translation $\mathbb{Z}^2$ par un groupe plus grand G.

Définition 2.24 (G-CNN)**.** Un **G-CNN** est un réseau dont les couches réalisent des convolutions sur un groupe G. La première couche « souleve »(*lifts*) le signal de $\mathbb{Z}^2$ vers G :

$$[\text{lift}(f)](g) = \sum_{x \in \mathbb{Z}^2} f(x)\, \psi(g^{-1} \cdot x)$$

Les couches suivantes opèrent entièrement sur G :

$$[f * \psi](g) = \sum_{h \in G} f(h)\, \psi(h^{-1} g)$$

G-CNN pour le groupe $p4$ (rotations 90°)

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

class P4Conv(nn.Module):
    """Convolution equivariante par le groupe p4 (rotations
    ↪ 0/90/180/270)."""
    def __init__(self, in_channels, out_channels, kernel_size=3):
        super().__init__()
        self.weight = nn.Parameter(
            torch.randn(out_channels, in_channels, kernel_size,
            ↪ kernel_size)
        )
        nn.init.kaiming_normal_(self.weight)

    def _rotate_kernel(self, w, k):
        """Rotation du noyau de k*90 degres."""
        return torch.rot90(w, k, dims=[-2, -1])

    def forward(self, x):
        # x: (batch, C_in, H, W) ou (batch, C_in, 4, H, W)
        outputs = []
        for k in range(4):  # 4 rotations
            w_rot = self._rotate_kernel(self.weight, k)
            out = torch.nn.functional.conv2d(
                x if x.dim() == 4 else x.sum(dim=2),
                w_rot, padding=1
            )
            outputs.append(out)
        # Stack sur la dimension du groupe
        return torch.stack(outputs, dim=2)  # (batch, C_out, 4, H, W)

# Test
conv = P4Conv(3, 16)
x = torch.randn(2, 3, 32, 32)
```

```
out = conv(x)
print(f"Sortie: {out.shape}")  # (2, 16, 4, 32, 32)
```

2.9 Invariants et équivariants linéaires

Théorème 2.25 (Caractérisation des applications linéaires équivariantes)**.** *L'espace des applications linéaires $T : V_1 \to V_2$ qui sont équivariantes par rapport aux représentations ρ_1 et ρ_2 est :*

$$\mathrm{Hom}_G(V_1, V_2) = \{T \in \mathrm{Hom}(V_1, V_2) : T\rho_1(g) = \rho_2(g)T,\ \forall g \in G\}$$

Sa dimension est donnée par :

$$\dim \mathrm{Hom}_G(V_1, V_2) = \langle \chi_1, \chi_2 \rangle_G = \frac{1}{|G|} \sum_{g \in G} \overline{\chi_1(g)}\, \chi_2(g)$$

où $\chi_i(g) = \mathrm{tr}(\rho_i(g))$ est le ***caractère*** *de ρ_i.*

Corollaire 2.26 (Nombre de paramètres libres)**.** *Pour une couche équivariante entre deux représentations décomposées en irréductibles $V_1 = \bigoplus_i n_i V_i^{irr}$ et $V_2 = \bigoplus_j m_j V_j^{irr}$, le nombre de paramètres libres est :*

$$\sum_k n_k \cdot m_k$$

où la somme porte sur les types irréductibles communs.

Exercices

Exercice 2.1 (Groupes de symétrie de molécules)**.**

1. Déterminer le groupe de symétrie de la molécule d'eau H_2O (groupe C_{2v}).
2. Énumérer toutes les représentations irréductibles de C_{2v}.
3. Expliquer comment ces symétries contraignent un modèle de prédiction d'énergie moléculaire.

Exercice 2.2 (Représentations de S_3)**.**

1. Énumérer les six éléments du groupe symétrique S_3.
2. Trouver toutes les représentations irréductibles de S_3 (il y en a trois).
3. Vérifier la relation $\sum_i (\dim V_i)^2 = |G|$.
4. Décomposer la représentation par permutation standard $\mathbb{R}^3$ en irréductibles.

Exercice 2.3 (Formule de Rodrigues)**.**

1. Vérifier la formule de Rodrigues pour $\theta = \pi/2$ et $\hat{\mathbf{n}} = (0, 0, 1)$.
2. Montrer que $\exp(\theta[\hat{\mathbf{n}}]_\times) \in \mathrm{SO}(3)$.

3. Implémenter l'application exponentielle $\mathfrak{so}(3) \to \mathrm{SO}(3)$ en PyTorch et vérifier l'orthogonalité du résultat.

Exercice 2.4 (Convolution de groupe discrète)**.** Implémenter une convolution de groupe sur le groupe diédral D_4 (symétries du carré).

1. Énumérer les 8 éléments de D_4.
2. Implémenter la table de multiplication de D_4.
3. Implémenter la convolution de groupe et vérifier son équivariance.

Chapitre 3

Réseaux de Neurones sur Graphes

Les réseaux de neurones convolutifs ont révolutionné la vision par ordinateur, mais ils reposent sur une hypothèse implicite : les données vivent sur une grille régulière. Que faire lorsque les données sont un réseau social, une molécule, un maillage 3D ou un réseau de transport ? Ces structures sont naturellement représentées par des *graphes*, où la notion de voisinage est irrégulière et variable. L'idée des *Graph Neural Networks* (GNN), formalisée par Franco Scarselli en 2009 puis redécouverte et étendue par Thomas Kipf, Petar Veličković et bien d'autres à partir de 2016, est d'adapter le mécanisme de convolution à cette géométrie irrégulière : chaque nœud met à jour sa représentation en agrégeant les messages de ses voisins.

Ce paradigme de *passage de messages* est devenu le cadre unificateur de l'apprentissage sur graphes. Ce chapitre en construit les fondations.

3.1 Représentation des graphes

Définition 3.1 (Graphe attribué)**.** Un **graphe attribué** est un tuple $G = (V, E, \mathbf{X}, \mathbf{E})$ où :

- $V = \{1, \dots, n\}$ est l'ensemble des **nœuds**.
- $E \subseteq V \times V$ est l'ensemble des **arêtes**.
- $\mathbf{X} \in \mathbb{R}^{n \times d}$ est la matrice des **attributs de nœuds**.
- $\mathbf{E} \in \mathbb{R}^{|E| \times d_e}$ contient les **attributs d'arêtes**.

Définition 3.2 (Matrices associées)**.** Pour un graphe $G = (V, E)$:

- **Matrice d'adjacence** : $A_{ij} = 1$ si $(i, j) \in E$, 0 sinon.
- **Matrice des degrés** : $D_{ii} = \sum_j A_{ij}$ (diagonale).
- **Laplacien** : $\mathbf{L} = \mathbf{D} - \mathbf{A}$.
- **Laplacien normalisé** : $\tilde{\mathbf{L}} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$.

Représentation de graphes dans PyTorch Geometric

```python
import torch
from torch_geometric.data import Data

# Graphe avec 4 noeuds et 5 aretes (non dirige)
edge_index = torch.tensor([
    [0, 1, 1, 2, 2, 3, 3, 0, 1, 3],
    [1, 0, 2, 1, 3, 2, 0, 3, 3, 1]
], dtype=torch.long)

# Attributs des noeuds (4 noeuds, 3 attributs)
x = torch.tensor([
    [1.0, 0.0, 0.0],  # noeud 0 : carbone
    [0.0, 1.0, 0.0],  # noeud 1 : azote
    [0.0, 0.0, 1.0],  # noeud 2 : oxygene
    [1.0, 0.0, 0.0],  # noeud 3 : carbone
], dtype=torch.float)

data = Data(x=x, edge_index=edge_index)
print(f"Noeuds: {data.num_nodes}, Aretes: {data.num_edges}")
print(f"Attributs: {data.x.shape}")
print(f"Isole: {data.has_isolated_nodes()}")
print(f"Self-loops: {data.has_self_loops()}")
```

3.2 Le paradigme du passage de messages

3.2.1 Formalisation générale

Définition 3.3 (Réseau de neurones à passage de messages (MPNN))**.** Un **MPNN** (Gilmer et al., 2017) met à jour les représentations des nœuds par des itérations de la forme :

$$\mathbf{m}_v^{(\ell+1)} = \bigoplus_{u \in \mathcal{N}(v)} \psi^{(\ell)}\left(\mathbf{h}_v^{(\ell)}, \mathbf{h}_u^{(\ell)}, \mathbf{e}_{uv}\right) \quad \text{(agrégation des messages)} \tag{3.1}$$

$$\mathbf{h}_v^{(\ell+1)} = \phi^{(\ell)}\left(\mathbf{h}_v^{(\ell)}, \mathbf{m}_v^{(\ell+1)}\right) \quad \text{(mise à jour)} \tag{3.2}$$

où :

- $\psi^{(\ell)}$ est la **fonction de message**.
- $\bigoplus$ est un opérateur d'**agrégation** invariant par permutation (somme, moyenne, max).
- $\phi^{(\ell)}$ est la **fonction de mise à jour**.
- $\mathbf{h}_v^{(0)} = \mathbf{x}_v$ (attributs initiaux).

Théorème 3.4 (Équivariance des MPNN)**.** *Toute couche MPNN avec agrégation invariante par permutation est* ***équivariante par permutation des nœuds*** *: si $\pi \in S_n$ est une permutation, alors :*

$$\mathbf{h}_{\pi(v)}^{(\ell+1)}(\pi \cdot G) = \mathbf{h}_v^{(\ell+1)}(G)$$

où $\pi \cdot G$ est le graphe avec nœuds renumérotés par π.

Démonstration. L'agrégation $\bigoplus$ est invariante par permutation par hypothèse. Les fonctions ψ et ϕ sont appliquées individuellement à chaque nœud. Le voisinage $\mathcal{N}(\pi(v))$ dans $\pi \cdot G$ est exactement $\{\pi(u) : u \in \mathcal{N}(v)\}$ dans G. Donc :

$$\begin{aligned}\mathbf{m}_{\pi(v)}^{(\ell+1)}(\pi \cdot G) &= \bigoplus_{w \in \mathcal{N}(\pi(v))} \psi(\mathbf{h}_{\pi(v)}, \mathbf{h}_w, \mathbf{e}_{w,\pi(v)}) \\ &= \bigoplus_{u \in \mathcal{N}(v)} \psi(\mathbf{h}_v, \mathbf{h}_u, \mathbf{e}_{uv}) = \mathbf{m}_v^{(\ell+1)}(G).\end{aligned}$$

□

3.2.2 Illustration du passage de messages

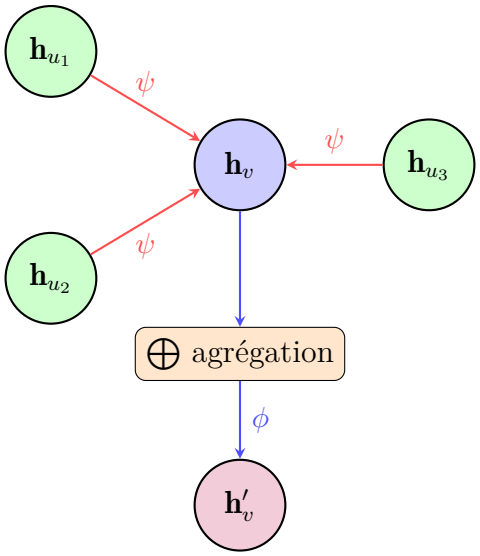

FIG. 3.1 : Schéma du passage de messages : messages, agrégation, mise à jour.

3.3 Graph Convolutional Network (GCN)

Définition 3.5 (Couche GCN — Kipf et Welling, 2017)**.** La couche GCN utilise la **règle de propagation** suivante :

$$\mathbf{H}^{(\ell+1)} = \sigma\left(\hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\,\hat{\mathbf{D}}^{-1/2}\,\mathbf{H}^{(\ell)}\,\mathbf{W}^{(\ell)}\right)$$

où :

- $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ (ajout de self-loops).
- $\hat{\mathbf{D}}_{ii} = \sum_j \hat{A}_{ij}$ (degrés ajustés).
- $\mathbf{W}^{(\ell)} \in \mathbb{R}^{d_\ell \times d_{\ell+1}}$ est la matrice de poids.
- σ est une activation non linéaire (ReLU).

Remarque 3.6 (Interprétation). Pour un nœud v, la couche GCN calcule :

$$\mathbf{h}_v^{(\ell+1)} = \sigma\left(\sum_{u \in \mathcal{N}(v)\cup\{v\}} \frac{1}{\sqrt{\hat{d}_v \hat{d}_u}} \mathbf{h}_u^{(\ell)} \mathbf{W}^{(\ell)}\right)$$

C'est une **moyenne pondérée** des représentations des voisins, suivie d'une transformation linéaire et d'une non-linéarité.

Proposition 3.7 (Lien avec le laplacien)**.** La couche GCN peut s'écrire :

$$\mathbf{H}^{(\ell+1)} = \sigma\Big((\mathbf{I} - \tilde{\mathbf{L}}_{\text{sym}})\mathbf{H}^{(\ell)}\mathbf{W}^{(\ell)}\Big)$$

où $\tilde{\mathbf{L}}_{\text{sym}}$ est le laplacien normalisé de $\hat{G}$ (graphe avec self-loops). La couche GCN réalise donc un **lissage laplacien** des attributs.

Implémentation GCN complète

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn import GCNConv, global_mean_pool
from torch_geometric.datasets import Planetoid
from torch_geometric.loader import DataLoader

class GCN(nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels,
                 num_layers=3, dropout=0.5):
        super().__init__()
        self.convs = nn.ModuleList()
        self.convs.append(GCNConv(in_channels, hidden_channels))
        for _ in range(num_layers - 2):
            self.convs.append(GCNConv(hidden_channels, hidden_channels))
        self.convs.append(GCNConv(hidden_channels, out_channels))
        self.dropout = dropout

    def forward(self, x, edge_index):
        for i, conv in enumerate(self.convs[:-1]):
            x = conv(x, edge_index)
            x = F.relu(x)
            x = F.dropout(x, p=self.dropout, training=self.training)
        x = self.convs[-1](x, edge_index)
        return x

# Classification de noeuds sur Cora
dataset = Planetoid(root='/tmp/Cora', name='Cora')
data = dataset[0]

model = GCN(dataset.num_features, 64, dataset.num_classes)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01,
↪  weight_decay=5e-4)
```

```python
# Entrainement
model.train()
for epoch in range(200):
    optimizer.zero_grad()
    out = model(data.x, data.edge_index)
    loss = F.cross_entropy(out[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()

# Evaluation
model.eval()
pred = model(data.x, data.edge_index).argmax(dim=1)
correct = (pred[data.test_mask] == data.y[data.test_mask]).sum()
acc = correct / data.test_mask.sum()
print(f"Precision test: {acc:.4f}")
```

3.4 Le problème du sur-lissage

Théorème 3.8 (Sur-lissage — Li et al., 2018)**.** *En empilant L couches GCN, les représentations de tous les nœuds convergent vers un point fixe :*

$$\lim_{L\to\infty} \mathbf{H}^{(L)} = \mathbf{1}\boldsymbol{\pi}^\top \mathbf{H}^{(0)}\mathbf{W}_\infty$$

où $\boldsymbol{\pi}$ est le vecteur propre dominant du graphe (lié à la distribution stationnaire du random walk). Tous les nœuds obtiennent la ***même représentation****.*

Conséquences pratiques

- Les GNN profonds ($L > 5$) perdent souvent en performance.
- Les représentations deviennent indistinguables entre nœuds.
- Solutions : connexions résiduelles, normalisation, DropEdge.

