Geometric Learning

Lecture Notes

Master M2 — 2025–2026

Yaë Ulrich Gaba

"Geometry is the art of reasoning well from badly drawn figures."
— Henri Poincaré

March 25, 2026

Contents

Preface

Why This Course?

Deep learning has revolutionized many areas of artificial intelligence: computer vision, natural language processing, speech recognition. However, classical architectures—fully connected networks, convolutional networks on regular grids—rest on strong structural assumptions: that data lives on regular Euclidean grids.

Yet an ever-growing number of fundamental problems in science and engineering involve data with **non-Euclidean** structure:

- **Graphs**: social networks, molecules, transportation networks, knowledge graphs.
- **Manifolds**: 3D surfaces, configuration spaces in robotics, spherical data (meteorology, astrophysics).
- **Point clouds**: LiDAR data, 3D reconstruction, molecular geometry.
- **Simplicial complexes**: higher-order relations, applied algebraic topology.

Geometric Deep Learning (GDL) is the unifying framework that extends deep learning principles to non-Euclidean domains by drawing on the intrinsic **symmetries** and **invariances** of data.

The Unifying Principle: Symmetries

The guiding thread of this course is the **equivariance principle**. The fundamental idea, inherited from theoretical physics and Felix Klein's Erlangen Programme, is that interesting mathematical structures are characterized by their **symmetry groups**.

The Erlangen Programme for deep learning

Just as Klein proposed classifying geometries by their transformation groups, geometric deep learning classifies neural network architectures by the symmetries they respect:

Domain	**Symmetry**	**Architecture**
Regular grid	Translation	CNN
Set	Permutation	Deep Sets
Graph	Node permutation	GNN
Sphere	Rotations SO(3)	Spherical CNN
3D space	SE(3)	SE(3)-Transformer

Target Audience

This course is designed for **Master's** and **PhD** students in applied mathematics, computer science, or physics, with prerequisites in:

- **Linear algebra**: vector spaces, eigenvalues, spectral decomposition.
- **Deep learning**: neural networks, backpropagation, gradient descent optimization.
- **Probability and statistics**: probability measures, expectation, estimation.
- **Python programming**: proficiency with PyTorch or an equivalent framework.

Notions of group theory, differential geometry, and algebraic topology are desirable but will be introduced in the course.

Course Organization

The course is structured in eleven chapters, progressing from mathematical foundations to applications and current research:

1. **Motivations**: why geometry matters in deep learning.
2. **Group theory**: symmetries, representations, equivariance.
3. **Graph Neural Networks (GNN)**: message passing, aggregation.
4. **GNN Variants**: GAT, GIN, GraphSAGE.
5. **Spectral methods**: graph Laplacian, spectral convolutions.
6. **Learning on manifolds**: differential geometry, intrinsic operators.
7. **Equivariant and invariant networks**: general formalism.
8. **Point clouds**: PointNet and extensions.
9. **Applications**: chemistry, physics, biology, 3D vision.
10. **Connections to TDA**: persistent homology, topological representations.
11. **Research directions**: current trends and open problems.

Conventions and Notation

Notation	**Meaning**
$G = (V, E)$	Graph with vertex set V and edge set E
$\mathbf{A}$	Adjacency matrix
$\mathbf{D}$	Degree matrix
$\mathbf{L} = \mathbf{D} - \mathbf{A}$	Combinatorial Laplacian
$\tilde{\mathbf{L}} = \mathbf{D}^{-1/2}\mathbf{L}\mathbf{D}^{-1/2}$	Normalized Laplacian
$\mathbf{X} \in \mathbb{R}^{n\times d}$	Node feature matrix
$\mathbf{h}_v^{(\ell)}$	Representation of node v at layer ℓ
$\mathcal{N}(v)$	Neighborhood of node v
$\rho : G \to \mathrm{GL}(V)$	Representation of a group G
$\mathcal{M}$	Differentiable manifold
$T_p\mathcal{M}$	Tangent space at p
$\|\cdot\|$	Norm
$\langle \cdot, \cdot \rangle$	Inner product
σ	Activation function (ReLU, etc.)
$\bigoplus$	Aggregation (sum, mean, max)

The 5G Framework of Bronstein et al.

The unifying framework of geometric deep learning rests on five fundamental domains, the "5Gs":

Definition 0.1 (The 5Gs)**.** The five domains of geometric deep learning are:

1. **Grids**: data on regular grids (images, time series).
2. **Groups**: global symmetries acting on data.
3. **Graphs**: relational data with permutation invariance.
4. **Geodesics**: data on curved manifolds.
5. **Gauges**: fiber bundles and connections for parallel transport.

Remark 0.2 (Unification). Each of these domains can be seen as a special case of a general framework where a **symmetry group** $\mathfrak{G}$ acts on a **domain** Ω and we seek functions $f : \mathcal{X}(\Omega) \to \mathcal{Y}$ that are **equivariant** with respect to the action of $\mathfrak{G}$:

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

Key References

This course draws on several foundational works and articles:

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

Software

Practical sessions primarily use:

- **PyTorch** (`torch`)—deep learning framework.
- **PyTorch Geometric** (`torch_geometric`)—graph learning library.
- **NetworkX**—graph manipulation and visualization.
- **Open3D**—3D point cloud processing.
- **GUDHI / Ripser**—persistent homology computation.

Environment verification

```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 available: {torch.cuda.is_available()}")

# Quick test: create a PyG graph
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 nodes, 16 features
data = Data(x=x, edge_index=edge_index)
print(f"Graph: {data.num_nodes} nodes, {data.num_edges} edges")
```

Acknowledgments

This course has benefited from the contributions of many researchers and students. We particularly thank the authors of the *Geometric Deep Learning Blueprint* for establishing the unifying conceptual framework that structures this entire course.

[Author Name]
March 2026

Chapter 1

Motivations — Why Geometry in DL

Classical neural networks — MLPs, CNNs, RNNs — are designed for data living on regular grids: images on pixels, time series on uniformly spaced steps. But what do you do when data lives on a graph (social networks, molecules), on a curved surface (3D models), or on a symmetry group (poses, rotations)? This is the question that *geometric deep learning* answers: an architectural revolution, driven by the work of Bronstein, Bruna, Cohen, and others, that places geometry back at the heart of deep learning.

1.1 Limitations of Classical Architectures

Convolutional neural networks (CNNs) have achieved spectacular success in computer vision, signal processing, and speech recognition. Their power stems from a fundamental **inductive bias**: exploiting the regular grid structure of data.

Definition 1.1 (Inductive bias)**.** An **inductive bias** is a set of prior assumptions built into a model's architecture to guide learning. For a CNN:

1. **Locality**: each neuron sees only a local neighborhood.
2. **Weight sharing**: the same filter is applied everywhere.
3. **Translation invariance**: pattern detection is independent of position.

However, many real-world data do **not** have grid structure:

Example 1.2 (Non-Euclidean data)**.**

1. A **molecule** is a graph of atoms connected by chemical bonds.
2. A **social network** is a graph of people connected by relationships.
3. The **Earth's surface** is a sphere, not a plane.
4. A **3D point cloud** from a LiDAR sensor has no canonical ordering.

Why not simply vectorize?

One might be tempted to represent a graph by flattening its adjacency matrix into a vector and using an MLP. This approach fails because:

- The representation depends on the **arbitrary numbering** of nodes.
- The number of parameters explodes: $\mathcal{O}(n^2)$ for n nodes.
- No **generalization** to graphs of different sizes.

1.2 Geometry as an Organizing Principle

1.2.1 The Erlangen Programme

In 1872, Felix Klein proposed in his *Erlanger Programm* to classify geometries by their symmetry groups. This profound idea finds a direct application in deep learning.

Theorem 1.3 (Erlangen Programme — informal version)**.** *A geometry is entirely determined by its* ***transformation group*** *G and the properties that are* ***invariant*** *under the action of G.*

Example 1.4 (Geometries and their groups)**.**

Geometry	**Group**	**Invariants**
Euclidean	Isometries $\mathrm{E}(n)$	Distances, angles
Affine	Affine transformations	Parallelism, ratios
Projective	Projective transformations	Cross-ratio

1.2.2 From CNNs to GNNs: From Grids to Graphs

A CNN can be viewed as a special case of a network operating on a regular graph (the pixel grid), with the translation group $\mathbb{Z}^2$ as its symmetry group.

Discrete convolution on grid vs. graph

Grid convolution:

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

Graph convolution (message passing):

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

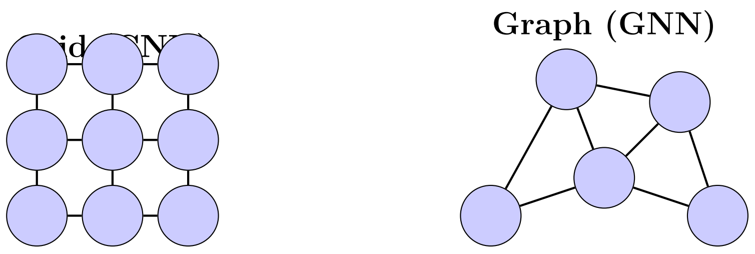

Figure 1.1: From a regular grid (CNN) to an irregular graph (GNN).

1.3 The Unifying Framework: Geometric Deep Learning

1.3.1 The Five Gs

Bronstein et al. (2021) proposed a unifying framework organized around five fundamental domains:

Definition 1.5 (GDL Domains)**.** 1. **Grids** ($\Omega = \mathbb{Z}^d$): translation symmetry.

2. **Groups** ($\Omega = G$): symmetry by the group itself.
3. **Graphs** ($\Omega = (V, E)$): permutation symmetry S_n.
4. **Geodesics** ($\Omega = \mathcal{M}$): isometry invariance.
5. **Gauges** ($\Omega = \mathcal{M}$ with bundle): gauge equivariance.

1.3.2 The General Equivariance Principle

Theorem 1.6 (Equivariance Principle)**.** *Let $\mathfrak{G}$ be a group acting on a signal space $\mathcal{X}(\Omega)$ via representation $\rho_{\mathcal{X}}$ and on a target space $\mathcal{Y}$ via $\rho_{\mathcal{Y}}$. A function $f : \mathcal{X}(\Omega) \to \mathcal{Y}$ is **equivariant** if:*

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

*When $\rho_{\mathcal{Y}}$ is the trivial representation, f is **invariant**.*

Equivariance in pictures

Consider image classification:

- Convolutional layers are **equivariant** to translation: if the input shifts, the feature maps shift accordingly.
- Global pooling produces an **invariant** output: the result does not depend on the object's position.

The same principle applies to GNNs: message-passing layers are equivariant to node permutations, and global readout is invariant.

1.4 The Importance of Permutation Invariance

1.4.1 Formalization

For a graph with n nodes, a **permutation** $\pi \in S_n$ reorders the nodes. This acts on:

- The adjacency matrix: $\mathbf{A} \mapsto \mathbf{P}_\pi \mathbf{A} \mathbf{P}_\pi^\top$
- The features: $\mathbf{X} \mapsto \mathbf{P}_\pi \mathbf{X}$

where $\mathbf{P}_\pi$ is the permutation matrix associated with π.

Definition 1.7 (Invariant / equivariant functions on graphs)**.** A graph-level function $f : (\mathbf{A}, \mathbf{X}) \mapsto y$ is **permutation invariant** if:

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

A node-level function $f : (\mathbf{A}, \mathbf{X}) \mapsto \mathbf{H}$ is **permutation equivariant** if:

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

1.4.2 Deep Sets: The Simplest Case

Theorem 1.8 (Zaheer et al., 2017)**.** *A function $f : 2^{\mathcal{X}} \to \mathbb{R}$ is permutation-invariant and continuous if and only if it can be written as:*

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

for continuous functions $\phi : \mathcal{X} \to \mathbb{R}^d$ and $\rho : \mathbb{R}^d \to \mathbb{R}$.

Deep Sets implementation in PyTorch

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

class DeepSets(nn.Module):
    """Permutation-invariant model for sets."""
    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        self.phi = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
        )
        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"Output: {out.shape}")  # (8, 10)

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

1.5 Motivating Examples

1.5.1 Molecular Property Prediction

Molecular property prediction is one of the flagship applications of GDL. A molecule is naturally represented as a graph:

- **Nodes**: atoms, with features (atomic type, charge, etc.).
- **Edges**: chemical bonds, with features (bond order, etc.).

Proposition 1.9 (Desired properties)**.** A model for molecular prediction should be:

1. **Permutation invariant** over atoms (no canonical numbering).
2. **Rotation and translation invariant** (energy is orientation-independent).
3. **Extensive / intensive** depending on the property (energy vs. HOMO-LUMO gap).

1.5.2 Weather on the Sphere

Weather data naturally lives on the sphere S^2. Classical CNNs cannot be directly applied because:

- Planar projections introduce **distortions** (Mercator, etc.).
- The pixel grid is not uniform on the sphere.
- Physical symmetries (rotations) are not respected.

Remark 1.10 (Spherical CNNs)*.* *Spherical CNNs* (Cohen et al., 2018) define convolutions directly on S^2 using spherical harmonics as a spectral basis, thus respecting SO(3) equivariance.

CHAPTER 1. MOTIVATION—WHY GEOMETRY IN DL

1.5.3 Particle Dynamics

Simulation of particle systems (fluids, granular materials) benefits from GDL because:

- Particles form an unordered set.
- Physical laws are equivariant under SE(3).
- Interactions are local (neighborhood graph).

1.6 Panorama of Geometric Architectures

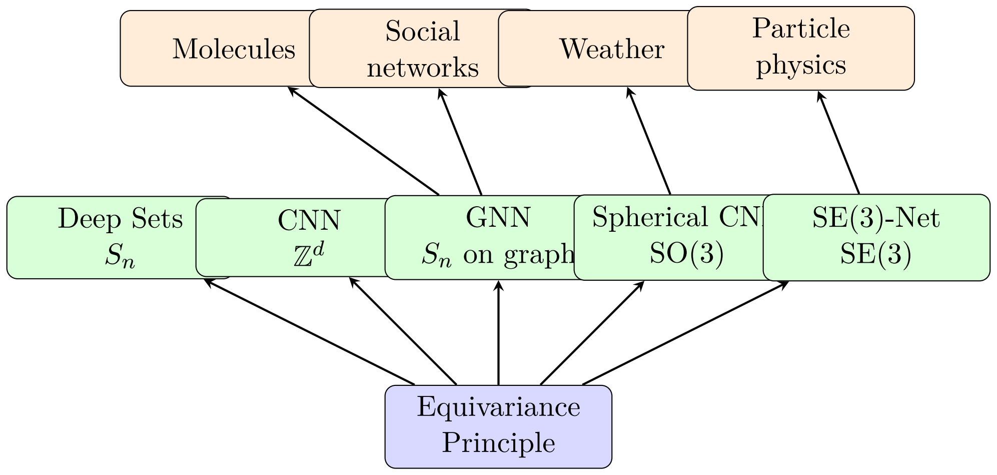

Figure 1.2: Panorama of geometric architectures and their applications.

1.7 Why Symmetries Improve Learning

Exploiting symmetries improves learning in three complementary ways:

Theorem 1.11 (Benefits of equivariance)**.** *Let $\mathfrak{G}$ be the symmetry group of a problem. An equivariant model benefits from:*

1. ***Sample complexity reduction**: the number of required examples is reduced by a factor of $|\mathfrak{G}|$ (or the group dimension for continuous groups).*
2. ***Improved generalization**: generalization to all transformations in $\mathfrak{G}$ is guaranteed "for free."*
3. ***Weight sharing**: parameter count is reduced, avoiding overfitting.*

Proposition 1.12 (Generalization bound)**.** For a model parameterized by $\theta \in \Theta$, the Rademacher complexity of the equivariant function class $\mathcal{F}_{\text{equiv}}$ satisfies:

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

where $\mathcal{F}$ is the unconstrained class.