3.5 Readout et prédiction au niveau du graphe

Définition 3.9 (Readout)**.** Pour obtenir une représentation au niveau du graphe à partir des représentations de nœuds, on utilise une fonction de **readout** invariante par permutation :

$$\mathbf{h}_G = \text{READOUT}(\{\mathbf{h}_v^{(L)} : v \in V\})$$

Options courantes :

- Somme : $\mathbf{h}_G = \sum_{v\in V} \mathbf{h}_v^{(L)}$
- Moyenne : $\mathbf{h}_G = \frac{1}{|V|}\sum_{v\in V} \mathbf{h}_v^{(L)}$
- Max : $\mathbf{h}_G = \max_{v\in V} \mathbf{h}_v^{(L)}$
- Attention : somme pondérée par des poids appris.

3.6 Pouvoir expressif des GNN

Théorème 3.10 (Lien avec le test de Weisfeiler–Leman — Xu et al., 2019)**.** *Les MPNN sont* ***au plus aussi expressifs*** *que le test de Weisfeiler–Leman d'ordre 1 (1-WL). Plus précisément :*

1. *Si le 1-WL distingue deux graphes G_1 et G_2, alors il existe un MPNN qui les distingue.*
2. *Si le 1-WL ne distingue pas G_1 et G_2, alors* ***aucun*** *MPNN ne les distingue.*

Définition 3.11 (Test de Weisfeiler–Leman (1-WL))**.** Le test 1-WL est un algorithme itératif de raffinement de couleurs :

1. **Initialisation** : $c_v^{(0)} =$ couleur initiale de v.
2. **Itération** : $c_v^{(t+1)} = \text{HASH}\Big(c_v^{(t)}, \{\!\{c_u^{(t)} : u \in \mathcal{N}(v)\}\!\}\Big)$
3. **Arrêt** : quand les couleurs se stabilisent.

où $\{\!\{\cdot\}\!\}$ dénote un multi-ensemble.

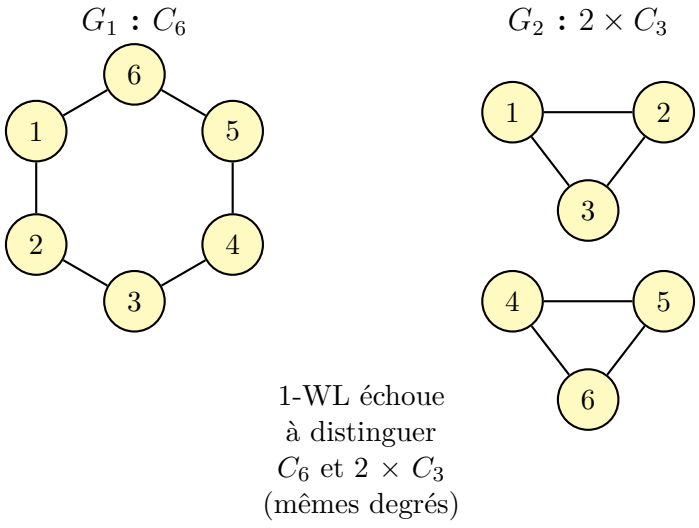

FIG. 3.2 : Deux graphes réguliers que le 1-WL (et donc les MPNN) ne distinguent pas.

3.7 GCN à la main

GCN implémentée from scratch

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

class GCNLayerManual(nn.Module):
    """Couche GCN implementee manuellement."""
    def __init__(self, in_features, out_features):
```

```
        super().__init__()
        self.W = nn.Parameter(torch.randn(in_features, out_features) *
        ↪  0.01)
        self.bias = nn.Parameter(torch.zeros(out_features))

    def forward(self, X, A):
        # A: matrice d'adjacence (n, n)
        # X: attributs (n, d)
        n = A.size(0)
        # Ajout self-loops
        A_hat = A + torch.eye(n, device=A.device)
        # Matrice des degres
        D_hat = torch.diag(A_hat.sum(dim=1))
        # Normalisation symetrique
        D_inv_sqrt = torch.diag(1.0 / torch.sqrt(A_hat.sum(dim=1)))
        A_norm = D_inv_sqrt @ A_hat @ D_inv_sqrt
        # Propagation
        H = A_norm @ X @ self.W + self.bias
        return torch.relu(H)

# Test sur un petit graphe
A = torch.tensor([[0, 1, 0, 1],
                  [1, 0, 1, 0],
                  [0, 1, 0, 1],
                  [1, 0, 1, 0]], dtype=torch.float)
X = torch.randn(4, 8)

layer = GCNLayerManual(8, 16)
out = layer(X, A)
print(f"Sortie: {out.shape}")  # (4, 16)
```

3.8 Types de tâches sur graphes

Définition 3.12 (Niveaux de prédiction)**.** Les tâches d'apprentissage sur graphes se répartissent en trois niveaux :

1. **Niveau nœud** : classification, régression de nœuds (ex. : classification de documents dans un réseau de citations).
2. **Niveau arête** : prédiction de liens, classification d'arêtes (ex. : recommandation).
3. **Niveau graphe** : classification, régression de graphes entiers (ex. : prédiction de propriétés moléculaires).

Classification de graphes moléculaires

```
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv, global_add_pool
from torch_geometric.datasets import TUDataset
```

```python
from torch_geometric.loader import DataLoader

class GraphClassifier(torch.nn.Module):
    def __init__(self, num_features, hidden_dim, num_classes):
        super().__init__()
        self.conv1 = GCNConv(num_features, hidden_dim)
        self.conv2 = GCNConv(hidden_dim, hidden_dim)
        self.conv3 = GCNConv(hidden_dim, hidden_dim)
        self.classifier = torch.nn.Sequential(
            torch.nn.Linear(hidden_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Dropout(0.5),
            torch.nn.Linear(hidden_dim, num_classes),
        )

    def forward(self, data):
        x, edge_index, batch = data.x, data.edge_index, data.batch
        x = F.relu(self.conv1(x, edge_index))
        x = F.relu(self.conv2(x, edge_index))
        x = F.relu(self.conv3(x, edge_index))
        # Readout invariant par permutation
        x = global_add_pool(x, batch)
        return self.classifier(x)

dataset = TUDataset(root='/tmp/PROTEINS', name='PROTEINS')
train_loader = DataLoader(dataset[:900], batch_size=64, shuffle=True)
test_loader = DataLoader(dataset[900:], batch_size=64)

model = GraphClassifier(dataset.num_features, 64, dataset.num_classes)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

for epoch in range(100):
    model.train()
    for data in train_loader:
        optimizer.zero_grad()
        out = model(data)
        loss = F.cross_entropy(out, data.y)
        loss.backward()
        optimizer.step()
```

Exercices

Exercice 3.1 (Dérivation de la couche GCN).

1. Partir de la convolution spectrale $g_\theta * x = U g_\theta(\Lambda) U^\top x$ et montrer que l'approximation du premier ordre en polynômes de Chebyshev donne la couche GCN.

2. Expliquer le rôle de la renormalisation $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$.

Exercice 3.2 (Champ récepteur). Montrer qu'après L couches MPNN, la représentation du nœud v dépend de tous les nœuds à distance $\leq L$ dans le graphe. En déduire le **champ**

récepteur d'un GNN.

Exercice 3.3 (Limites d'expressivité)**.** Construire deux graphes non isomorphes de 8 nœuds que le test 1-WL ne distingue pas. Vérifier expérimentalement qu'un GCN produit les mêmes représentations pour ces deux graphes.

Exercice 3.4 (Passage de messages avec attributs d'arêtes)**.** Étendre l'implémentation GCN "from scratch" pour supporter des attributs d'arêtes $\mathbf{e}_{uv} \in \mathbb{R}^{d_e}$ dans la fonction de message.

Chapitre 4

Variantes de GNN (GAT, GIN, GraphSAGE)

Le GCN de Kipf et Welling (2017) a ouvert la voie, mais ses limitations sont vite apparues : agrégation isotrope, expressivité plafonnée au test de Weisfeiler-Leman d'ordre 1, incapacité à traiter des graphes de grande taille de manière inductive. En réponse, une vague d'architectures alternatives a déferlé : les *Graph Attention Networks* (GAT) introduisent des mécanismes d'attention pour pondérer les voisins ; les *Graph Isomorphism Networks* (GIN) maximisent l'expressivité théorique ; *GraphSAGE* permet l'apprentissage inductif par échantillonnage de voisinages. Ce chapitre explore ces variantes et les compromis qu'elles incarnent.

4.1 Au-delà du GCN : motivations

Le GCN de Kipf et Welling présente plusieurs limitations :

- Agrégation **isotrope** : tous les voisins sont traités de la même manière (pondérés uniquement par le degré).
- **Expressivité limitée** : équivalence stricte au 1-WL.
- **Scalabilité** : nécessite la matrice d'adjacence complète.

Ce chapitre présente trois variantes majeures qui répondent à ces limitations.

4.2 Graph Attention Networks (GAT)

4.2.1 Principe de l'attention sur graphes

Définition 4.1 (Couche GAT — Veličković et al., 2018)**.** La couche GAT utilise un mécanisme d'**attention** pour pondérer les contributions des voisins :

$$\mathbf{h}_v^{(\ell+1)} = \sigma\left(\sum_{u \in \mathcal{N}(v)\cup\{v\}} \alpha_{vu}^{(\ell)}\, \mathbf{W}^{(\ell)} \mathbf{h}_u^{(\ell)}\right)$$

où les coefficients d'attention α_{vu} sont calculés par :

$$e_{vu}^{(\ell)} = \text{LeakyReLU}\left(\mathbf{a}^\top \left[\mathbf{W}\mathbf{h}_v^{(\ell)} \| \mathbf{W}\mathbf{h}_u^{(\ell)}\right]\right) \tag{4.1}$$

$$\alpha_{vu}^{(\ell)} = \frac{\exp(e_{vu}^{(\ell)})}{\sum_{w \in \mathcal{N}(v)\cup\{v\}} \exp(e_{vw}^{(\ell)})} \tag{4.2}$$

où $\mathbf{a} \in \mathbb{R}^{2d'}$ est le vecteur d'attention appris et $\|$ dénote la concaténation.

Remarque 4.2 (Attention multi-tête). Pour stabiliser l'apprentissage, GAT utilise K têtes d'attention indépendantes :

$$\mathbf{h}_v^{(\ell+1)} = \Big\|_{k=1}^{K} \sigma\left(\sum_{u \in \mathcal{N}(v)} \alpha_{vu}^{k} \mathbf{W}^k \mathbf{h}_u^{(\ell)}\right)$$

La dernière couche utilise typiquement la moyenne au lieu de la concaténation.

Pourquoi l'attention ?

L'attention permet au modèle d'apprendre **quels voisins sont importants** pour chaque nœud, contrairement au GCN où la pondération est fixée par la structure du graphe. Cela est particulièrement utile quand :

- Les voisins ont des **pertinences variables**.
- Le graphe contient du **bruit** (arêtes non informatives).
- Les relations sont **asymétriques** (graphes dirigés).

Implémentation GAT avec PyTorch Geometric

```python
import torch
import torch.nn.functional as F
from torch_geometric.nn import GATConv
from torch_geometric.datasets import Planetoid

class GAT(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels,
                 heads=8, dropout=0.6):
        super().__init__()
        self.conv1 = GATConv(in_channels, hidden_channels,
                             heads=heads, dropout=dropout)
        self.conv2 = GATConv(hidden_channels * heads, out_channels,
                             heads=1, concat=False, dropout=dropout)
        self.dropout = dropout

    def forward(self, x, edge_index):
        x = F.dropout(x, p=self.dropout, training=self.training)
        x = F.elu(self.conv1(x, edge_index))
        x = F.dropout(x, p=self.dropout, training=self.training)
        x = self.conv2(x, edge_index)
```

```
        return x

dataset = Planetoid(root='/tmp/Cora', name='Cora')
data = dataset[0]
model = GAT(dataset.num_features, 8, dataset.num_classes, heads=8)
optimizer = torch.optim.Adam(model.parameters(), lr=0.005,
↪  weight_decay=5e-4)

for epoch in range(200):
    model.train()
    optimizer.zero_grad()
    out = model(data.x, data.edge_index)
    loss = F.cross_entropy(out[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()

model.eval()
pred = model(data.x, data.edge_index).argmax(dim=1)
acc = (pred[data.test_mask] == data.y[data.test_mask]).float().mean()
print(f"GAT - Precision test: {acc:.4f}")
```

4.2.2 GATv2 : attention dynamique

Définition 4.3 (GATv2 — Brody et al., 2022)**.** GATv2 corrige une limitation de GAT où l'attention est **statique** (le classement des voisins ne dépend pas du nœud requête). GATv2 utilise :

$$e_{vu} = \mathbf{a}^\top \text{LeakyReLU}(\mathbf{W}[\mathbf{h}_v \| \mathbf{h}_u])$$

La non-linéarité est appliquée **avant** le produit scalaire avec $\mathbf{a}$, rendant l'attention véritablement **dynamique**.

4.3 Graph Isomorphism Network (GIN)

4.3.1 Maximiser le pouvoir expressif

Théorème 4.4 (GIN — Xu et al., 2019)**.** *Pour qu'un MPNN atteigne la puissance maximale du test 1-WL, les conditions suivantes sont nécessaires et suffisantes :*

1. *L'agrégation doit être* ***injective*** *sur les multi-ensembles.*
2. *La fonction de mise à jour doit être* ***injective****.*

Définition 4.5 (Couche GIN)**.** La couche GIN réalise :

$$\mathbf{h}_v^{(\ell+1)} = \text{MLP}^{(\ell)}\left((1+\epsilon^{(\ell)}) \cdot \mathbf{h}_v^{(\ell)} + \sum_{u \in \mathcal{N}(v)} \mathbf{h}_u^{(\ell)}\right)$$

où ϵ est un paramètre appris ou fixé.

Proposition 4.6 (Injectivité de la somme)**.** Parmi les agrégations classiques (somme, moyenne, max), seule la **somme** est injective sur les multi-ensembles :

- $\max\{1, 1, 2\} = \max\{1, 2\} = 2$ (perte d'information).
- $\text{mean}\{1, 1\} = \text{mean}\{1\} = 1$ (perte de cardinalité).
- $\text{sum}\{1, 1, 2\} = 4 \neq \text{sum}\{1, 2\} = 3$ (injectif).

Implémentation GIN

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn import GINConv, global_add_pool
from torch_geometric.datasets import TUDataset
from torch_geometric.loader import DataLoader

class GIN(nn.Module):
    def __init__(self, num_features, hidden_dim, num_classes,
    ↪  num_layers=5):
        super().__init__()
        self.convs = nn.ModuleList()
        self.bns = nn.ModuleList()
        for i in range(num_layers):
            in_dim = num_features if i == 0 else hidden_dim
            mlp = nn.Sequential(
                nn.Linear(in_dim, hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, hidden_dim),
            )
            self.convs.append(GINConv(mlp, train_eps=True))
            self.bns.append(nn.BatchNorm1d(hidden_dim))
        self.classifier = nn.Linear(hidden_dim, num_classes)

    def forward(self, data):
        x, edge_index, batch = data.x, data.edge_index, data.batch
        for conv, bn in zip(self.convs, self.bns):
            x = F.relu(bn(conv(x, edge_index)))
        x = global_add_pool(x, batch)
        return self.classifier(x)

dataset = TUDataset(root='/tmp/MUTAG', name='MUTAG')
loader = DataLoader(dataset, batch_size=32, shuffle=True)
model = GIN(dataset.num_features, 64, dataset.num_classes)
print(f"Parametres: {sum(p.numel() for p in model.parameters()):,}")
```