1.8 From Theory to Practice

First GNN with PyTorch Geometric

```
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
        # Permutation-equivariant layers
        x = F.relu(self.conv1(x, edge_index))
        x = F.relu(self.conv2(x, edge_index))
        # Permutation-invariant readout
        x = global_mean_pool(x, batch)
        return self.lin(x)

# Load a graph classification dataset
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)

# Training loop
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}")
```

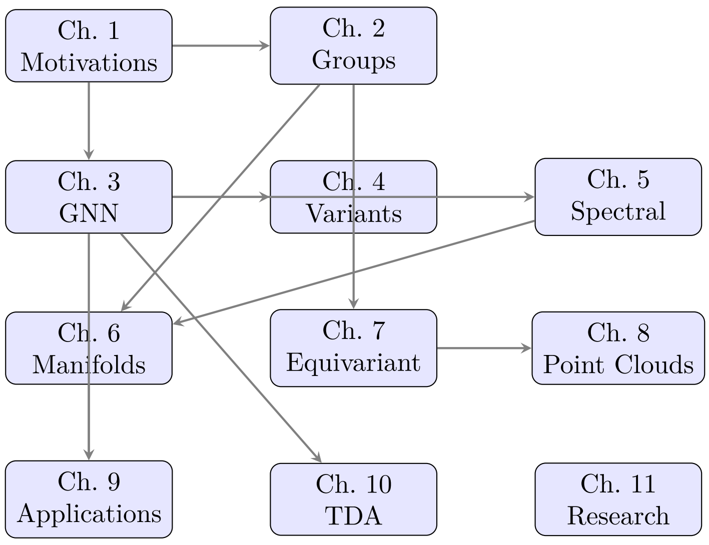

Figure 1.3: Dependency graph between chapters.

1.9 Course Outline and Dependencies

Exercises

Exercise 1.1 (Invariance vs. equivariance)**.** Let $f : \mathbb{R}^{n\times d} \to \mathbb{R}^{n\times d'}$ be defined by $f(\mathbf{X}) = \sigma(\mathbf{XW})$ where $\mathbf{W} \in \mathbb{R}^{d\times d'}$.

1. Show that f is equivariant to permutation of the rows of $\mathbf{X}$.
2. Propose a modification of f to obtain an invariant function $g : \mathbb{R}^{n\times d} \to \mathbb{R}^{d'}$.
3. Implement both functions in PyTorch and verify experimentally.

Exercise 1.2 (Limitations of vectorization)**.** Consider a random graph $G \sim$ Erdős-Renyi(n, p).

1. How many different matrix representations $\mathbf{A}$ correspond to the same (isomorphic) graph?
2. Deduce why an MLP applied to vec$(\mathbf{A})$ has exponential sample complexity.
3. Show that a GNN avoids this problem by construction.

Exercise 1.3 (Deep Sets for symmetric function approximation)**.** Implement a Deep Sets model to approximate the function $f(\{x_1, \ldots, x_n\}) = \max_i x_i - \min_i x_i$ (the range of a set).

1. Train the model on sets of variable size $n \in [5, 50]$.
2. Evaluate generalization to $n = 100$.
3. Compare with a max aggregation instead of sum.

Chapter 2

Group Theory for Deep Learning

2.1 Introduction to Group Theory

Symmetry is everywhere. A snowflake looks the same after a $60°$ rotation. The laws of physics do not change if you translate your laboratory ten metres to the left. A molecule's chemical properties depend on which rearrangements of its atoms leave its structure invariant. But what *is* a symmetry, precisely? The answer was forged in the early nineteenth century by two brilliant and tragically short-lived mathematicians: Évariste Galois, who died in a duel at twenty, and Niels Henrik Abel, who succumbed to tuberculosis at twenty-six. Their legacy is *group theory*—the mathematical language of symmetry.

A group is nothing more than a set of transformations that can be composed, reversed, and that include "doing nothing." Yet from this minimal definition springs an astonishingly rich theory. For geometric deep learning, groups are the key to building neural networks that *respect* the symmetries of their input data: if rotating an image should not change its classification, the network's architecture should encode rotational symmetry from the start. This chapter builds the algebraic foundations that make such constructions possible.

Definition 2.1 (Group)**.** A **group** is a set G equipped with a binary operation $\cdot : G \times G \to G$ satisfying:

1. **Associativity**: $(g \cdot h) \cdot k = g \cdot (h \cdot k)$ for all $g, h, k \in G$.
2. **Identity**: there exists $e \in G$ such that $e \cdot g = g \cdot e = g$ for all $g \in G$.
3. **Inverse**: for every $g \in G$, there exists $g^{-1} \in G$ such that $g \cdot g^{-1} = g^{-1} \cdot g = e$.

Example 2.2 (Fundamental groups in GDL)**.**
1. **Symmetric group** S_n: permutations of n elements. $|S_n| = n!$.
2. **Cyclic group** $\mathbb{Z}_n = \mathbb{Z}/n\mathbb{Z}$: discrete rotations.
3. **Orthogonal group** $\mathrm{O}(n)$: matrices $\mathbf{Q} \in \mathbb{R}^{n \times n}$ with $\mathbf{Q}^\top \mathbf{Q} = \mathbf{I}$.
4. **Special orthogonal group** $\mathrm{SO}(n)$: rotations, $\det(\mathbf{Q}) = +1$.
5. **Euclidean group** $\mathrm{E}(n) = \mathrm{O}(n) \ltimes \mathbb{R}^n$: rotations, reflections, and translations.
6. **Special Euclidean group** $\mathrm{SE}(n) = \mathrm{SO}(n) \ltimes \mathbb{R}^n$: rotations and translations (rigid motions).

CHAPTER 2. GROUP THEORY FOR DEEP LEARNING

2.1.1 Subgroups and Homomorphisms

Definition 2.3 (Subgroup)**.** A subset $H \subseteq G$ is a **subgroup** of G (denoted $H \leq G$) if H is itself a group under the operation of G.

Definition 2.4 (Group homomorphism)**.** A **homomorphism** of groups is a map $\varphi : G \to H$ such that $\varphi(g_1 \cdot g_2) = \varphi(g_1) \cdot \varphi(g_2)$ for all $g_1, g_2 \in G$. If φ is bijective, it is an **isomorphism**.

Proposition 2.5 (Properties of homomorphisms)**.** Let $\varphi : G \to H$ be a homomorphism. Then:

1. $\varphi(e_G) = e_H$.
2. $\varphi(g^{-1}) = \varphi(g)^{-1}$ for all $g \in G$.
3. $\ker(\varphi) = \{g \in G : \varphi(g) = e_H\}$ is a normal subgroup of G.

2.2 Group Actions

Definition 2.6 (Group action)**.** A (left) **action** of a group G on a set X is a map $\cdot : G \times X \to X$ such that:

1. $e \cdot x = x$ for all $x \in X$.
2. $(g \cdot h) \cdot x = g \cdot (h \cdot x)$ for all $g, h \in G$ and $x \in X$.

Example 2.7 (Actions in deep learning)**.**
1. S_n acts on $\mathbb{R}^n$ by permuting coordinates: $\pi \cdot (x_1, \ldots, x_n) = (x_{\pi^{-1}(1)}, \ldots, x_{\pi^{-1}(n)})$.
2. SO(3) acts on $\mathbb{R}^3$ by rotation: $R \cdot \mathbf{x} = R\mathbf{x}$.
3. $\mathbb{Z}^2$ acts on images by translation: $\mathbf{t} \cdot f(\mathbf{x}) = f(\mathbf{x} - \mathbf{t})$.

Definition 2.8 (Orbit and stabilizer)**.** Let G act on X. For $x \in X$:

- The **orbit** of x is $G \cdot x = \{g \cdot x : g \in G\}$.
- The **stabilizer** of x is $G_x = \{g \in G : g \cdot x = x\}$.