4.4 GraphSAGE : apprentissage inductif

4.4.1 Motivation : passage à l'échelle

Définition 4.7 (GraphSAGE — Hamilton et al., 2017)**.** GraphSAGE (*SAmple and aggreGatE*) utilise un **échantillonnage de voisinage** pour permettre le passage à l'échelle :

$$\mathbf{h}_v^{(\ell+1)} = \sigma\big(\mathbf{W}^{(\ell)} \cdot \mathrm{CONCAT}\big(\mathbf{h}_v^{(\ell)},\ \mathrm{AGG}(\{\mathbf{h}_u^{(\ell)} : u \in \mathcal{S}(v)\})\big)\big)$$

où $\mathcal{S}(v) \subseteq \mathcal{N}(v)$ est un sous-ensemble échantillonné de voisins (typiquement $|\mathcal{S}(v)| \leq k$).

Proposition 4.8 (Avantages de GraphSAGE)**.**

1. **Inductif** : peut généraliser à des nœuds non vus.
2. **Scalable** : complexité contrôlée par le nombre d'échantillons.
3. **Mini-batch** : compatible avec l'entraînement par mini-lots.

Échantillonnage de sous-graphes pour GraphSAGE

1. Pour chaque nœud cible v, échantillonner k_1 voisins directs.
2. Pour chacun de ces voisins, échantillonner k_2 voisins (2-hop).
3. Former le sous-graphe induit et propager les messages.
4. La complexité par batch est $\mathcal{O}(B \cdot k_1 \cdot k_2)$ au lieu de $\mathcal{O}(n)$.

GraphSAGE avec échantillonnage

```python
import torch
import torch.nn.functional as F
from torch_geometric.nn import SAGEConv
from torch_geometric.loader import NeighborLoader
from torch_geometric.datasets import Planetoid

dataset = Planetoid(root='/tmp/Cora', name='Cora')
data = dataset[0]

class GraphSAGE(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels):
        super().__init__()
        self.conv1 = SAGEConv(in_channels, hidden_channels)
        self.conv2 = SAGEConv(hidden_channels, out_channels)

    def forward(self, x, edge_index):
        x = F.relu(self.conv1(x, edge_index))
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.conv2(x, edge_index)
        return x

# Echantillonnage de voisinage pour mini-batch
train_loader = NeighborLoader(
```

```
    data,
    num_neighbors=[25, 10],  # k1=25, k2=10
    batch_size=256,
    input_nodes=data.train_mask,
)

model = GraphSAGE(dataset.num_features, 64, dataset.num_classes)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

for epoch in range(20):
    model.train()
    for batch in train_loader:
        optimizer.zero_grad()
        out = model(batch.x, batch.edge_index)[:batch.batch_size]
        loss = F.cross_entropy(out, batch.y[:batch.batch_size])
        loss.backward()
        optimizer.step()
```

4.5 Comparaison des variantes

Propriété	GCN	GAT	GIN	SAGE
Attention	Non	Oui	Non	Non
Expressivité 1-WL	$<$	$<$	$=$	$<$
Inductif	Non	Oui	Oui	Oui
Scalable (mini-batch)	Difficile	Modéré	Modéré	Oui
Attributs d'arêtes	Non	Oui (v2)	Non	Non

4.6 Architectures avancées

4.6.1 PNA — Principal Neighbourhood Aggregation

Définition 4.9 (PNA — Corso et al., 2020)**.** PNA combine **plusieurs agrégateurs** et **scalers** :

$$\mathbf{h}_v^{(\ell+1)} = U\left(\mathbf{h}_v^{(\ell)}, \Big\|_{\substack{\oplus \in \{\text{sum, mean,} \\ \text{max, std}\}}} \Big\|_{s \in \{1, \sqrt{d}, 1/\sqrt{d}\}} s \cdot \bigoplus_{u \in \mathcal{N}(v)} M(\mathbf{h}_v^{(\ell)}, \mathbf{h}_u^{(\ell)})\right)$$

La concaténation de $4 \times 3 = 12$ canaux capture différents aspects de la distribution du voisinage.

4.6.2 Connexions résiduelles pour GNN profonds

Définition 4.10 (GCNII — Chen et al., 2020)**.** GCNII ajoute une **connexion résiduelle initiale** et un **rétrécissement d'identité** pour éviter le sur-lissage :

$$\mathbf{H}^{(\ell+1)} = \sigma\Big(\Big((1-\alpha_\ell)\hat{\mathbf{P}}\mathbf{H}^{(\ell)} + \alpha_\ell \mathbf{H}^{(0)}\Big)\big((1-\beta_\ell)\mathbf{I} + \beta_\ell \mathbf{W}^{(\ell)}\big)\Big)$$

où $\hat{\mathbf{P}} = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}$ et α_ℓ, β_ℓ sont des hyperparamètres.

Exercices

Exercice 4.1 (Comparaison expérimentale)**.** Comparer GCN, GAT, GIN et GraphSAGE sur le dataset Cora :

1. Implémenter les quatre modèles avec le même nombre de paramètres.
2. Entraîner 200 époques avec les mêmes hyperparamètres.
3. Reporter la précision test et la courbe de perte.
4. Analyser les différences de performance.

Exercice 4.2 (Visualisation de l'attention GAT)**.**

1. Entraîner un GAT sur Cora et extraire les coefficients d'attention α_{vu}.
2. Visualiser les poids d'attention sur un sous-graphe de 20 nœuds.
3. Les coefficients d'attention sont-ils corrélés avec les labels des nœuds ?

Exercice 4.3 (Pouvoir expressif du GIN)**.**

1. Construire deux graphes que le GCN ne distingue pas mais que le GIN distingue.
2. Vérifier expérimentalement.
3. Construire deux graphes que même le GIN ne distingue pas (échec du 1-WL).

Exercice 4.4 (Mini-batch training)**.** Implémenter l'entraînement par mini-batch avec échantillonnage de voisinage pour un GCN (sans utiliser `NeighborLoader`).

1. Implémenter la stratégie d'échantillonnage.
2. Comparer la vitesse d'entraînement avec le mode full-batch.
3. Évaluer l'impact du nombre de voisins échantillonnés sur la précision.

Chapitre 5

Apprentissage Spectral sur Graphes

La transformée de Fourier est l'un des outils les plus puissants de l'analyse — mais elle suppose que les données vivent sur une grille régulière. Comment définir une transformée de Fourier sur un graphe ? La réponse passe par le *laplacien de graphe*, un opérateur dont les vecteurs propres jouent le rôle des sinusoïdes. Les basses fréquences correspondent à des signaux lisses (voisins similaires), les hautes fréquences à des signaux oscillants. Cette analogie spectrale, développée par Fan Chung dans les années 1990 et appliquée à l'apprentissage profond par Bruna et al. (2014), fournit les fondements mathématiques des réseaux convolutifs spectraux sur graphes.

5.1 Théorie spectrale des graphes

5.1.1 Le laplacien de graphe

Définition 5.1 (Laplacien de graphe)**.** Soit $G = (V, E)$ un graphe non dirigé à n nœuds. Le **laplacien combinatoire** est :

$$\mathbf{L} = \mathbf{D} - \mathbf{A}$$

Le **laplacien normalisé** est :

$$\tilde{\mathbf{L}} = \mathbf{D}^{-1/2}\mathbf{L}\mathbf{D}^{-1/2} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$$

Théorème 5.2 (Propriétés du laplacien)**.** *Le laplacien* $\mathbf{L}$ *satisfait :*

1. $\mathbf{L}$ *est* ***semi-défini positif*** *:* $\mathbf{x}^\top \mathbf{L}\mathbf{x} \geq 0$ *pour tout* $\mathbf{x} \in \mathbb{R}^n$.
2. ***Forme quadratique*** *:* $\mathbf{x}^\top \mathbf{L}\mathbf{x} = \sum_{(i,j)\in E}(x_i - x_j)^2$.
3. $\mathbf{L}$ *a pour* ***valeurs propres*** $0 = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$.
4. *La* ***multiplicité*** *de* $\lambda = 0$ *égale le nombre de composantes connexes.*

Démonstration. Pour la forme quadratique :

$$\mathbf{x}^\top \mathbf{L}\mathbf{x} = \mathbf{x}^\top(\mathbf{D} - \mathbf{A})\mathbf{x} = \sum_i d_i x_i^2 - \sum_{(i,j)\in E} 2x_i x_j = \sum_{(i,j)\in E}(x_i^2 + x_j^2 - 2x_i x_j) = \sum_{(i,j)\in E}(x_i - x_j)^2.$$

□

Le laplacien comme opérateur de lissage

La forme quadratique $\mathbf{x}^\top \mathbf{L}\mathbf{x}$ mesure la **variation totale** du signal $\mathbf{x}$ sur le graphe. Un signal $\mathbf{x}$ est lisse si les nœuds voisins ont des valeurs proches, ce qui correspond à une petite valeur de $\mathbf{x}^\top \mathbf{L}\mathbf{x}$.

5.1.2 Décomposition spectrale

Définition 5.3 (Spectre du graphe)**.** Le **spectre** du graphe est l'ensemble des valeurs propres du laplacien :

$$\mathbf{L} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top$$

où $\mathbf{U} = [\mathbf{u}_1, \ldots, \mathbf{u}_n]$ est la matrice des vecteurs propres (la **base de Fourier du graphe**) et $\mathbf{\Lambda} = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$.

Transformée de Fourier sur graphe

La **transformée de Fourier** d'un signal $\mathbf{x} \in \mathbb{R}^n$ sur le graphe est :

$$\hat{\mathbf{x}} = \mathbf{U}^\top \mathbf{x}$$

La **transformée inverse** est :

$$\mathbf{x} = \mathbf{U}\hat{\mathbf{x}}$$

Le coefficient $\hat{x}_k = \mathbf{u}_k^\top \mathbf{x}$ mesure la projection du signal sur le k-ième mode de Fourier du graphe.

Décomposition spectrale et transformée de Fourier

```python
import torch
import numpy as np
import networkx as nx

# Creer un graphe
G = nx.karate_club_graph()
n = G.number_of_nodes()

# Laplacien normalise
L = nx.normalized_laplacian_matrix(G).toarray()
L = torch.tensor(L, dtype=torch.float)

# Decomposition spectrale
eigenvalues, eigenvectors = torch.linalg.eigh(L)
print(f"Plus petites valeurs propres: {eigenvalues[:5]}")
print(f"Plus grande valeur propre: {eigenvalues[-1]:.4f}")

# Transformee de Fourier d'un signal
x = torch.randn(n)  # signal aleatoire
x_hat = eigenvectors.T @ x  # domaine spectral
x_reconstructed = eigenvectors @ x_hat  # reconstruction
print(f"Erreur reconstruction: {(x - x_reconstructed).norm():.2e}")
```

5.2 Convolution spectrale sur graphes

5.2.1 Définition

Définition 5.4 (Convolution spectrale)**.** Par analogie avec le théorème de convolution classique (la convolution dans le domaine spatial est un produit dans le domaine fréquentiel), la **convolution spectrale** sur graphe est définie par :

$$\mathbf{x} *_G \mathbf{g} = \mathbf{U}\big(\mathbf{U}^\top \mathbf{x} \odot \mathbf{U}^\top \mathbf{g}\big) = \mathbf{U}\,\hat{\mathbf{g}} \odot \hat{\mathbf{x}}$$

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

Définition 5.5 (Filtre spectral paramétré)**.** Un **filtre spectral** est défini par une fonction $g_\theta : \mathbb{R} \to \mathbb{R}$ appliquée aux valeurs propres :

$$g_\theta(\mathbf{L})\,\mathbf{x} = \mathbf{U}\,g_\theta(\mathbf{\Lambda})\,\mathbf{U}^\top \mathbf{x} = \mathbf{U}\,\mathrm{diag}(g_\theta(\lambda_1), \ldots, g_\theta(\lambda_n))\,\mathbf{U}^\top \mathbf{x}$$

où θ sont les paramètres appris du filtre.

Coût computationnel

La convolution spectrale naïve a trois problèmes :

1. La diagonalisation $\mathbf{L} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top$ coûte $\mathcal{O}(n^3)$.
2. Le nombre de paramètres est n (un par valeur propre).
3. Le filtre n'est **pas localisé** dans l'espace spatial.

5.2.2 ChebNet : polynômes de Chebyshev

Définition 5.6 (Filtre de Chebyshev — Defferrard et al., 2016)**.** Pour résoudre les problèmes ci-dessus, on approche g_θ par un polynôme de Chebyshev d'ordre K :

$$g_\theta(\mathbf{L}) = \sum_{k=0}^{K} \theta_k\, T_k(\tilde{\mathbf{L}})$$

où $\tilde{\mathbf{L}} = \frac{2}{\lambda_{\max}}\mathbf{L} - \mathbf{I}$ (rescaling dans $[-1, 1]$) et T_k est le k-ième polynôme de Chebyshev :

$$T_0(x) = 1, \quad T_1(x) = x \tag{5.1}$$
$$T_{k+1}(x) = 2x\,T_k(x) - T_{k-1}(x) \tag{5.2}$$

Théorème 5.7 (Propriétés du filtre de Chebyshev)**.**

1. ***Localisation*** *: le filtre d'ordre K est localisé dans un voisinage de rayon K (support spatial de taille K).*
2. ***Complexité*** *: $\mathcal{O}(K \cdot |E|)$ au lieu de $\mathcal{O}(n^3)$ — linéaire en le nombre d'arêtes.*
3. ***Paramètres*** *: $K + 1$ au lieu de n.*
4. ***Pas de décomposition spectrale*** *nécessaire.*

Implémentation ChebNet

```python
import torch
import torch.nn as nn
from torch_geometric.nn import ChebConv
from torch_geometric.datasets import Planetoid

class ChebNet(nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels, K=3):
        super().__init__()
        self.conv1 = ChebConv(in_channels, hidden_channels, K=K)
        self.conv2 = ChebConv(hidden_channels, out_channels, K=K)

    def forward(self, x, edge_index):
        x = torch.relu(self.conv1(x, edge_index))
        x = torch.dropout(x, p=0.5, train=self.training)
        x = self.conv2(x, edge_index)
        return x

dataset = Planetoid(root='/tmp/Cora', name='Cora')
data = dataset[0]
model = ChebNet(dataset.num_features, 32, dataset.num_classes, K=3)
print(f"Parametres ChebNet: {sum(p.numel() for p in
↪  model.parameters()):,}")
```

5.2.3 Du ChebNet au GCN

Proposition 5.8 (GCN comme cas particulier)**.** La couche GCN de Kipf et Welling s'obtient en prenant $K = 1$ dans le filtre de Chebyshev et en posant $\lambda_{\max} \approx 2$:

$$g_\theta * \mathbf{x} \approx \theta_0 \mathbf{x} + \theta_1(\mathbf{L} - \mathbf{I})\mathbf{x} = \theta_0 \mathbf{x} - \theta_1 \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}\mathbf{x}$$

En posant $\theta_0 = -\theta_1 = \theta$ et en ajoutant la renormalisation :

$$g_\theta * \mathbf{x} = \theta\, \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{x}$$

5.3 Analyse spectrale avancée

5.3.1 Coupure spectrale et Fiedler

Définition 5.9 (Valeur de Fiedler)**.** La **valeur de Fiedler** λ_2 (deuxième plus petite valeur propre du laplacien) mesure la **connectivité algébrique** du graphe. Le vecteur propre associé $\mathbf{u}_2$ (vecteur de Fiedler) définit une partition en deux du graphe.

Théorème 5.10 (Inégalité de Cheeger)**.** *Pour un graphe G, la constante isopérimétrique $h(G)$ satisfait :*

$$\frac{\lambda_2}{2} \leq h(G) \leq \sqrt{2\lambda_2}$$

où $h(G) = \min_{S \subset V} \frac{|\partial S|}{\min(\mathrm{vol}(S), \mathrm{vol}(\bar{S}))}$ et $|\partial S|$ est le nombre d'arêtes coupant S et $\bar{S}$.

5.3.2 Encodages positionnels spectraux

Définition 5.11 (Encodage positionnel laplacien (LPE))**.** Les **encodages positionnels laplaciens** utilisent les k premiers vecteurs propres du laplacien comme attributs supplémentaires des nœuds :

$$\mathrm{LPE}(v) = [\mathbf{u}_2(v), \mathbf{u}_3(v), \ldots, \mathbf{u}_{k+1}(v)] \in \mathbb{R}^k$$

Ambiguïté de signe

Les vecteurs propres sont définis à un signe près : si $\mathbf{u}$ est vecteur propre, alors $-\mathbf{u}$ l'est aussi. Pour lever cette ambiguïté :

- **SignNet** (Lim et al., 2022) : $\phi(\mathbf{u}) + \phi(-\mathbf{u})$.
- **BasisNet** : traiter l'ensemble des vecteurs propres comme base d'un sous-espace.

Encodages positionnels spectraux

```python
import torch
from torch_geometric.transforms import AddLaplacianEigenvectorPE
from torch_geometric.datasets import Planetoid

dataset = Planetoid(root='/tmp/Cora', name='Cora',
                    transform=AddLaplacianEigenvectorPE(k=8))
data = dataset[0]
print(f"Attributs originaux: {data.x.shape}")
print(f"Encodages positionnels: {data.laplacian_eigenvector_pe.shape}")

# Concatenation pour l'entree du GNN
x_augmented = torch.cat([data.x, data.laplacian_eigenvector_pe], dim=1)
print(f"Attributs augmentes: {x_augmented.shape}")
```

5.4 Graph Transformers

Définition 5.12 (Graph Transformer)**.** Un **Graph Transformer** applique l'attention sur tous les nœuds (pas seulement les voisins), brisant la localité du graphe. Les encodages positionnels spectraux fournissent l'information structurelle :

$$\mathbf{h}_v^{(\ell+1)} = \sum_{u \in V} \frac{\exp(\mathbf{q}_v^\top \mathbf{k}_u / \sqrt{d})}{\sum_{w \in V} \exp(\mathbf{q}_v^\top \mathbf{k}_w / \sqrt{d})} \mathbf{v}_u$$

où $\mathbf{q}_v, \mathbf{k}_u, \mathbf{v}_u$ sont les projections query, key, value.

Remarque 5.13 (Graphormer — Ying et al., 2021). Graphormer encode la structure du graphe via :

1. **Biais de centralité** : $\mathbf{h}_v^{(0)} = \mathbf{x}_v + \mathbf{z}_{d_v^+} + \mathbf{z}_{d_v^-}$.
2. **Biais spatial** : $b_{vu} = g(\mathrm{SPD}(v, u))$ (distance du plus court chemin).
3. **Biais d'arête** : encodage des attributs d'arêtes le long du chemin.

5.5 Wavelets sur graphes

Définition 5.14 (Ondelettes spectrales)**.** Les **ondelettes sur graphes** (Hammond et al., 2011) utilisent une fonction d'échelle g dans le domaine spectral :

$$\psi_s(v, u) = \sum_{k=1}^{n} g(s\lambda_k)\, u_k(v)\, u_k(u)$$

où $s > 0$ est le paramètre d'échelle. Les ondelettes à différentes échelles capturent des structures à différentes résolutions.

Exercices

Exercice 5.1 (Spectre du graphe complet et du cycle)**.**

1. Calculer analytiquement le spectre du laplacien du graphe complet K_n.
2. Calculer le spectre du cycle C_n.
3. Implémenter et vérifier numériquement.
4. Visualiser les premiers vecteurs propres (modes de Fourier).

Exercice 5.2 (Filtrage spectral)**.**

1. Implémenter un filtre passe-bas spectral $g(\lambda) = e^{-\alpha\lambda}$.
2. L'appliquer à un signal bruité sur le graphe du club de karaté.
3. Comparer avec le lissage par propagation GCN.

Exercice 5.3 (Partitionnement spectral)**.** Implémenter le partitionnement spectral d'un graphe en utilisant le vecteur de Fiedler.

1. Générer un graphe à deux communautés claires.
2. Calculer le vecteur de Fiedler.
3. Partitionner selon le signe des composantes.
4. Comparer avec l'algorithme de Louvain.

Exercice 5.4 (ChebNet vs. GCN)**.** Comparer expérimentalement ChebNet (avec $K = 2, 3, 5, 10$) et GCN sur Cora.

1. Tracer la précision en fonction de K.
2. Analyser le compromis localité / expressivité.
3. Mesurer le temps d'entraînement.

Chapitre 6

Apprentissage sur les Variétés

Les graphes sont des structures discrètes, mais de nombreuses données vivent sur des espaces *continus* non euclidiens : la surface de la Terre, l'espace des rotations 3D, la variété des matrices de covariance. Peut-on définir des réseaux de neurones directement sur ces *variétés différentiables* ? L'approche géométrique de l'apprentissage profond, systématisée par Bronstein et al. dans leur « blueprint »de 2021, montre que c'est possible — à condition de remplacer les opérations euclidiennes (convolution, translation) par leurs analogues riemanniens (transport parallèle, application exponentielle). Ce chapitre pose les fondations géométriques nécessaires.

6.1 Géométrie différentielle : rappels

6.1.1 Variétés différentiables

Définition 6.1 (Variété différentiable)**.** Une **variété différentiable** de dimension d est un espace topologique $\mathcal{M}$ qui est localement homéomorphe à $\mathbb{R}^d$. Plus précisément, pour chaque point $p \in \mathcal{M}$, il existe un voisinage ouvert U et un homéomorphisme (carte locale) $\varphi : U \to \mathbb{R}^d$. Les changements de cartes $\varphi_\beta \circ \varphi_\alpha^{-1}$ sont C^∞.

Exemple 6.2 (Variétés courantes)**.**

1. La **sphère** $S^2 = \{(x, y, z) \in \mathbb{R}^3 : x^2+y^2+z^2 = 1\}$ — dimension 2.
2. Le **tore** $T^2 = S^1 \times S^1$ — dimension 2.
3. Le groupe $\mathrm{SO}(3)$ — dimension 3.
4. L'espace des matrices symétriques définies positives $\mathrm{Sym}^+(n)$.

6.1.2 Espace tangent et métrique riemannienne

Définition 6.3 (Espace tangent)**.** L'**espace tangent** $T_p\mathcal{M}$ en un point $p \in \mathcal{M}$ est l'espace vectoriel des vecteurs tangents à $\mathcal{M}$ en p. Il est de dimension $d = \dim(\mathcal{M})$.

Définition 6.4 (Métrique riemannienne)**.** Une **métrique riemannienne** sur $\mathcal{M}$ est la donnée, pour chaque $p \in \mathcal{M}$, d'un produit scalaire $g_p : T_p\mathcal{M} \times T_p\mathcal{M} \to \mathbb{R}$ variant lissement avec p. La paire $(\mathcal{M}, g)$ est une **variété riemannienne**.

Objets de base en géométrie riemannienne

- **Longueur d'une courbe** $\gamma : [0,1] \to \mathcal{M}$: $L(\gamma) = \int_0^1 \sqrt{g_{\gamma(t)}(\dot\gamma(t), \dot\gamma(t))}\, dt$
- **Distance géodésique** : $d(p,q) = \inf_\gamma L(\gamma)$ sur les courbes reliant p à q.
- **Application exponentielle** : $\exp_p : T_p\mathcal{M} \to \mathcal{M}$, envoie v sur le point atteint par la géodésique partant de p dans la direction v.
- **Application logarithme** : $\log_p : \mathcal{M} \to T_p\mathcal{M}$, inverse (locale) de $\exp_p$.

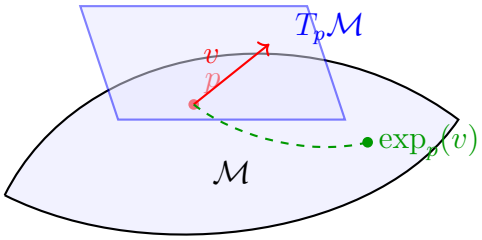

FIG. 6.1 : Espace tangent, vecteur tangent et application exponentielle.

6.2 Opérateurs différentiels sur les variétés

Définition 6.5 (Opérateur de Laplace–Beltrami)**.** L'opérateur de **Laplace–Beltrami** est la généralisation du laplacien à une variété riemannienne :

$$\Delta_{\mathcal{M}} f = \operatorname{div}(\operatorname{grad} f) = \frac{1}{\sqrt{|g|}} \sum_{i,j} \partial_i \Big(\sqrt{|g|}\, g^{ij}\, \partial_j f \Big)$$

où g^{ij} sont les composantes de l'inverse du tenseur métrique et $|g| = \det(g_{ij})$.

Théorème 6.6 (Propriétés de $\Delta_{\mathcal{M}}$)**.**

1. *$\Delta_{\mathcal{M}}$ est **auto-adjoint** et **semi-défini négatif** (avec la convention $\Delta f = -\operatorname{div}(\nabla f)$ on a le signe opposé).*
2. *Il est **intrinsèque** : indépendant du système de coordonnées.*
3. *Il généralise le laplacien de graphe : sur un graphe, $\mathbf{L}$ est une discrétisation de $-\Delta_{\mathcal{M}}$.*
4. *Il admet une décomposition spectrale : $\Delta_{\mathcal{M}} \phi_k = -\lambda_k \phi_k$ avec $0 = \lambda_0 \le \lambda_1 \le \cdots$*

Proposition 6.7 (Équation de la chaleur sur $\mathcal{M}$)**.** L'équation de la chaleur sur une variété s'écrit :

$$\frac{\partial u}{\partial t} = \Delta_{\mathcal{M}} u$$

Sa solution est $u(p,t) = \int_{\mathcal{M}} K_t(p,q)\, u_0(q)\, dq$ où $K_t(p,q) = \sum_k e^{-\lambda_k t} \phi_k(p) \phi_k(q)$ est le **noyau de la chaleur**.

6.3 Convolutions sur les variétés

6.3.1 Défis

Sur une variété, la convolution classique pose plusieurs problèmes :

- Pas de **structure de groupe de translation** (sauf pour $\mathbb{R}^n$ et le tore).
- Pas de **système de coordonnées global**.
- Le **transport parallèle** est nécessaire pour comparer des vecteurs en différents points.

6.3.2 Approche spectrale : MoNet et variantes

Définition 6.8 (MoNet — Monti et al., 2017)**.** MoNet (*Mixture Model Networks*) définit des filtres paramétriques dans l'espace des coordonnées locales pseudo-polaires :

$$\mathbf{h}_v^{(\ell+1)} = \sum_{j=1}^{J} \mathbf{W}_j^{(\ell)} \sum_{u \in \mathcal{N}(v)} w_j(\mathbf{c}(v,u))\, \mathbf{h}_u^{(\ell)}$$

où $\mathbf{c}(v,u) = (\rho_{vu}, \theta_{vu})$ sont les coordonnées pseudo-polaires de u dans le repère local de v, et $w_j(\mathbf{c}) = \exp\!\big(-\frac{1}{2}(\mathbf{c}-\boldsymbol{\mu}_j)^\top \boldsymbol{\Sigma}_j^{-1}(\mathbf{c}-\boldsymbol{\mu}_j)\big)$ sont des noyaux gaussiens appris.

6.3.3 Convolution par descripteurs spectraux

Définition 6.9 (Noyau de diffusion)**.** Le **noyau de diffusion** d'une variété est défini par :

$$K_t(p,q) = \sum_{k=0}^{\infty} e^{-\lambda_k t}\, \phi_k(p)\, \phi_k(q)$$

Il définit un produit scalaire dans un espace de Hilbert à noyau reproduisant (RKHS) et fournit des descripteurs de forme intrinsèques.