Theorem 2.9 (Orbit–stabilizer theorem)**.** *For a finite group G acting on X and $x \in X$:*

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

2.3 Group Representations

Definition 2.10 (Representation)**.** A **representation** of a group G on a vector space V is a homomorphism $\rho : G \to \mathrm{GL}(V)$, i.e., a map assigning to each $g \in G$ an invertible linear transformation $\rho(g) : V \to V$ such that $\rho(g_1 g_2) = \rho(g_1)\rho(g_2)$.

Example 2.11 (Common representations)**.**
1. **Trivial representation**: $\rho(g) = \mathrm{Id}$ for all g.
2. **Permutation representation**: $\rho(\pi) = \mathbf{P}_\pi$ (permutation matrix).
3. **Standard representation of** SO(3): $\rho(R) = R$ (the rotation matrix itself).
4. **Regular representation**: $[\rho(g)f](h) = f(g^{-1}h)$.

2.3.1 Irreducible Representations

Definition 2.12 (Irreducible representation)**.** A representation $\rho : G \to \mathrm{GL}(V)$ is **irreducible** if the only subspaces of V invariant under the action of G are $\{0\}$ and V.

Theorem 2.13 (Maschke's theorem)**.** *Every finite-dimensional representation of a finite group (over $\mathbb{R}$ or $\mathbb{C}$) can be decomposed as a direct sum of irreducible representations:*

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

where each V_i is an irreducible subspace.

Irreducible representations of SO(3)

The irreducible representations of SO(3) are indexed by $\ell \in \{0, 1, 2, \ldots\}$ and have dimension $2\ell + 1$:

- $\ell = 0$: scalars (dimension 1)—trivial representation.
- $\ell = 1$: vectors (dimension 3)—standard representation.
- $\ell = 2$: traceless symmetric tensors (dimension 5).

The representation matrices are the **Wigner D-matrices** $D^{\ell}_{mm'}(R)$, related to spherical harmonics:

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

2.4 Equivariance and Invariance

Definition 2.14 (Equivariant map)**.** Let $\rho_1 : G \to \mathrm{GL}(V_1)$ and $\rho_2 : G \to \mathrm{GL}(V_2)$ be two representations. A linear map $T : V_1 \to V_2$ is **equivariant** (or an **intertwiner**) if:

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

When ρ_2 is the trivial representation, T is **invariant**.

Theorem 2.15 (Schur's lemma)**.** *Let ρ_1 and ρ_2 be two irreducible representations of G and $T : V_1 \to V_2$ an intertwiner:*

1. *If $\rho_1 \ncong \rho_2$, then $T = 0$.*
2. *If $\rho_1 \cong \rho_2$ (over $\mathbb{C}$), then $T = \lambda\,\mathrm{Id}$ for some $\lambda \in \mathbb{C}$.*

Importance of Schur's lemma

Schur's lemma is fundamental because it **strongly constrains** the form of equivariant layers in a neural network. If equivariance is imposed, the space of allowed linear maps is drastically reduced, which justifies weight sharing.

2.5 Tensor Products of Representations

Definition 2.16 (Tensor product)**.** The **tensor product** of two representations $\rho_1 : G \to \mathrm{GL}(V_1)$ and $\rho_2 : G \to \mathrm{GL}(V_2)$ is the representation $\rho_1 \otimes \rho_2 : G \to \mathrm{GL}(V_1 \otimes V_2)$ defined by:

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

Theorem 2.17 (Clebsch–Gordan decomposition for SO(3))**.** *The tensor product of two irreducible representations of* SO(3) *decomposes as:*

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

The change-of-basis coefficients are the ***Clebsch–Gordan coefficients*** $C^{\ell m}_{\ell_1 m_1, \ell_2 m_2}$.

Tensor product with e3nn

```
import torch
from e3nn import o3

# Irreducible representations of SO(3)
irreps_1 = o3.Irreps("1x0e + 1x1o")  # scalar + vector
irreps_2 = o3.Irreps("1x1o")           # vector

# Tensor product
tp = o3.FullTensorProduct(irreps_1, irreps_2)
print(f"Input 1: {irreps_1}")
print(f"Input 2: {irreps_2}")
print(f"Output: {tp.irreps_out}")
# 0e x 1o -> 1o, 1o x 1o -> 0e + 1e + 2e

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

2.6 Lie Groups

Definition 2.18 (Lie group)**.** A **Lie group** is a group G that is also a smooth manifold, such that the multiplication $(g, h) \mapsto gh$ and inversion $g \mapsto g^{-1}$ operations are smooth.

Example 2.19 (Lie groups in GDL)**.**

Group	**Dimension**	**Compact?**	**Usage**
SO(2)	1	Yes	2D rotations
SO(3)	3	Yes	3D rotations
SE(3)	6	No	Rigid motions
O(3)	3	Yes	Rotations + reflections
$\mathrm{GL}(n, \mathbb{R})$	n^2	No	Linear transformations

2.6.1 Lie Algebra

Definition 2.20 (Lie algebra)**.** The **Lie algebra** $\mathfrak{g}$ of a Lie group G is the tangent space to G at the identity e, equipped with the **Lie bracket** $[\cdot,\cdot] : \mathfrak{g} \times \mathfrak{g} \to \mathfrak{g}$.

Exponential map

The **exponential map** $\exp : \mathfrak{g} \to G$ connects the Lie algebra to the group:

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

For SO(3), the Lie algebra $\mathfrak{so}(3)$ is the space of 3×3 skew-symmetric matrices. Rodrigues' formula gives:

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

where $[\hat{\mathbf{n}}]_\times$ is the skew-symmetric matrix of the unit vector $\hat{\mathbf{n}}$.

2.7 Group Convolution

Definition 2.21 (Group convolution)**.** Let G be a locally compact group with Haar measure μ. The **group convolution** of two functions $f, g : G \to \mathbb{R}$ is:

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

Proposition 2.22 (Equivariance of convolution)**.** Group convolution is equivariant to left translation: if $L_a f(x) = f(a^{-1}x)$, then:

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

This property justifies the use of convolution as the fundamental layer in equivariant networks.

Theorem 2.23 (CNN as convolution on $\mathbb{Z}^2$)**.** *A classical convolutional network performs a group convolution on $G = \mathbb{Z}^2$ (the discrete translation group). The first layer performs a cross-correlation:*

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

which is equivariant to translations of f.

2.8 Group Equivariant CNNs (G-CNNs)

Cohen and Welling (2016) generalized CNNs by replacing the translation group $\mathbb{Z}^2$ with a larger group G.

Definition 2.24 (G-CNN)**.** A **G-CNN** is a network whose layers perform convolutions on a group G. The first layer *lifts* the signal from $\mathbb{Z}^2$ to G:

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

Subsequent layers operate entirely on G:

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

G-CNN for group $p4$ (90° rotations)