Définition 6.10 (HKS — Heat Kernel Signature)**.** Le **Heat Kernel Signature** est un descripteur intrinsèque de la géométrie locale :

$$\mathrm{HKS}(p,t) = K_t(p,p) = \sum_{k=0}^{\infty} e^{-\lambda_k t}\, \phi_k(p)^2$$

Il encode l'information multi-échelle : petits t capturent la géométrie locale, grands t la géométrie globale.

Calcul du HKS sur un maillage

```python
import torch
import numpy as np

def compute_hks(eigenvalues, eigenvectors, time_scales):
    """Calcul du Heat Kernel Signature.
```

```
    Args:
        eigenvalues: (K,) valeurs propres du laplacien
        eigenvectors: (N, K) vecteurs propres
        time_scales: (T,) echelles de temps
    Returns:
        hks: (N, T) descripteurs HKS
    """
    # eigenvalues: (K,), eigenvectors: (N, K), time_scales: (T,)
    # exp(-lambda_k * t): (K, T)
    exp_vals = torch.exp(
        -eigenvalues.unsqueeze(1) * time_scales.unsqueeze(0)
    )
    # phi_k(p)^2: (N, K)
    phi_sq = eigenvectors ** 2
    # HKS(p, t) = sum_k exp(-lambda_k * t) * phi_k(p)^2
    hks = phi_sq @ exp_vals  # (N, T)
    return hks

# Exemple avec 100 valeurs propres et 16 echelles
eigenvalues = torch.linspace(0, 10, 100)
eigenvectors = torch.randn(500, 100)  # 500 sommets
eigenvectors, _ = torch.linalg.qr(eigenvectors)
time_scales = torch.logspace(-2, 2, 16)

hks = compute_hks(eigenvalues, eigenvectors[:, :100], time_scales)
print(f"Descripteurs HKS: {hks.shape}")  # (500, 16)
```

6.4 Convolution par transport parallèle

Définition 6.11 (Transport parallèle)**.** Le **transport parallèle** le long d'une courbe $\gamma : [0,1] \to \mathcal{M}$ est une application linéaire $\Gamma_\gamma : T_{\gamma(0)}\mathcal{M} \to T_{\gamma(1)}\mathcal{M}$ qui transporte un vecteur tangent le long de γ en préservant le produit scalaire et en satisfaisant l'équation :

$$\nabla_{\dot{\gamma}} V = 0$$

où ∇ est la connexion de Levi-Civita.

Définition 6.12 (Gauge Equivariant CNN — de Haan et al., 2021)**.** Un **Gauge Equivariant CNN** définit des convolutions sur une variété en utilisant des **repères locaux** (gauges) et le transport parallèle :

$$[f * \psi](p) = \int_{\mathcal{M}} f(\Gamma_{q \to p} \cdot)\, \psi(\exp_p^{-1}(q))\, dq$$

Le réseau est équivariant par changement de jauge : le choix du repère local n'affecte pas la sortie.

6.5 Apprentissage sur des variétés spécifiques

6.5.1 Convolutions sur la sphère

Définition 6.13 (Spherical CNN — Cohen et al., 2018)**.** Les **Spherical CNNs** définissent des convolutions sur S^2 en utilisant les harmoniques sphériques comme base spectrale :

$$[f * \psi](\hat{\mathbf{r}}) = \sum_{\ell=0}^{L} \sum_{m=-\ell}^{\ell} \hat{f}_\ell^m \, \hat{\psi}_\ell \, Y_\ell^m(\hat{\mathbf{r}})$$

où $\hat{f}_\ell^m$ et $\hat{\psi}_\ell$ sont les coefficients dans la base des harmoniques sphériques. Pour un filtre **zonal** (invariant par rotation autour de l'axe), $\hat{\psi}_\ell$ ne dépend que de ℓ.

Théorème 6.14 (Équivariance par SO(3))**.** *La convolution sphérique est équivariante par rotation : si $R \in \mathrm{SO}(3)$ et $L_R f(\hat{\mathbf{r}}) = f(R^{-1}\hat{\mathbf{r}})$, alors :*

$$L_R(f * \psi) = (L_R f) * \psi$$

6.5.2 Apprentissage sur les groupes de Lie

Définition 6.15 (LieConv — Finzi et al., 2020)**.** LieConv définit des convolutions directement sur les groupes de Lie en utilisant un **noyau continu** paramétré dans l'algèbre de Lie :

$$[f * \psi](g) = \int_G f(h) \, \psi(\log(h^{-1} g)) \, d\mu(h)$$

En pratique, l'intégrale est approchée par Monte Carlo sur un voisinage échantillonné.

6.6 Applications : analyse de formes 3D

Définition 6.16 (Correspondance de formes)**.** Le problème de **correspondance de formes** consiste à trouver une application $T : \mathcal{M}_1 \to \mathcal{M}_2$ entre deux surfaces qui préserve la structure géométrique. Les méthodes spectrales utilisent les **cartes fonctionnelles** :

$$\mathbf{C} = \mathbf{\Phi}_2^\dagger \, T_f \, \mathbf{\Phi}_1$$

où $\mathbf{\Phi}_i$ sont les matrices des fonctions propres du laplacien tronqué et T_f est l'opérateur de correspondance fonctionnelle.

Cartes fonctionnelles pour la correspondance de formes

```python
import torch

def functional_map(phi_1, phi_2, descriptors_1, descriptors_2, k=30):
    """Calcul d'une carte fonctionnelle.

    Args:
        phi_1, phi_2: (N, k) fonctions propres tronquees
        descriptors_1, descriptors_2: (N, d) descripteurs (HKS, etc.)
        k: nombre de fonctions propres
    Returns:
```

```
        C: (k, k) matrice de carte fonctionnelle
    """
    # Projection des descripteurs dans la base spectrale
    A1 = phi_1[:, :k].T @ descriptors_1  # (k, d)
    A2 = phi_2[:, :k].T @ descriptors_2  # (k, d)

    # Resolution au sens des moindres carres : C * A1 = A2
    C, _ = torch.linalg.lstsq(A1.T, A2.T)
    C = C.T  # (k, k)
    return C

# Exemple
N = 1000  # nombre de sommets
k = 30    # taille de la base
phi_1 = torch.randn(N, k)
phi_2 = torch.randn(N, k)
desc_1 = torch.randn(N, 16)
desc_2 = torch.randn(N, 16)
C = functional_map(phi_1, phi_2, desc_1, desc_2, k=k)
print(f"Carte fonctionnelle: {C.shape}")  # (30, 30)
```

6.7 Optimisation sur les variétés

Définition 6.17 (Gradient riemannien)**.** Le **gradient riemannien** d'une fonction $f : \mathcal{M} \to \mathbb{R}$ en p est l'unique vecteur $\operatorname{grad} f(p) \in T_p\mathcal{M}$ tel que :

$$g_p(\operatorname{grad} f(p), v) = df_p(v), \quad \forall v \in T_p\mathcal{M}$$

Descente de gradient riemannienne

1. Calculer le gradient euclidien $\nabla f(\mathbf{x})$.
2. Projeter sur l'espace tangent : $\xi = \operatorname{Proj}_{T_{\mathbf{x}}\mathcal{M}}(\nabla f)$.
3. Effectuer un pas de rétraction : $\mathbf{x}' = \operatorname{Retr}_{\mathbf{x}}(-\eta\,\xi)$.

Pour $\mathrm{SO}(n)$, la rétraction est $\operatorname{Retr}_Q(\xi) = \operatorname{qr}(Q + \xi)$.

Optimisation sur la sphère avec geoopt

```
import torch
import geoopt

# Variable sur la sphere S^{n-1}
sphere = geoopt.Sphere()
x = sphere.random(torch.Size([10]), dtype=torch.float)
x = geoopt.ManifoldParameter(x, manifold=sphere)

# Fonction objectif : minimiser la distance a un point cible
```

```python
target = sphere.random(torch.Size([10]), dtype=torch.float)
optimizer = geoopt.optim.RiemannianAdam([x], lr=0.01)

for step in range(100):
    optimizer.zero_grad()
    loss = (x - target).norm() ** 2
    loss.backward()
    optimizer.step()

print(f"Distance finale: {(x - target).norm():.6f}")
print(f"Sur la sphere: {x.norm():.6f}")  # doit etre ~1
```

Exercices

Exercice 6.1 (Laplace–Beltrami sur la sphère).

1. Montrer que les harmoniques sphériques Y_ℓ^m sont fonctions propres de Δ_{S^2} avec valeurs propres $\lambda_\ell = \ell(\ell+1)$.
2. Calculer les premières harmoniques sphériques et les visualiser.

Exercice 6.2 (Noyau de la chaleur).

1. Calculer analytiquement le noyau de la chaleur sur le cercle S^1.
2. Implémenter le calcul numérique du noyau de la chaleur sur un maillage triangulaire.
3. Visualiser la diffusion de la chaleur à différentes échelles t.

Exercice 6.3 (Transport parallèle).

1. Calculer le transport parallèle le long d'un méridien de la sphère.
2. Montrer que le transport parallèle autour d'un triangle sphérique induit une rotation d'angle égal à l'excès sphérique.
3. Implémenter le transport parallèle discret sur un maillage.

Exercice 6.4 (Correspondance de formes). Implémenter un pipeline de correspondance de formes utilisant :

1. Les descripteurs HKS comme attributs de nœuds.
2. Un GNN pour apprendre des descripteurs équivariants.
3. Les cartes fonctionnelles pour la correspondance.

Chapitre 7

Réseaux Équivariants et Invariants

Les réseaux de neurones convolutifs doivent leur succès à une propriété mathématique précise : l'équivariance par translation. Un chat reste un chat, qu'il soit à gauche ou à droite de l'image. Mais que se passe-t-il si les symétries pertinentes ne sont pas des translations ? En chimie moléculaire, les propriétés d'une molécule sont invariantes par rotation et translation de l'ensemble des atomes. En physique des particules, les interactions obéissent à des symétries de jauge. Taco Cohen et Max Welling, en 2016, ont ouvert la voie des *réseaux équivariants* en montrant comment construire des couches de neurones qui respectent les symétries d'un groupe donné. Cette approche, formalisée par le cadre géométrique de Bronstein et al. unifie les CNN, les GNN et les transformeurs comme cas particuliers d'un principe unique : la structure du réseau doit refléter les symétries du problème.

7.1 Formalisme général

7.1.1 Rappel : équivariance

Définition 7.1 (Équivariance générale)**.** Soit G un groupe agissant sur les espaces $\mathcal{X}$ et $\mathcal{Y}$ via les représentations $\rho_{\mathcal{X}}$ et $\rho_{\mathcal{Y}}$. Une fonction $f : \mathcal{X} \to \mathcal{Y}$ est G**-équivariante** si :

$$f(\rho_{\mathcal{X}}(g) \cdot x) = \rho_{\mathcal{Y}}(g) \cdot f(x), \quad \forall g \in G,\ \forall x \in \mathcal{X}.$$

Proposition 7.2 (Composition d'applications équivariantes)**.** Si $f : \mathcal{X} \to \mathcal{Y}$ et $g : \mathcal{Y} \to \mathcal{Z}$ sont équivariantes, alors $g \circ f : \mathcal{X} \to \mathcal{Z}$ est équivariante. Cela justifie l'empilement de couches équivariantes dans un réseau profond.

7.1.2 Construction de couches équivariantes

Théorème 7.3 (Caractérisation des couches linéaires équivariantes)**.** *L'espace des applications linéaires G-équivariantes $T : V \to W$ est :*

$$\mathrm{Hom}_G(V, W) = \{T \in \mathrm{Lin}(V, W) : T\rho_V(g) = \rho_W(g)T,\ \forall g \in G\}$$

Sa dimension est donnée par le produit scalaire des caractères : $\dim \mathrm{Hom}_G(V, W) = \langle \chi_V, \chi_W \rangle_G$.

Contrainte d'équivariance = partage de poids

Imposer l'équivariance revient à **contraindre la matrice de poids** de la couche linéaire. Plus le groupe de symétrie est grand, plus la contrainte est forte, et moins il y a de paramètres libres.

Groupe	Contrainte	Paramètres
Trivial $\{e\}$	Aucune	n^2
Translation $\mathbb{Z}^d$	Toeplitz (convolution)	k^d
Permutation S_n	Doublement stochastique	2
SO(3)	Clebsch–Gordan	$\sum_k n_k m_k$

7.2 Réseaux équivariants par SE(3)

7.2.1 Motivation : physique et chimie

De nombreux problèmes physiques possèdent une symétrie SE(3) (rotations + translations) :

- Prédiction d'énergie moléculaire.
- Dynamique des protéines.
- Simulation de fluides.

7.2.2 Représentations de type ℓ

Définition 7.4 (Champs tensoriels de type ℓ)**.** Un **champ tensoriel de type** ℓ est une fonction $f : \mathbb{R}^3 \to \mathbb{R}^{2\ell+1}$ qui se transforme selon la représentation irréductible D^ℓ sous rotation :

$$[L_R f](\mathbf{x}) = D^\ell(R)\, f(R^{-1}\mathbf{x})$$

- $\ell = 0$: scalaires (1D) — énergie, charge.
- $\ell = 1$: vecteurs (3D) — forces, vitesses.
- $\ell = 2$: tenseurs d'ordre 2 sans trace (5D) — contraintes.

7.2.3 Tensor Field Networks (TFN)

Définition 7.5 (TFN — Thomas et al., 2018)**.** Les **Tensor Field Networks** définissent des couches équivariantes par SE(3) en utilisant les produits tensoriels de représentations irréductibles. Pour un nœud v avec position $\mathbf{r}_v$:

$$\mathbf{h}_v^{\ell,(\text{out})} = \sum_{\ell_1,\ell_2} \sum_{u \in \mathcal{N}(v)} W^{\ell_1\ell_2\ell}(r_{vu}) \left(\mathbf{h}_u^{\ell_1} \otimes_{\text{CG}} Y^{\ell_2}(\hat{\mathbf{r}}_{vu})\right)^\ell$$

où :

- $\hat{\mathbf{r}}_{vu} = (\mathbf{r}_v - \mathbf{r}_u)/r_{vu}$ est la direction normalisée.

- $Y^{\ell_2}(\hat{\mathbf{r}})$ sont les harmoniques sphériques.
- $\otimes_{\text{CG}}$ est le produit tensoriel avec les coefficients de Clebsch–Gordan.
- $W^{\ell_1\ell_2\ell}(r)$ est un noyau radial (scalaire, donc invariant).

Théorème 7.6 (Équivariance de TFN)**.** *Les couches TFN sont équivariantes par* SE(3) *: pour* $R \in \mathrm{SO}(3)$ *et* $\mathbf{t} \in \mathbb{R}^3$ *:*

$$f(R\mathbf{r}_1 + \mathbf{t}, \ldots, R\mathbf{r}_n + \mathbf{t}) = \bigoplus_{\ell} D^{\ell}(R)\, f(\mathbf{r}_1, \ldots, \mathbf{r}_n)$$

La preuve repose sur la propriété de transformation des harmoniques sphériques et la décomposition de Clebsch–Gordan.

Réseau équivariant avec e3nn

```python
import torch
from e3nn import o3
from e3nn.nn.models.gate_points_2101 import Network

# Configuration du reseau
model = Network(
    irreps_in="5x0e",           # 5 scalaires en entree
    irreps_hidden="32x0e + 8x1o + 4x2e",  # representations cachees
    irreps_out="1x0e",          # 1 scalaire en sortie (energie)
    irreps_node_attr="1x0e",    # attributs de noeud
    irreps_edge_attr=o3.Irreps.spherical_harmonics(lmax=2),
    layers=3,
    max_radius=5.0,
    number_of_basis=10,
    radial_layers=2,
    radial_neurons=64,
    num_neighbors=12,
    num_nodes=20,
)

# Donnees : positions 3D et attributs
positions = torch.randn(20, 3)
node_features = torch.randn(20, 5)
node_attrs = torch.ones(20, 1)

# Verification de l'equivariance
R = o3.rand_matrix()  # rotation aleatoire
out_original = model({"node_input": node_features,
                      "positions": positions,
                      "node_attrs": node_attrs})
out_rotated = model({"node_input": node_features,
                     "positions": positions @ R.T,
                     "node_attrs": node_attrs})
print(f"Invariance scalaire: "
      f"{torch.allclose(out_original, out_rotated, atol=1e-4)}")
```

7.3 EGNN : Équivariance simplifiée

Définition 7.7 (EGNN — Satorras et al., 2021)**.** Les **E(n) Equivariant Graph Neural Networks** (EGNN) sont une architecture simplifiée qui atteint l'équivariance par $\mathrm{E}(n)$ **sans** utiliser les représentations irréductibles :

$$\mathbf{m}_{ij} = \phi_e(\mathbf{h}_i, \mathbf{h}_j, d_{ij}^2, a_{ij}) \tag{7.1}$$

$$\mathbf{h}_i' = \phi_h\left(\mathbf{h}_i, \sum_{j \neq i} \mathbf{m}_{ij}\right) \tag{7.2}$$

$$\mathbf{x}_i' = \mathbf{x}_i + \sum_{j \neq i} (\mathbf{x}_i - \mathbf{x}_j)\, \phi_x(\mathbf{m}_{ij}) \tag{7.3}$$

où $d_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|$ et ϕ_e, ϕ_h, ϕ_x sont des MLP.

Proposition 7.8 (Équivariance de l'EGNN)**.**

1. Les **attributs scalaires** $\mathbf{h}_i$ sont **invariants** par $\mathrm{E}(n)$.
2. Les **coordonnées** $\mathbf{x}_i$ sont **équivariantes** par $\mathrm{E}(n)$.
3. La preuve repose sur le fait que d_{ij}^2 et $\mathbf{x}_i - \mathbf{x}_j$ se transforment correctement.

Implémentation EGNN

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

class EGNNLayer(nn.Module):
    """Couche E(n) equivariante."""
    def __init__(self, node_dim, hidden_dim):
        super().__init__()
        self.phi_e = nn.Sequential(
            nn.Linear(2 * node_dim + 1, hidden_dim),
            nn.SiLU(),
            nn.Linear(hidden_dim, hidden_dim),
        )
        self.phi_h = nn.Sequential(
            nn.Linear(node_dim + hidden_dim, hidden_dim),
            nn.SiLU(),
            nn.Linear(hidden_dim, node_dim),
        )
        self.phi_x = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
            nn.SiLU(),
            nn.Linear(hidden_dim, 1),
        )

    def forward(self, h, x, edge_index):
        row, col = edge_index
        # Distances au carre (invariant)
        diff = x[row] - x[col]
        dist_sq = (diff ** 2).sum(dim=-1, keepdim=True)
```

```
        # Messages
        m = self.phi_e(torch.cat([h[row], h[col], dist_sq], dim=-1))
        # Aggregation pour h
        agg = torch.zeros_like(h)
        agg.index_add_(0, row, m)
        h_new = h + self.phi_h(torch.cat([h, agg], dim=-1))
        # Mise a jour des positions (equivariante)
        x_weights = self.phi_x(m)
        x_agg = torch.zeros_like(x)
        x_agg.index_add_(0, row, diff * x_weights)
        x_new = x + x_agg
        return h_new, x_new

# Test d'equivariance
layer = EGNNLayer(32, 64)
h = torch.randn(10, 32)
x = torch.randn(10, 3)
edge_index = torch.randint(0, 10, (2, 30))

h_out, x_out = layer(h, x, edge_index)

# Rotation aleatoire
R = torch.linalg.qr(torch.randn(3, 3))[0]
if R.det() < 0:
    R[:, 0] *= -1
h_rot, x_rot = layer(h, x @ R.T, edge_index)
print(f"h invariant: {torch.allclose(h_out, h_rot, atol=1e-4)}")
print(f"x equivariant: {torch.allclose(x_out @ R.T, x_rot, atol=1e-4)}")
```

7.4 Réseaux invariants

Définition 7.9 (Invariants fondamentaux)**.** Pour construire un réseau invariant par $\mathrm{E}(n)$, on peut utiliser les **invariants fondamentaux** :

1. **Distances** : $d_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|$.
2. **Angles** : $\theta_{ijk} = \arccos\left(\frac{(\mathbf{x}_j - \mathbf{x}_i)\cdot(\mathbf{x}_k - \mathbf{x}_i)}{\|\mathbf{x}_j - \mathbf{x}_i\|\|\mathbf{x}_k - \mathbf{x}_i\|}\right)$.
3. **Dihedral** : angle entre les plans définis par (i, j, k) et (j, k, l).

Théorème 7.10 (Complétude des invariants)**.** *Les distances interatomiques $\{d_{ij}\}$ déterminent la géométrie à réflexion près. L'ajout des angles dièdres lève l'ambiguïté de réflexion. Tout invariant de* $\mathrm{SE}(3)$ *peut s'exprimer comme fonction des distances et des angles.*

Définition 7.11 (SchNet — Schütt et al., 2018)**.** SchNet utilise les distances interatomiques comme seule information géométrique :

$$\mathbf{h}_i^{(\ell+1)} = \mathbf{h}_i^{(\ell)} + \sum_{j \in \mathcal{N}(i)} \mathbf{W}^{(\ell)} \mathbf{h}_j^{(\ell)} \odot \mathrm{RBF}(d_{ij})$$

où $\text{RBF}(d) = [\exp(-\gamma_1(d-\mu_1)^2), \ldots, \exp(-\gamma_K(d-\mu_K)^2)]$ est une base de fonctions radiales.

7.5 Hiérarchie de modèles équivariants

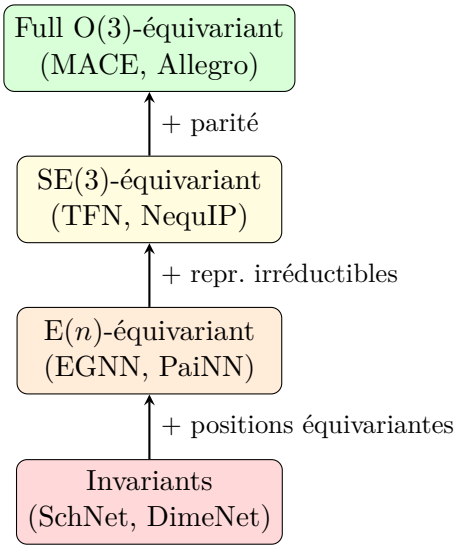

FIG. 7.1 : Hiérarchie des architectures équivariantes.

Exercices

Exercice 7.1 (Vérification d'équivariance)**.** Pour chacun des modèles suivants, vérifier numériquement l'équivariance en appliquant des rotations, translations et réflexions aléatoires :

1. SchNet (invariance E(3)).
2. EGNN (équivariance E(n)).
3. Un TFN avec sorties vectorielles (équivariance SE(3)).

Exercice 7.2 (Produits tensoriels)**.**

1. Calculer le produit tensoriel $1 \otimes 1$ pour SO(3) et vérifier la décomposition $D^1 \otimes D^1 = D^0 \oplus D^1 \oplus D^2$.
2. Implémenter avec `e3nn` et vérifier.
3. Construire une couche équivariante qui prend deux vecteurs et produit un scalaire (produit scalaire), un vecteur (produit vectoriel) et un tenseur d'ordre 2.

Exercice 7.3 (EGNN from scratch)**.** Implémenter un EGNN complet pour la prédiction d'énergie moléculaire sur le dataset QM9.

1. Charger QM9 depuis PyTorch Geometric.
2. Implémenter 4 couches EGNN.

3. Entraîner sur la propriété U_0 (énergie interne).

4. Vérifier l'invariance E(3) de la prédiction.

Exercice 7.4 (Comparaison invariant vs. équivariant)**.** Comparer SchNet (invariant) et EGNN (équivariant) sur la prédiction de forces moléculaires (quantité vectorielle).

1. Pourquoi SchNet ne peut-il pas prédire directement des forces ?

2. Comment obtient-on les forces à partir d'un modèle d'énergie ?

3. L'EGNN prédit-il directement des forces équivariantes ?

Chapitre 8

Représentations de Nuages de Points

En 2017, Charles Qi et ses collaborateurs de Stanford publient PointNet, une architecture de réseau de neurones capable de traiter directement des nuages de points 3D — sans les convertir en voxels ni en maillages. L'idée clé est élégante : puisqu'un nuage de points est un ensemble *non ordonné*, l'architecture doit être *invariante par permutation*. PointNet y parvient en appliquant une fonction partagée à chaque point puis en aggrégeant par un max global. PointNet++, Dynamic Graph CNN (DGCNN), et les méthodes basées sur les transformeurs étendront ensuite cette approche en capturant les structures locales et les relations de voisinage. Ce chapitre explore ces architectures, des fondements théoriques (théorème de Zaheer sur les fonctions d'ensembles) aux applications en conduite autonome, robotique et reconstruction 3D.

Intuition

Un nuage de points 3D est un ensemble **non ordonné** de points $\{x_1, \ldots, x_n\} \subset \mathbb{R}^3$. Contrairement aux images (grille régulière) ou aux graphes (structure combinatoire), il n'a pas de connectivité canonique. Le défi est de concevoir des architectures qui soient **invariantes par permutation** des points et, idéalement, **équivariantes** par rapport aux transformations géométriques (rotations, translations).

8.1 Le problème de l'invariance par permutation

Définition 8.1 (Fonction invariante par permutation)**.** Une fonction $f : \mathbb{R}^{n \times d} \to \mathbb{R}^k$ est **invariante par permutation** si pour toute matrice de permutation $\Pi \in \{0,1\}^{n \times n}$:

$$f(\Pi X) = f(X).$$

Théorème 8.2 (Caractérisation — Zaheer et al. 2017)**.** *Toute fonction continue $f : 2^{\mathbb{R}^d} \to \mathbb{R}$ invariante par permutation peut s'écrire sous la forme :*

$$f(\{x_1, \ldots, x_n\}) = \rho\left(\sum_{i=1}^{n} \phi(x_i)\right)$$

pour des fonctions continues $\phi : \mathbb{R}^d \to \mathbb{R}^q$ et $\rho : \mathbb{R}^q \to \mathbb{R}$ (sous réserve d'un q suffisamment grand).

Remarque 8.3. Ce théorème fonde l'architecture **Deep Sets** et, par extension, **PointNet**. L'opération clé est l'agrégation par somme (ou max) qui garantit l'invariance par permutation.

8.2 PointNet

Définition 8.4 (Architecture PointNet — Qi et al. 2017)**.** **PointNet** traite directement un nuage de n points $X \in \mathbb{R}^{n\times 3}$:

1. **Transformation spatiale** : un mini-réseau (T-Net) prédit une matrice $T \in \mathbb{R}^{3\times 3}$ qui aligne le nuage : $X' = X \cdot T$.

2. **MLP partagé** : chaque point est transformé indépendamment par un MLP ϕ : $\mathbb{R}^3 \to \mathbb{R}^{1024}$: $h_i = \phi(x'_i)$.

3. **Agrégation symétrique** : un max-pooling global produit un vecteur de features global : $g = \max_{i=1}^{n} h_i$.

4. **Tête de classification/segmentation** : un MLP final sur g (ou $[g; h_i]$ pour la segmentation).

PointNet — équation fondamentale

$$f(X) = \rho\Big(\max_{i=1}^{n} \phi(x_i)\Big)$$

où ϕ est un MLP point par point et ρ un MLP global.

Proposition 8.5 (Approximation universelle de PointNet)**.** PointNet peut approcher toute fonction continue invariante par permutation sur les nuages de points compacts, au sens du théorème 8.2.

Limitations de PointNet

PointNet traite chaque point **indépendamment** avant l'agrégation : il ne capture pas les structures locales (voisinages). Deux nuages avec les mêmes points mais des géométries locales différentes peuvent avoir le même vecteur global.

8.3 PointNet++

Définition 8.6 (Architecture PointNet++ — Qi et al. 2017)**.** **PointNet++** introduit une hiérarchie de traitements locaux :

1. **Échantillonnage** : sélection de n' centres par *farthest point sampling* (FPS).

2. **Regroupement** : pour chaque centre c_j, sélection des K plus proches voisins (ou ball query de rayon r).

3. **PointNet local** : application de PointNet sur chaque groupe, produisant un feature $g_j \in \mathbb{R}^{d'}$.

4. **Itération** : répétition sur les centres avec leurs features, créant une pyramide multi-échelle.

Définition 8.7 (Set Abstraction Layer)**.** La couche de **Set Abstraction** (SA) combine les trois opérations :

$$\mathrm{SA}(X) = \{g_j\}_{j=1}^{n'}, \qquad g_j = \max_{x_i \in \mathcal{N}(c_j)} \phi(x_i - c_j)$$

où $\mathcal{N}(c_j)$ est le voisinage du centre c_j.

Intuition

PointNet++ est au nuage de points ce que le CNN est à l'image : il applique des opérations locales de manière hiérarchique. La différence est qu'il n'y a pas de grille : les voisinages sont déterminés dynamiquement.

8.4 Convolutions sur nuages de points

Définition 8.8 (Convolution continue sur nuage de points)**.** Une **convolution continue** sur un nuage de points est :

$$(f * g)(x_i) = \sum_{x_j \in \mathcal{N}(x_i)} g(x_j - x_i) \cdot h_j$$

où $g : \mathbb{R}^3 \to \mathbb{R}^{d' \times d}$ est un filtre continu et $h_j \in \mathbb{R}^d$ sont les features du point x_j.

Exemple 8.9 (Architectures basées sur la convolution)**.**

- **PointConv** (Wu et al. 2019) : g est paramétré par un MLP appliqué à la position relative et à la densité locale.
- **KPConv** (Thomas et al. 2019) : g est défini par des points noyaux fixes $\{y_k\}_{k=1}^{K}$ avec des fonctions de corrélation : $g(x) = \sum_k W_k \cdot \max(0, 1 - \|x - y_k\| / \sigma)$.
- **DGCNN** (Wang et al. 2019) : convolution sur un graphe dynamique de k-plus proches voisins dans l'espace des features.

8.5 Équivariance par rotation

Définition 8.10 (Équivariance SO(3))**.** Une fonction $f : \mathbb{R}^{n \times 3} \to \mathbb{R}^{n \times d}$ est **SO(3)-équivariante** si pour toute rotation $R \in \mathrm{SO}(3)$:

$$f(R \cdot X) = \rho(R) \cdot f(X)$$

où $\rho(R)$ est une représentation de R agissant sur l'espace de sortie.

Théorème 8.11 (Convolutions équivariantes — Thomas et al. 2018)**.** *Les filtres équivariants sous* SO(3) *s'expriment dans la base des harmoniques sphériques :*

$$g^{(\ell)}(r, \hat{x}) = R^{(\ell)}(r) \cdot Y^{(\ell)}(\hat{x})$$

où $Y^{(\ell)}$ sont les harmoniques sphériques de degré ℓ, $R^{(\ell)}$ est une fonction radiale apprise, et $\hat{x} = x / \|x\|$.

Remarque 8.12. Les architectures équivariantes (Tensor Field Networks, SE(3)-Transformers, EGNN) garantissent que les prédictions se transforment correctement sous rotation. Cela élimine le besoin d'augmentation de données par rotations aléatoires.

8.6 Applications

Exemple 8.13 (Classification d'objets 3D)**.** Sur le benchmark **ModelNet40** (12 311 modèles, 40 classes) :

Méthode	Accuracy (%)
PointNet	89.2
PointNet++	91.9
DGCNN	92.9
KPConv	92.9
Point Transformer	93.7

Exemple 8.14 (Segmentation sémantique de scènes)**.** Pour la segmentation de nuages de points LiDAR (S3DIS, ScanNet), PointNet++ et KPConv atteignent des performances compétitives. La segmentation attribue à chaque point une étiquette sémantique (sol, mur, meuble, etc.).