```
import torch
import torch.nn as nn

class P4Conv(nn.Module):
    """Equivariant convolution for group p4 (0/90/180/270 rotations)."""
    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):
        """Rotate kernel by k*90 degrees."""
        return torch.rot90(w, k, dims=[-2, -1])

    def forward(self, x):
        # x: (batch, C_in, H, W) or (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 on group dimension
        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"Output: {out.shape}")  # (2, 16, 4, 32, 32)
```

2.9 Linear Invariants and Equivariants

Theorem 2.25 (Characterization of equivariant linear maps)**.** *The space of linear maps $T : V_1 \to V_2$ that are equivariant with respect to representations ρ_1 and ρ_2 is:*

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

Its dimension is given by:

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

where $\chi_i(g) = \mathrm{tr}(\rho_i(g))$ is the ***character*** *of ρ_i.*

Corollary 2.26 (Number of free parameters)**.** *For an equivariant layer between two representations decomposed into irreducibles $V_1 = \bigoplus_i n_i V_i^{irr}$ and $V_2 = \bigoplus_j m_j V_j^{irr}$, the number of free parameters is:*

$$\sum_k n_k \cdot m_k$$

where the sum is over common irreducible types.

Exercises

Exercise 2.1 (Symmetry groups of molecules)**.**

1. Determine the symmetry group of the water molecule H_2O (group C_{2v}).
2. Enumerate all irreducible representations of C_{2v}.
3. Explain how these symmetries constrain a molecular energy prediction model.

Exercise 2.2 (Representations of S_3)**.**

1. Enumerate the six elements of the symmetric group S_3.
2. Find all irreducible representations of S_3 (there are three).
3. Verify the relation $\sum_i (\dim V_i)^2 = |G|$.
4. Decompose the standard permutation representation $\mathbb{R}^3$ into irreducibles.

Exercise 2.3 (Rodrigues' formula)**.**

1. Verify Rodrigues' formula for $\theta = \pi/2$ and $\hat{\mathbf{n}} = (0, 0, 1)$.
2. Show that $\exp(\theta[\hat{\mathbf{n}}]_\times) \in \mathrm{SO}(3)$.
3. Implement the exponential map $\mathfrak{so}(3) \to \mathrm{SO}(3)$ in PyTorch and verify orthogonality of the result.

Exercise 2.4 (Discrete group convolution)**.** Implement a group convolution on the dihedral group D_4 (symmetries of the square).

1. Enumerate the 8 elements of D_4.
2. Implement the multiplication table of D_4.
3. Implement the group convolution and verify its equivariance.

CHAPTER 2. GROUP THEORY FOR DEEP LEARNING

Chapter 3

Graph Neural Networks

Convolutional neural networks revolutionised computer vision, but they rest on an implicit assumption: data lives on a regular grid. What happens when the data is a social network, a molecule, a 3D mesh, or a transportation network? These structures are naturally represented by *graphs*, where the notion of neighbourhood is irregular and variable. The idea of *Graph Neural Networks* (GNNs), formalised by Franco Scarselli in 2009 and then rediscovered and extended by Thomas Kipf, Petar Veličković, and many others from 2016 onward, is to adapt the convolution mechanism to this irregular geometry: each node updates its representation by aggregating messages from its neighbours.

This *message passing* paradigm has become the unifying framework for learning on graphs. This chapter builds its foundations.

3.1 Graph Representation

Definition 3.1 (Attributed graph)**.** An **attributed graph** is a tuple $G = (V, E, \mathbf{X}, \mathbf{E})$ where:

- $V = \{1, \dots, n\}$ is the set of **nodes**.
- $E \subseteq V \times V$ is the set of **edges**.
- $\mathbf{X} \in \mathbb{R}^{n \times d}$ is the **node feature matrix**.
- $\mathbf{E} \in \mathbb{R}^{|E| \times d_e}$ contains the **edge features**.

Definition 3.2 (Associated matrices)**.** For a graph $G = (V, E)$:

- **Adjacency matrix**: $A_{ij} = 1$ if $(i, j) \in E$, 0 otherwise.
- **Degree matrix**: $D_{ii} = \sum_j A_{ij}$ (diagonal).
- **Laplacian**: $\mathbf{L} = \mathbf{D} - \mathbf{A}$.
- **Normalized Laplacian**: $\tilde{\mathbf{L}} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$.

Graph representation in PyTorch Geometric

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

```
# Graph with 4 nodes and 5 edges (undirected)
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)

# Node features (4 nodes, 3 features)
x = torch.tensor([
    [1.0, 0.0, 0.0],  # node 0: carbon
    [0.0, 1.0, 0.0],  # node 1: nitrogen
    [0.0, 0.0, 1.0],  # node 2: oxygen
    [1.0, 0.0, 0.0],  # node 3: carbon
], dtype=torch.float)

data = Data(x=x, edge_index=edge_index)
print(f"Nodes: {data.num_nodes}, Edges: {data.num_edges}")
print(f"Features: {data.x.shape}")
print(f"Isolated: {data.has_isolated_nodes()}")
print(f"Self-loops: {data.has_self_loops()}")
```

3.2 The Message Passing Paradigm

3.2.1 General Formalization

Definition 3.3 (Message Passing Neural Network (MPNN))**.** An **MPNN** (Gilmer et al., 2017) updates node representations through iterations of the form:

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

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

where:

- $\psi^{(\ell)}$ is the **message function**.
- $\bigoplus$ is a permutation-invariant **aggregation** operator (sum, mean, max).
- $\phi^{(\ell)}$ is the **update function**.
- $\mathbf{h}_v^{(0)} = \mathbf{x}_v$ (initial features).

Theorem 3.4 (Equivariance of MPNNs)**.** *Any MPNN layer with permutation-invariant aggregation is* ***permutation equivariant****: if $\pi \in S_n$ is a permutation, then:*

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

where $\pi \cdot G$ is the graph with nodes renumbered by π.

Proof. The aggregation $\bigoplus$ is permutation invariant by assumption. The functions ψ and ϕ are applied individually to each node. The neighborhood $\mathcal{N}(\pi(v))$ in $\pi \cdot G$ is exactly $\{\pi(u) : u \in \mathcal{N}(v)\}$ in G. Therefore:

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

3.2.2 Message Passing Illustration

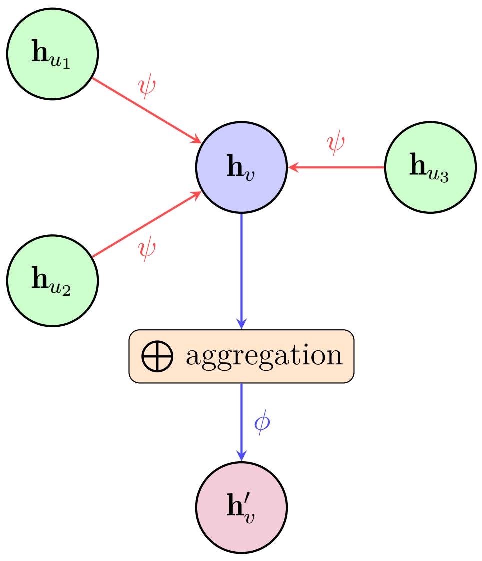

Figure 3.1: Message passing scheme: messages, aggregation, update.

3.3 Graph Convolutional Network (GCN)

Definition 3.5 (GCN layer — Kipf and Welling, 2017)**.** The GCN layer uses the following **propagation rule**:

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

where:

- $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ (self-loop addition).
- $\hat{\mathbf{D}}_{ii} = \sum_j \hat{A}_{ij}$ (adjusted degrees).
- $\mathbf{W}^{(\ell)} \in \mathbb{R}^{d_\ell \times d_{\ell+1}}$ is the weight matrix.
- σ is a nonlinear activation (ReLU).

Remark 3.6 (Interpretation)*.* For a node v, the GCN layer computes:

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

This is a **weighted average** of neighbor representations, followed by a linear transformation and nonlinearity.

Proposition 3.7 (Connection to the Laplacian)**.** The GCN layer can be written as:

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

where $\tilde{\mathbf{L}}_{\text{sym}}$ is the normalized Laplacian of $\hat{G}$ (graph with self-loops). The GCN layer thus performs **Laplacian smoothing** of features.

Complete GCN implementation

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

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

# Node classification on 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)

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()
```

```
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"Test accuracy: {acc:.4f}")
```

3.4 The Over-Smoothing Problem

Theorem 3.8 (Over-smoothing — Li et al., 2018)**.** *Stacking L GCN layers causes all node representations to converge to a fixed point:*

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

where $\boldsymbol{\pi}$ is the dominant eigenvector of the graph (related to the stationary distribution of the random walk). All nodes obtain the ***same representation****.*

Practical consequences

- Deep GNNs ($L > 5$) often lose performance.
- Representations become indistinguishable across nodes.
- Solutions: residual connections, normalization, DropEdge.

3.5 Readout and Graph-Level Prediction

Definition 3.9 (Readout)**.** To obtain a graph-level representation from node representations, a permutation-invariant **readout** function is used:

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

Common options:

- Sum: $\mathbf{h}_G = \sum_{v\in V} \mathbf{h}_v^{(L)}$
- Mean: $\mathbf{h}_G = \frac{1}{|V|}\sum_{v\in V} \mathbf{h}_v^{(L)}$
- Max: $\mathbf{h}_G = \max_{v\in V} \mathbf{h}_v^{(L)}$
- Attention: weighted sum with learned weights.

3.6 Expressive Power of GNNs

Theorem 3.10 (Connection to Weisfeiler–Leman test — Xu et al., 2019)**.** *MPNNs are* ***at most as expressive*** *as the 1-dimensional Weisfeiler–Leman (1-WL) test. More precisely:*

1. *If 1-WL distinguishes two graphs G_1 and G_2, then there exists an MPNN that distinguishes them.*
2. *If 1-WL does not distinguish G_1 and G_2, then* ***no*** *MPNN can distinguish them.*

Definition 3.11 (Weisfeiler–Leman test (1-WL))**.** The 1-WL test is an iterative color-refinement algorithm:

1. **Initialization**: $c_v^{(0)}$ = initial color of v.
2. **Iteration**: $c_v^{(t+1)} = \text{HASH}\Big(c_v^{(t)}, \{\!\{c_u^{(t)} : u \in \mathcal{N}(v)\}\!\}\Big)$
3. **Termination**: when colors stabilize.

where $\{\!\{\cdot\}\!\}$ denotes a multiset.

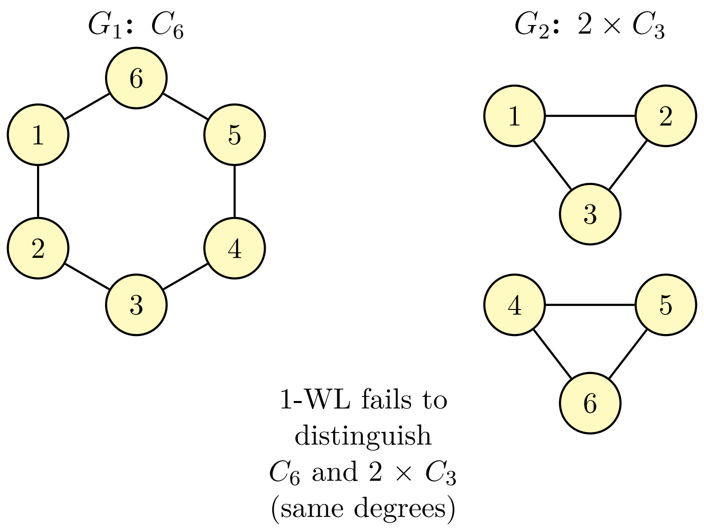
G_1: C_6
G_2: $2 \times C_3$
1-WL fails to distinguish C_6 and $2 \times C_3$ (same degrees)

Figure 3.2: Two regular graphs that 1-WL (and thus MPNNs) cannot distinguish.

3.7 GCN from Scratch

GCN implemented from scratch

```
import torch
import torch.nn as nn

class GCNLayerManual(nn.Module):
    """Manually implemented GCN layer."""
    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):
        n = A.size(0)
        A_hat = A + torch.eye(n, device=A.device)
        D_hat = torch.diag(A_hat.sum(dim=1))
        D_inv_sqrt = torch.diag(1.0 / torch.sqrt(A_hat.sum(dim=1)))
        A_norm = D_inv_sqrt @ A_hat @ D_inv_sqrt
        H = A_norm @ X @ self.W + self.bias
        return torch.relu(H)

# Test on a small graph
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"Output: {out.shape}")  # (4, 16)
```

3.8 Types of Graph Tasks

Definition 3.12 (Prediction levels)**.** Graph learning tasks fall into three levels:

1. **Node-level**: node classification, regression (e.g., document classification in a citation network).
2. **Edge-level**: link prediction, edge classification (e.g., recommendation).
3. **Graph-level**: classification, regression of entire graphs (e.g., molecular property prediction).

Molecular graph classification

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

CHAPTER 3. GRAPH NEURAL NETWORKS

```
        )

    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))
        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()
```

Exercises

Exercise 3.1 (Deriving the GCN layer).

1. Start from the spectral convolution $g_\theta * x = U g_\theta(\Lambda) U^\top x$ and show that the first-order Chebyshev polynomial approximation yields the GCN layer.

2. Explain the role of the renormalization $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$.

Exercise 3.2 (Receptive field). Show that after L MPNN layers, the representation of node v depends on all nodes at distance $\leq L$ in the graph. Deduce the **receptive field** of a GNN.

Exercise 3.3 (Expressivity limitations). Construct two non-isomorphic graphs on 8 nodes that the 1-WL test cannot distinguish. Verify experimentally that a GCN produces the same representations for both graphs.

Exercise 3.4 (Message passing with edge features). Extend the from-scratch GCN implementation to support edge features $\mathbf{e}_{uv} \in \mathbb{R}^{d_e}$ in the message function.

Chapter 4

GNN Variants (GAT, GIN, GraphSAGE)

The GCN of Kipf and Welling (2017) opened the way, but its limitations quickly became apparent: isotropic aggregation, expressivity capped at the 1-Weisfeiler-Leman test, and inability to handle large graphs inductively. In response, a wave of alternative architectures emerged: *Graph Attention Networks* (GAT) introduce attention mechanisms to weight neighbours; *Graph Isomorphism Networks* (GIN) maximise theoretical expressivity; *GraphSAGE* enables inductive learning through neighbourhood sampling. This chapter explores these variants and the trade-offs they embody.

4.1 Beyond GCN: Motivations

The GCN of Kipf and Welling has several limitations:

- **Isotropic** aggregation: all neighbors are treated equally (weighted only by degree).
- **Limited expressivity**: strict equivalence to 1-WL.
- **Scalability**: requires the full adjacency matrix.

This chapter presents three major variants addressing these limitations.

4.2 Graph Attention Networks (GAT)

4.2.1 Attention on Graphs

Definition 4.1 (GAT layer — Veličković et al., 2018)**.** The GAT layer uses an **attention** mechanism to weight neighbor contributions:

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

where the attention coefficients α_{vu} are computed by:

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

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

where $\mathbf{a} \in \mathbb{R}^{2d'}$ is the learned attention vector and $\|$ denotes concatenation.

Remark 4.2 (Multi-head attention)*.* To stabilize training, GAT uses K independent attention heads:

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

The last layer typically uses averaging instead of concatenation.

Why attention?

Attention allows the model to learn **which neighbors matter** for each node, unlike GCN where weighting is fixed by graph structure. This is especially useful when:

- Neighbors have **varying relevance**.
- The graph contains **noise** (uninformative edges).
- Relationships are **asymmetric** (directed graphs).

GAT implementation with 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 test accuracy: {acc:.4f}")
```

4.2.2 GATv2: Dynamic Attention

Definition 4.3 (GATv2 — Brody et al., 2022)**.** GATv2 fixes a limitation of GAT where attention is **static** (the ranking of neighbors does not depend on the query node). GATv2 uses:

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

The nonlinearity is applied **before** the dot product with **a**, making the attention truly **dynamic**.

4.3 Graph Isomorphism Network (GIN)

4.3.1 Maximizing Expressive Power

Theorem 4.4 (GIN — Xu et al., 2019)**.** *For an MPNN to achieve the maximum power of the 1-WL test, the following conditions are necessary and sufficient:*

1. *The aggregation must be* ***injective*** *on multisets.*
2. *The update function must be* ***injective****.*

Definition 4.5 (GIN layer)**.** The GIN layer computes:

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

where ϵ is a learned or fixed parameter.

Proposition 4.6 (Injectivity of sum)**.** Among classical aggregations (sum, mean, max), only **sum** is injective on multisets:

- $\max\{1,1,2\} = \max\{1,2\} = 2$ (information loss).
- $\text{mean}\{1,1\} = \text{mean}\{1\} = 1$ (cardinality loss).
- $\text{sum}\{1,1,2\} = 4 \neq \text{sum}\{1,2\} = 3$ (injective).

CHAPTER 4. GNN VARIANTS (GAT, GIN, GRAPHSAGE)

GIN implementation