```python
import torch
from torch_geometric.nn import PointNetConv, fps, radius

class PointNetPPLayer(torch.nn.Module):
    """Une couche Set Abstraction de PointNet++."""
    def __init__(self, ratio, r, nn):
        super().__init__()
        self.ratio = ratio
        self.r = r
        self.conv = PointNetConv(nn)

    def forward(self, x, pos, batch):
        # Farthest Point Sampling
        idx = fps(pos, batch, ratio=self.ratio)
        # Ball query
        row, col = radius(
            pos, pos[idx], self.r, batch, batch[idx]
        )
        edge_index = torch.stack([col, row], dim=0)
        # PointNet local
        x_out = self.conv(x, (pos, pos[idx]), edge_index)
        return x_out, pos[idx], batch[idx]
```

8.7 Exercices

Exercice 8.1. Montrer que le max-pooling global est invariant par permutation. Donner un contre-exemple montrant que la concaténation ne l'est pas.

Exercice 8.2. Calculer le nombre de paramètres de PointNet pour un nuage de $n = 1024$ points en 3D, avec un MLP point par point $[3, 64, 128, 1024]$ et un classificateur $[1024, 512, 256, 40]$.

Exercice 8.3 (Équivariance)**.** Montrer que PointNet avec T-Net est **approximativement** invariant par rotation mais pas exactement équivariant. Quelle propriété manque-t-il ?

Exercice 8.4 (Implémentation)**.** Implémenter un PointNet simple en PyTorch et l'entraîner sur ModelNet10. Comparer avec et sans le T-Net.

Exercice 8.5 (Convolution sur nuage)**.** Expliquer pourquoi KPConv utilise des points noyaux fixes plutôt qu'un MLP pour définir le filtre. Quel avantage cela apporte-t-il en termes d'efficacité et d'interprétabilité ?

Chapitre 9

Applications

L'apprentissage géométrique n'est pas un exercice théorique : il résout des problèmes concrets que les méthodes classiques ne pouvaient pas aborder. En chimie, les GNN prédisent les propriétés moléculaires avec une précision qui rivalise avec la mécanique quantique ab initio, mais mille fois plus vite. En physique des particules, ils reconstructent les trajectoires dans les détecteurs du LHC au CERN. En biologie, AlphaFold (Jumper et al., 2021) utilise des transformeurs géométriques pour prédire la structure 3D des protéines, résolvant un problème ouvert depuis cinquante ans. Dans les réseaux sociaux, les GNN détectent les communautés et prédisent les liens. Ce chapitre passe en revue ces applications, montrant comment les principes géométriques se traduisent en performances pratiques.

Intuition

L'apprentissage géométrique trouve des applications dans de nombreux domaines où les données possèdent une structure non euclidienne naturelle : molécules (graphes), réseaux sociaux (graphes), surfaces (variétés), simulations physiques (maillages et particules). Ce chapitre présente les applications les plus marquantes.

9.1 Découverte de médicaments

Définition 9.1 (Représentation moléculaire par graphe)**.** Une molécule est représentée comme un graphe $G = (V, E)$ où :

- Les nœuds $v \in V$ représentent les atomes, avec des features h_v (type atomique, charge, hybridation, etc.).
- Les arêtes $e \in E$ représentent les liaisons chimiques, avec des features h_e (type de liaison, distance, etc.).

Exemple 9.2 (Prédiction de propriétés moléculaires)**.** Le benchmark **MoleculeNet** (Wu et al. 2018) regroupe plusieurs tâches :

1. **ESOL** : prédiction de la solubilité aqueuse (régression).
2. **BBBP** : passage de la barrière hémato-encéphalique (classification binaire).
3. **Tox21** : toxicité sur 12 cibles biologiques (classification multi-tâche).

Les GNN (SchNet, DimeNet, GemNet) obtiennent les meilleurs résultats publiés sur ces benchmarks grâce à leur capacité à encoder la structure moléculaire.

Définition 9.3 (Message passing pour les molécules)**.** Pour un graphe moléculaire, le message passing de SchNet (Schütt et al. 2017) intègre les distances interatomiques :

$$m_{ij} = \phi_{\text{msg}}(h_i, h_j, \|x_i - x_j\|)$$

où $x_i \in \mathbb{R}^3$ sont les positions atomiques. Le filtre continu sur la distance rend le modèle invariant par rotation et translation.

```python
import torch
from torch_geometric.nn import SchNet

# SchNet pour la prediction d'energie moleculaire
model = SchNet(
    hidden_channels=128,
    num_filters=128,
    num_interactions=6,
    num_gaussians=50,
    cutoff=10.0,
)

# z: numeros atomiques, pos: positions 3D, batch: indices
# energy = model(z, pos, batch)
```

9.2 Prédiction de propriétés moléculaires

Théorème 9.4 (Universalité des GNN équivariants pour les molécules)**.** *Un GNN* E(3)*-équivariant avec des features tensorielles de degré suffisant peut approcher toute fonction continue équivariante sur les graphes moléculaires avec positions atomiques.*

Remarque 9.5. Les modèles récents (DimeNet, SphereNet, PaiNN, MACE) utilisent non seulement les distances mais aussi les angles et les dihèdres pour améliorer l'expressivité.

9.3 Analyse de réseaux sociaux

Définition 9.6 (Tâches sur les réseaux sociaux)**.** Sur un réseau social modélisé comme un graphe :

1. **Classification de nœuds** : prédire les intérêts ou la communauté d'un utilisateur.
2. **Prédiction de liens** : prédire les futures connexions entre utilisateurs.
3. **Détection de communautés** : clustering des nœuds.

Exemple 9.7 (Classification de nœuds sur Cora)**.** Le dataset **Cora** contient 2 708 articles scientifiques (nœuds) reliés par des citations (arêtes). Chaque article doit être classé parmi 7 catégories. Un GCN à 2 couches atteint $\sim 81\%$ de précision avec seulement 20 labels par classe.

Proposition 9.8 (Over-smoothing)**.** Pour un GCN avec L couches, les embeddings des nœuds convergent vers un sous-espace de dimension au plus 1 quand $L \to \infty$:

$$\lim_{L\to\infty} h_v^{(L)} = c \cdot \sqrt{d_v} \quad \text{pour tout } v \in V$$

où d_v est le degré du nœud. C'est le phénomène d'**over-smoothing** qui limite la profondeur des GNN.

9.4 Systèmes de recommandation

Définition 9.9 (Filtrage collaboratif sur graphe biparti)**.** Un système de recommandation peut être modélisé comme un graphe biparti $G = (U \cup I, E)$ où U sont les utilisateurs, I les items, et $(u, i) \in E$ si l'utilisateur u a interagi avec l'item i.

Exemple 9.10 (PinSage — Ying et al. 2018)**. PinSage** est un GNN déployé par Pinterest sur un graphe de 3 milliards de nœuds et 18 milliards d'arêtes. Il utilise :

- **Échantillonnage de voisins** : aléatoire pondéré par importance (random walks).
- **Agrégation locale** : similaire à GraphSAGE.
- **Mini-batch training** : sur des sous-graphes échantillonnés.

9.5 Simulation physique

Définition 9.11 (Simulateur appris par GNN)**.** Un simulateur appris modélise un système physique comme un graphe d'interactions :

- Les nœuds représentent les particules (position, vitesse, type).
- Les arêtes représentent les interactions (proches voisins).

Le GNN prédit l'accélération de chaque particule :

$$\ddot{x}_i = f_\theta\big(h_i, \{h_j, x_j - x_i\}_{j\in\mathcal{N}(i)}\big)\,.$$

Exemple 9.12 (GNS — Sanchez-Gonzalez et al. 2020)**.** Le **Graph Network Simulator** (GNS) simule des fluides, matériaux granulaires et solides déformables. Il généralise à des conditions initiales non vues et à des géométries de domaine différentes.

Théorème 9.13 (Conservation d'énergie par construction)**.** *En paramétrant un GNN comme un hamiltonien H_θ et en intégrant les équations de Hamilton :*

$$\dot{q}_i = \frac{\partial H_\theta}{\partial p_i}, \qquad \dot{p}_i = -\frac{\partial H_\theta}{\partial q_i}$$

on obtient un simulateur qui conserve l'énergie par construction (Hamiltonian GNN).

9.6 Prévision météorologique

Exemple 9.14 (GraphCast — Lam et al. 2023)**. GraphCast** (DeepMind) utilise un GNN sur un maillage icosaédrique de la Terre pour la prévision météo à 10 jours. Il surpasse le modèle numérique HRES de l'ECMWF sur 90% des métriques, avec un temps de calcul de **moins d'une minute** contre des heures pour les modèles physiques.

L'architecture utilise :

1. Un **encodeur** grille → maillage multi-résolution.
2. Un **processeur** GNN sur le maillage (16 couches de message passing).
3. Un **décodeur** maillage → grille.

Remarque 9.15. Le succès de GraphCast illustre le potentiel des GNN pour les problèmes définis sur des domaines non planaires (la sphère terrestre).

9.7 Exercices

Exercice 9.1. Pour une molécule de benzène (C_6H_6), construire le graphe moléculaire (nœuds, arêtes, features). Combien de messages sont échangés en une itération de message passing ?

Exercice 9.2. Expliquer pourquoi l'invariance par permutation des nœuds est essentielle pour les graphes moléculaires. Que se passerait-il si on numérotait les atomes différemment ?

Exercice 9.3 (Over-smoothing)**.** Pour un graphe complet K_n avec un GCN sans biais ni non-linéarité, montrer que les features convergent vers un vecteur constant après L couches quand L est grand.

Exercice 9.4 (Implémentation)**.** Implémenter un GNN pour la prédiction de solubilité sur le dataset ESOL avec PyTorch Geometric. Comparer GCN, GAT et SchNet.

Exercice 9.5 (Simulation)**.** Implémenter un simple GNS pour simuler N particules interagissant par un potentiel de Lennard-Jones. Comparer avec une intégration de Verlet classique.

Chapitre 10

Liens avec la TDA

L'apprentissage géométrique et l'analyse topologique des données (TDA) sont deux réponses complémentaires à la même question : comment exploiter la *structure* des données ? Les GNN capturent les relations locales et les symétries ; la TDA extrait des invariants topologiques globaux (nombres de Betti, homologie persistante) qui sont robustes au bruit et invariants par déformation continue. Combiner les deux — par exemple en ajoutant des features topologiques aux représentations d'un GNN, ou en utilisant des couches de persistance différentiables — améliore l'expressivité et la performance. Ce chapitre explore ces connexions, de la théorie (limites d'expressivité des GNN révélées par la topologie) aux applications (filtration apprise, couches de persistance).

Intuition

L'analyse topologique des données (TDA) et l'apprentissage géométrique partagent une préoccupation commune : exploiter la **structure** des données. La TDA apporte des invariants topologiques (nombres de Betti, homologie persistante) qui complètent les features apprises par les GNN et améliorent leur **expressivité**.

10.1 Rappels d'homologie persistante

Définition 10.1 (Homologie persistante)**.** Étant donnée une filtration de complexes simpliciaux $K_0 \subseteq K_1 \subseteq \cdots \subseteq K_m$, l'**homologie persistante** suit l'évolution des classes d'homologie à travers la filtration. Chaque classe naît à un indice b et meurt à un indice $d > b$, donnant un point (b, d) dans le **diagramme de persistance**.

Remarque 10.2. Pour un graphe $G = (V, E)$, on peut construire différentes filtrations :

- **Par degré** : $f(v) = \deg(v)$ et $f(e) = \max(f(u), f(v))$.
- **Par centralité** : $f(v) = \text{betweenness}(v)$.
- **Par features apprises** : $f(v) = g_\theta(h_v)$ où g_θ est un réseau de neurones.

10.2 Features topologiques dans les GNN

Définition 10.3 (Features topologiques pour les graphes)**.** Les **features topologiques** d'un graphe sont extraites de ses diagrammes de persistance :

- H_0 : les composantes connexes et leur stabilité.
- H_1 : les cycles et leur persistance.

Ces features sont vectorisées (paysages, images de persistance) et concaténées aux features du GNN.

Proposition 10.4 (Complémentarité)**.** Les features topologiques capturent des propriétés globales du graphe (cycles, cavités) que le message passing local ne peut pas détecter en un nombre borné d'itérations : détecter un cycle de longueur k nécessite au moins $k/2$ couches de message passing.