```
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"Parameters: {sum(p.numel() for p in model.parameters()):,}")
```

4.4 GraphSAGE: Inductive Learning

4.4.1 Motivation: Scalability

Definition 4.7 (GraphSAGE — Hamilton et al., 2017)**.** GraphSAGE (*SAmple and aggreGatE*) uses **neighborhood sampling** for scalability:

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

where $\mathcal{S}(v) \subseteq \mathcal{N}(v)$ is a sampled subset of neighbors (typically $|\mathcal{S}(v)| \leq k$).

Proposition 4.8 (Advantages of GraphSAGE)**.** 1. **Inductive**: can generalize to unseen nodes.

2. **Scalable**: complexity controlled by sample count.
3. **Mini-batch**: compatible with mini-batch training.

Subgraph sampling for GraphSAGE

1. For each target node v, sample k_1 direct neighbors.
2. For each of those neighbors, sample k_2 neighbors (2-hop).
3. Form the induced subgraph and propagate messages.
4. Per-batch complexity is $\mathcal{O}(B \cdot k_1 \cdot k_2)$ instead of $\mathcal{O}(n)$.

GraphSAGE with neighborhood sampling

```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

train_loader = NeighborLoader(
    data,
    num_neighbors=[25, 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 Variant Comparison

Property	GCN	GAT	GIN	SAGE
Attention	No	Yes	No	No
1-WL expressivity	$<$	$<$	$=$	$<$
Inductive	No	Yes	Yes	Yes
Scalable (mini-batch)	Hard	Moderate	Moderate	Yes
Edge features	No	Yes (v2)	No	No

4.6 Advanced Architectures

4.6.1 PNA — Principal Neighbourhood Aggregation

Definition 4.9 (PNA — Corso et al., 2020). PNA combines **multiple aggregators** and **scalers**:

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

The concatenation of $4 \times 3 = 12$ channels captures different aspects of the neighborhood distribution.

4.6.2 Residual Connections for Deep GNNs

Definition 4.10 (GCNII — Chen et al., 2020). GCNII adds an **initial residual connection** and **identity mapping** to avoid over-smoothing:

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

where $\hat{\mathbf{P}} = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}$ and α_ℓ, β_ℓ are hyperparameters.

Exercises

Exercise 4.1 (Experimental comparison). Compare GCN, GAT, GIN, and GraphSAGE on the Cora dataset:

1. Implement all four models with the same parameter count.
2. Train for 200 epochs with the same hyperparameters.
3. Report test accuracy and loss curves.

4. Analyze performance differences.

Exercise 4.2 (GAT attention visualization).

1. Train a GAT on Cora and extract attention coefficients α_{vu}.
2. Visualize attention weights on a subgraph of 20 nodes.
3. Are attention coefficients correlated with node labels?

Exercise 4.3 (GIN expressive power).

1. Construct two graphs that GCN cannot distinguish but GIN can.
2. Verify experimentally.
3. Construct two graphs that even GIN cannot distinguish (1-WL failure).

Exercise 4.4 (Mini-batch training). Implement mini-batch training with neighborhood sampling for a GCN (without using `NeighborLoader`).

1. Implement the sampling strategy.
2. Compare training speed with full-batch mode.
3. Evaluate the impact of the number of sampled neighbors on accuracy.

CHAPTER 4. GNN VARIANTS (GAT, GIN, GRAPHSAGE)

Chapter 5

Spectral Graph Learning

The Fourier transform is one of the most powerful tools in analysis—but it assumes that data lives on a regular grid. How does one define a Fourier transform on a graph? The answer passes through the *graph Laplacian*, an operator whose eigenvectors play the role of sinusoids. Low frequencies correspond to smooth signals (similar neighbours), high frequencies to oscillating signals. This spectral analogy, developed by Fan Chung in the 1990s and applied to deep learning by Bruna et al. (2014), provides the mathematical foundations for spectral convolutional networks on graphs.

5.1 Spectral Graph Theory

5.1.1 The Graph Laplacian

Definition 5.1 (Graph Laplacian)**.** Let $G = (V, E)$ be an undirected graph with n nodes. The **combinatorial Laplacian** is:

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

The **normalized Laplacian** is:

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

Theorem 5.2 (Laplacian properties)**.** *The Laplacian* $\mathbf{L}$ *satisfies:*

1. $\mathbf{L}$ *is* ***positive semi-definite****:* $\mathbf{x}^\top\mathbf{L}\mathbf{x} \geq 0$ *for all* $\mathbf{x} \in \mathbb{R}^n$*.*
2. ***Quadratic form****:* $\mathbf{x}^\top\mathbf{L}\mathbf{x} = \sum_{(i,j)\in E}(x_i - x_j)^2$*.*
3. $\mathbf{L}$ *has* ***eigenvalues*** $0 = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$*.*
4. *The* ***multiplicity*** *of* $\lambda = 0$ *equals the number of connected components.*

Proof. For the quadratic form:

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

□

CHAPTER 5. SPECTRAL GRAPH LEARNING

The Laplacian as a smoothing operator

The quadratic form $\mathbf{x}^\top \mathbf{L}\mathbf{x}$ measures the **total variation** of the signal $\mathbf{x}$ on the graph. A signal $\mathbf{x}$ is smooth when neighboring nodes have similar values, corresponding to a small value of $\mathbf{x}^\top \mathbf{L}\mathbf{x}$.

5.1.2 Spectral Decomposition

Definition 5.3 (Graph spectrum)**.** The **spectrum** of the graph is the set of eigenvalues of the Laplacian:

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

where $\mathbf{U} = [\mathbf{u}_1, \dots, \mathbf{u}_n]$ is the matrix of eigenvectors (the **graph Fourier basis**) and $\boldsymbol{\Lambda} = \mathrm{diag}(\lambda_1, \dots, \lambda_n)$.

Graph Fourier Transform

The **Fourier transform** of a signal $\mathbf{x} \in \mathbb{R}^n$ on the graph is:

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

The **inverse transform** is:

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

The coefficient $\hat{x}_k = \mathbf{u}_k^\top \mathbf{x}$ measures the projection of the signal onto the k-th graph Fourier mode.

Spectral decomposition and Fourier transform

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

# Create a graph
G = nx.karate_club_graph()
n = G.number_of_nodes()

# Normalized Laplacian
L = nx.normalized_laplacian_matrix(G).toarray()
L = torch.tensor(L, dtype=torch.float)

# Spectral decomposition
eigenvalues, eigenvectors = torch.linalg.eigh(L)
print(f"Smallest eigenvalues: {eigenvalues[:5]}")
print(f"Largest eigenvalue: {eigenvalues[-1]:.4f}")

# Fourier transform of a signal
x = torch.randn(n)
x_hat = eigenvectors.T @ x  # spectral domain
x_reconstructed = eigenvectors @ x_hat
print(f"Reconstruction error: {(x - x_reconstructed).norm():.2e}")
```

5.2 Spectral Convolution on Graphs

5.2.1 Definition

Definition 5.4 (Spectral convolution)**.** By analogy with the classical convolution theorem (convolution in the spatial domain is multiplication in the frequency domain), the **spectral convolution** on a graph is defined as:

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

where $\odot$ denotes element-wise multiplication.

Definition 5.5 (Parameterized spectral filter)**.** A **spectral filter** is defined by a function $g_\theta : \mathbb{R} \to \mathbb{R}$ applied to the eigenvalues:

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

where θ are the learned filter parameters.

Computational cost

Naive spectral convolution has three problems:

1. The diagonalization $\mathbf{L} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^\top$ costs $\mathcal{O}(n^3)$.
2. The number of parameters is n (one per eigenvalue).
3. The filter is **not localized** in the spatial domain.

5.2.2 ChebNet: Chebyshev Polynomials

Definition 5.6 (Chebyshev filter — Defferrard et al., 2016)**.** To address the above issues, g_θ is approximated by a Chebyshev polynomial of order K:

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

where $\tilde{\mathbf{L}} = \frac{2}{\lambda_{\max}}\mathbf{L} - \mathbf{I}$ (rescaling to $[-1, 1]$) and T_k is the k-th Chebyshev polynomial:

$$T_0(x) = 1, \quad T_1(x) = x \tag{5.1}$$

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

Theorem 5.7 (Chebyshev filter properties)**.** *1.* ***Localization****: a filter of order K is localized to a K-hop neighborhood.*

2. ***Complexity****: $\mathcal{O}(K \cdot |E|)$ instead of $\mathcal{O}(n^3)$—linear in the number of edges.*

3. ***Parameters****: $K + 1$ instead of n.*

4. ***No spectral decomposition*** *required.*

CHAPTER 5. SPECTRAL GRAPH LEARNING

ChebNet implementation