```python
import torch
import numpy as np
from gudhi import RipsComplex
from gudhi.representations import PersistenceImage

def topo_features(pos, max_edge=5.0, resolution=10):
    """Calcule les features topologiques d'un nuage de points."""
    rips = RipsComplex(points=pos, max_edge_length=max_edge)
    st = rips.create_simplex_tree(max_dimension=2)
    st.persistence()

    # Diagrammes par dimension
    dgm_h0 = st.persistence_intervals_in_dimension(0)
    dgm_h1 = st.persistence_intervals_in_dimension(1)

    # Vectorisation par images de persistance
    pi = PersistenceImage(
        bandwidth=0.1, resolution=[resolution, resolution]
    )
    feats = []
    for dgm in [dgm_h0, dgm_h1]:
        if len(dgm) > 0:
            dgm = dgm[np.isfinite(dgm[:, 1])]
        if len(dgm) > 0:
            feats.append(pi.fit_transform([dgm])[0])
        else:
            feats.append(np.zeros(resolution ** 2))
    return np.concatenate(feats)
```

10.3 Message passing topologique

Définition 10.5 (Message passing sur complexes simpliciaux)**.** Le **message passing simplicial** (Bodnar et al. 2021) généralise le message passing des graphes aux complexes simpliciaux de dimension supérieure :

$$h_\sigma^{(t+1)} = \phi\left(h_\sigma^{(t)}, \bigoplus_{\tau \in \mathcal{B}(\sigma)} \psi_{\mathcal{B}}(h_\tau^{(t)}), \bigoplus_{\tau \in \mathcal{C}(\sigma)} \psi_{\mathcal{C}}(h_\tau^{(t)})\right)$$

où $\mathcal{B}(\sigma)$ sont les faces (boundary) et $\mathcal{C}(\sigma)$ les co-faces (coboundary) du simplexe σ.

Intuition

Dans un graphe, les messages circulent le long des arêtes (1-simplexes). Dans un complexe simplicial, on ajoute des messages le long des triangles (2-simplexes) et des structures de dimension supérieure. Cela permet de capturer des relations **d'ordre supérieur** entre les nœuds.

Définition 10.6 (Message passing sur complexes cellulaires)**.** Les **CW Networks** (Bodnar et al. 2021) étendent le message passing aux complexes cellulaires, qui généralisent les complexes simpliciaux en autorisant des cellules de forme arbitraire (pas seulement des simplexes).

10.4 Hiérarchie de Weisfeiler-Leman et topologie

Définition 10.7 (Test de Weisfeiler-Leman)**.** Le **test de Weisfeiler-Leman** (WL) de niveau k est un algorithme de raffinement de couleurs qui caractérise les graphes : deux graphes non distingués par k-WL sont dits **k-WL équivalents**.

Théorème 10.8 (Expressivité des GNN — Xu et al. 2019, Morris et al. 2019)**.** *Un GNN de type message passing est* ***au plus aussi puissant*** *que le test* 1*-WL pour distinguer des graphes non isomorphes.*

Théorème 10.9 (Au-delà de WL par la topologie — Bodnar et al. 2021)**.** *Le message passing sur des complexes simpliciaux avec des features topologiques (homologie persistante) peut distinguer des graphes que k-WL ne distingue pas, pour tout k fixé.*

Exemple 10.10 (Graphes de Rook)**.** Les graphes de Rook $R_{3\times3}$ et $R_{4\times4}$ ne sont pas distingués par 3-WL. Cependant, leurs diagrammes de persistance H_1 (avec filtration par degré) diffèrent : $R_{4\times4}$ a davantage de cycles persistants.

10.5 Expressivité et topologie

Proposition 10.11 (Enrichissement topologique)**.** En ajoutant des features d'homologie persistante aux features de nœuds d'un GNN, l'expressivité résultante est **strictement supérieure** à celle du GNN seul pour le problème de classification de graphes.

Définition 10.12 (Filtration apprise jointe)**.** Dans une **filtration apprise jointe**, les valeurs de filtration sont définies par le GNN lui-même :

$$f_\theta(v) = \mathrm{MLP}_\theta(h_v^{(L)})$$

où $h_v^{(L)}$ est l'embedding final du nœud v après L couches de message passing. La persistance est calculée sur cette filtration, et le gradient rétropropagé à travers le calcul de persistance (cf. persistance différentiable).

Pipeline GNN + TDA

$$X \xrightarrow{\text{GNN}} H^{(L)} \xrightarrow{f_\theta} \text{Filtration} \xrightarrow{\text{PH}} \text{Dgm} \xrightarrow{\text{PersLay}} z_{\text{topo}} \xrightarrow{\oplus} \text{Prédiction}$$

10.6 Exercices

Exercice 10.1. Pour le graphe de Petersen, calculer le diagramme de persistance H_0 et H_1 avec la filtration par degré. Tous les nœuds ont le même degré : que peut-on en déduire ?

Exercice 10.2. Montrer que deux graphes isomorphes ont les mêmes diagrammes de persistance pour toute filtration définie à partir de propriétés structurelles du graphe.

Exercice 10.3 (Message passing simplicial)**.** Écrire les équations de message passing pour un triangle (v_1, v_2, v_3) dans un complexe simplicial. Quels messages reçoit l'arête (v_1, v_2) ?

Exercice 10.4. Donner un exemple de deux graphes non isomorphes ayant les mêmes diagrammes de persistance pour la filtration par degré. Proposer une filtration qui les distingue.

Exercice 10.5 (Projet)**.** Implémenter un GNN enrichi de features topologiques sur le benchmark ZINC et comparer avec un GCN de base. Mesurer l'amélioration en MAE.

Chapitre 11

Directions de Recherche

Intuition

L'apprentissage géométrique est un domaine en pleine expansion. Ce chapitre présente les directions de recherche les plus actives et les problèmes ouverts qui définiront les prochaines avancées du domaine.

11.1 Modèles de fondation géométriques

Définition 11.1 (Modèle de fondation géométrique)**.** Un **modèle de fondation géométrique** est un modèle pré-entraîné à grande échelle sur des données structurées (graphes, molécules, nuages de points) et transférable à de multiples tâches en aval par *fine-tuning* ou *prompting*.

Exemple 11.2 (Exemples de modèles de fondation géométriques)**.**

- **GEM** (Fang et al. 2022) : modèle de fondation pour les molécules, pré-entraîné sur 20M de structures.
- **Uni-Mol** (Zhou et al. 2023) : représentation unifiée 2D/3D pour les molécules.
- **Point-MAE** (Pang et al. 2022) : auto-encodeur masqué pour les nuages de points 3D.

Remarque 11.3. Contrairement aux modèles de fondation textuels (GPT, Claude), les modèles géométriques doivent respecter les **symétries** des données (invariance par permutation, équivariance par rotation). Cela impose des contraintes architecturales fortes.

Proposition 11.4 (Défis du scaling)**.** Le passage à l'échelle des GNN pose des problèmes spécifiques :

1. **Explosion du voisinage** : k couches impliquent $\mathcal{O}(d^k)$ voisins (d = degré moyen).
2. **Over-smoothing** : les représentations convergent avec la profondeur.
3. **Over-squashing** : l'information longue distance est compressée dans des bottlenecks.
4. **Graphes de taille variable** : le batching est non trivial.

11.2 Transformers équivariants

Définition 11.5 (Transformer géométrique)**.** Un **Transformer géométrique** est un Transformer dont l'attention intègre des informations géométriques tout en respectant les symétries :

$$\alpha_{ij} = \frac{\exp\left(\frac{q_i^\top k_j}{\sqrt{d}} + b(x_i, x_j)\right)}{\sum_l \exp\left(\frac{q_i^\top k_l}{\sqrt{d}} + b(x_i, x_l)\right)}$$

où $b(x_i, x_j)$ est un biais d'attention dépendant de la géométrie (distance, angle, orientation relative).

Exemple 11.6 (Exemples d'architectures)**.**

- **SE(3)-Transformer** (Fuchs et al. 2020) : attention SE(3)-équivariante avec features tensorielles.
- **Equiformer** (Liao & Smidt, 2023) : Transformer équivariant avec produits tensoriels dans l'attention.
- **GPS** (Rampášek et al. 2022) : combine message passing local et attention globale.

Théorème 11.7 (Expressivité des Graph Transformers)**.** *Un Graph Transformer avec encodage positionnel spectral et attention globale est* ***strictement plus expressif*** *que le* 1*-WL test. Avec des encodages positionnels* SE(3)*-équivariants, il peut approcher toute fonction équivariante continue.*

Attention équivariante

$$h_i' = \sum_j \alpha_{ij} \cdot V(h_j, x_j - x_i)$$

où $V : \mathbb{R}^d \times \mathbb{R}^3 \to \mathbb{R}^{d'}$ est une fonction de valeur qui dépend de la position relative, et α_{ij} est invariant par SE(3).

11.3 Modèles de diffusion sur variétés

Définition 11.8 (Diffusion riemannienne)**.** Un **modèle de diffusion riemannien** (De Bortoli et al. 2022) généralise les modèles de diffusion (DDPM) aux variétés riemanniennes $(\mathcal{M}, g)$:

1. **Processus forward** : mouvement brownien riemannien $dX_t = \sqrt{2\beta_t}\, dB_t^{\mathcal{M}}$.
2. **Score matching** : apprentissage du score $\nabla_{\mathcal{M}} \log p_t(x)$ sur la variété.
3. **Processus reverse** : intégration de l'équation reverse sur $\mathcal{M}$ avec le score appris.

Exemple 11.9 (Applications de la diffusion géométrique)**.**

- **Génération de molécules 3D** : diffusion sur $\mathbb{R}^{3n}$ avec équivariance SE(3) (EDM, Hoogeboom et al. 2022).
- **Génération de conformations** : diffusion sur les angles de torsion (variété torique).
- **Génération de protéines** : diffusion sur $\mathrm{SO}(3)^n$ pour les frames des résidus (RFDiffusion, Watson et al. 2023).

Difficultés techniques

La diffusion sur variétés nécessite :

- La carte exponentielle $\exp_x : T_x\mathcal{M} \to \mathcal{M}$ pour les pas d'intégration.
- Le logarithme riemannien $\log_x : \mathcal{M} \to T_x\mathcal{M}$ pour le score.
- Le transport parallèle pour déplacer les vecteurs entre espaces tangents.

11.4 Apprentissage auto-supervisé géométrique

Définition 11.10 (Apprentissage auto-supervisé sur graphes)**.** L'**apprentissage auto-supervisé** (SSL) sur graphes apprend des représentations sans labels via des tâches auxiliaires :

1. **Contrastif** : maximiser la similarité entre deux vues augmentées du même graphe (GraphCL, GCA).
2. **Prédictif** : prédire des propriétés masquées (features de nœuds, arêtes, sous-graphes).
3. **Génératif** : reconstruire le graphe à partir d'une représentation compressée (GraphMAE).

Proposition 11.11 (Augmentations géométriques)**.** Les augmentations pour les graphes incluent :

- Suppression aléatoire de nœuds ou d'arêtes.
- Perturbation des features.
- Sous-graphe aléatoire.
- Diffusion de chaleur (lissage des features).

Pour les données 3D, on ajoute : rotation, translation, bruit gaussien, échantillonnage partiel.

Théorème 11.12 (Garanties du SSL contrastif)**.** *Sous des hypothèses de régularité sur la distribution des données et des augmentations, les représentations contrastives sont* ***linéairement séparables*** *pour les tâches en aval, avec une borne d'erreur contrôlée par l'alignement et l'uniformité des représentations.*

11.5 Problèmes ouverts

Remarque 11.13 (Problèmes ouverts majeurs).

1. **Expressivité optimale** : quelle est l'architecture GNN minimale pour égaler k-WL pour un k donné ?
2. **Over-squashing** : comment permettre l'information longue distance sans attention globale coûteuse ($\mathcal{O}(n^2)$) ?

3. **Généralisation hors distribution** : les GNN généralisent-ils à des graphes de structures très différentes de l'entraînement ?
4. **Équivariance approchée** : comment gérer les symétries approximatives (quasi-symétries) des données réelles ?
5. **Interprétabilité** : comment expliquer les décisions d'un GNN en termes de sous-structures significatives ?
6. **Passage à l'échelle** : comment entraîner des GNN sur des graphes à des milliards de nœuds ?

Définition 11.14 (Over-squashing)**.** L'**over-squashing** désigne le phénomène où l'information provenant de nœuds distants est compressée de manière exponentielle en traversant les couches de message passing. Formellement, le jacobien :

$$\left\| \frac{\partial h_v^{(L)}}{\partial h_u^{(0)}} \right\| \leq C \cdot \lambda_{\max}^{L} \cdot \prod_{\ell=1}^{L} \mathrm{Lip}(\sigma_\ell)$$

décroît exponentiellement avec la distance dans le graphe si $\lambda_{\max} < 1$.

11.6 Exercices

Exercice 11.1. Expliquer pourquoi les encodages positionnels laplaciens ne sont pas uniques (signe, ordre). Proposer une solution pour les rendre invariants.

Exercice 11.2. Pour un graphe moléculaire, proposer trois augmentations contrastives qui préservent l'identité chimique et trois qui ne la préservent pas. Justifier.

Exercice 11.3 (Over-squashing)**.** Calculer la borne sur $\left\| \frac{\partial h_v^{(L)}}{\partial h_u^{(0)}} \right\|$ pour un graphe en chemin de longueur L avec un GCN à poids unitaires et activation ReLU.

Exercice 11.4. Montrer que le mouvement brownien sur la sphère S^2 converge vers la distribution uniforme. Quel est le taux de convergence en fonction de la première valeur propre non nulle du laplacien ?

Exercice 11.5 (Projet)**.** Implémenter un modèle de diffusion équivariant simple pour générer des nuages de points 3D (par exemple, des formes de la base ShapeNet). Évaluer la qualité par la Chamfer distance et les nombres de Betti.

Bibliographie

[1] Bronstein, M.M. et al., *Geometric Deep Learning : Grids, Groups, Graphs, Geodesics, and Gauges*, arXiv :2104.13478, 2021.

[2] Hamilton, W.L., *Graph Representation Learning*, Morgan & Claypool, 2020.

[3] Kipf, T.N. and Welling, M., Semi-Supervised Classification with Graph Convolutional Networks, *ICLR*, 2017.

[4] Veličković, P. et al., Graph Attention Networks, *ICLR*, 2018.