```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"ChebNet parameters: {sum(p.numel() for p in
↪  model.parameters()):,}")
```

5.2.3 From ChebNet to GCN

Proposition 5.8 (GCN as a special case)**.** The GCN layer of Kipf and Welling is obtained by setting $K = 1$ in the Chebyshev filter and $\lambda_{\max} \approx 2$:

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

Setting $\theta_0 = -\theta_1 = \theta$ and adding renormalization:

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

5.3 Advanced Spectral Analysis

5.3.1 Spectral Cuts and Fiedler

Definition 5.9 (Fiedler value)**.** The **Fiedler value** λ_2 (second smallest eigenvalue of the Laplacian) measures the **algebraic connectivity** of the graph. The associated eigenvector $\mathbf{u}_2$ (Fiedler vector) defines a bipartition of the graph.

Theorem 5.10 (Cheeger inequality)**.** *For a graph G, the isoperimetric constant $h(G)$ satisfies:*

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

where $h(G) = \min_{S \subset V} \frac{|\partial S|}{\min(\mathrm{vol}(S), \mathrm{vol}(\bar{S}))}$ and $|\partial S|$ is the number of edges cutting S and $\bar{S}$.

5.3.2 Spectral Positional Encodings

Definition 5.11 (Laplacian Positional Encoding (LPE))**. Laplacian positional encodings** use the first k eigenvectors of the Laplacian as additional node features:

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

Sign ambiguity

Eigenvectors are defined up to a sign: if $\mathbf{u}$ is an eigenvector, so is $-\mathbf{u}$. To resolve this ambiguity:

- **SignNet** (Lim et al., 2022): $\phi(\mathbf{u}) + \phi(-\mathbf{u})$.
- **BasisNet**: treat the set of eigenvectors as a subspace basis.

Spectral positional encodings

```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"Original features: {data.x.shape}")
print(f"Positional encodings: {data.laplacian_eigenvector_pe.shape}")

x_augmented = torch.cat([data.x, data.laplacian_eigenvector_pe], dim=1)
print(f"Augmented features: {x_augmented.shape}")
```

5.4 Graph Transformers

Definition 5.12 (Graph Transformer)**.** A **Graph Transformer** applies attention over all nodes (not just neighbors), breaking graph locality. Spectral positional encodings provide structural information:

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

where $\mathbf{q}_v, \mathbf{k}_u, \mathbf{v}_u$ are query, key, value projections.

Remark 5.13 (Graphormer — Ying et al., 2021)*.* Graphormer encodes graph structure via:

1. **Centrality encoding**: $\mathbf{h}_v^{(0)} = \mathbf{x}_v + \mathbf{z}_{d_v^+} + \mathbf{z}_{d_v^-}$.
2. **Spatial encoding**: $b_{vu} = g(\text{SPD}(v, u))$ (shortest path distance).
3. **Edge encoding**: encoding edge features along the path.

5.5 Wavelets on Graphs

Definition 5.14 (Spectral wavelets)**. Graph wavelets** (Hammond et al., 2011) use a scaling function g in the spectral domain:

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

where $s > 0$ is the scale parameter. Wavelets at different scales capture structures at different resolutions.

Exercises

Exercise 5.1 (Spectrum of the complete graph and cycle)**.** 1. Analytically compute the Laplacian spectrum of the complete graph K_n.

2. Compute the spectrum of the cycle C_n.
3. Implement and verify numerically.
4. Visualize the first eigenvectors (Fourier modes).

Exercise 5.2 (Spectral filtering)**.** 1. Implement a spectral low-pass filter $g(\lambda) = e^{-\alpha\lambda}$.

2. Apply it to a noisy signal on the karate club graph.
3. Compare with GCN propagation-based smoothing.

Exercise 5.3 (Spectral partitioning)**.** Implement spectral graph partitioning using the Fiedler vector.

1. Generate a graph with two clear communities.
2. Compute the Fiedler vector.
3. Partition according to the sign of the components.
4. Compare with the Louvain algorithm.

Exercise 5.4 (ChebNet vs. GCN)**.** Experimentally compare ChebNet (with $K = 2, 3, 5, 10$) and GCN on Cora.

1. Plot accuracy as a function of K.
2. Analyze the locality/expressivity trade-off.
3. Measure training time.

Chapter 6

Learning on Manifolds

Graphs are discrete structures, but much data lives on *continuous* non-Euclidean spaces: the surface of the Earth, the space of 3D rotations, the manifold of covariance matrices. Can one define neural networks directly on these *differentiable manifolds*? The geometric approach to deep learning, systematised by Bronstein et al. in their 2021 "blueprint," shows that this is possible—provided that Euclidean operations (convolution, translation) are replaced by their Riemannian analogues (parallel transport, exponential map). This chapter lays the necessary geometric foundations.

6.1 Differential Geometry: Review

6.1.1 Differentiable Manifolds

Definition 6.1 (Differentiable manifold)**.** A **differentiable manifold** of dimension d is a topological space $\mathcal{M}$ that is locally homeomorphic to $\mathbb{R}^d$. More precisely, for each point $p \in \mathcal{M}$, there exists an open neighborhood U and a homeomorphism (local chart) $\varphi : U \to \mathbb{R}^d$. The transition maps $\varphi_\beta \circ \varphi_\alpha^{-1}$ are C^∞.

Example 6.2 (Common manifolds)**.** 1. The **sphere** $S^2 = \{(x, y, z) \in \mathbb{R}^3 : x^2 + y^2 + z^2 = 1\}$ — dimension 2.

2. The **torus** $T^2 = S^1 \times S^1$ — dimension 2.

3. The group SO(3) — dimension 3.

4. The space of symmetric positive definite matrices $\mathrm{Sym}^+(n)$.

6.1.2 Tangent Space and Riemannian Metric

Definition 6.3 (Tangent space)**.** The **tangent space** $T_p\mathcal{M}$ at a point $p \in \mathcal{M}$ is the vector space of tangent vectors to $\mathcal{M}$ at p. It has dimension $d = \dim(\mathcal{M})$.

Definition 6.4 (Riemannian metric)**.** A **Riemannian metric** on $\mathcal{M}$ assigns to each $p \in \mathcal{M}$ an inner product $g_p : T_p\mathcal{M} \times T_p\mathcal{M} \to \mathbb{R}$ varying smoothly with p. The pair $(\mathcal{M}, g)$ is a **Riemannian manifold**.

CHAPTER 6. LEARNING ON MANIFOLDS

Basic objects in Riemannian geometry

- **Curve length** $\gamma : [0,1] \to \mathcal{M}$: $L(\gamma) = \int_0^1 \sqrt{g_{\gamma(t)}(\dot{\gamma}(t), \dot{\gamma}(t))}\, dt$
- **Geodesic distance**: $d(p,q) = \inf_\gamma L(\gamma)$ over curves connecting p and q.
- **Exponential map**: $\exp_p : T_p\mathcal{M} \to \mathcal{M}$, sends v to the point reached by the geodesic starting at p in direction v.
- **Logarithmic map**: $\log_p : \mathcal{M} \to T_p\mathcal{M}$, (local) inverse of $\exp_p$.

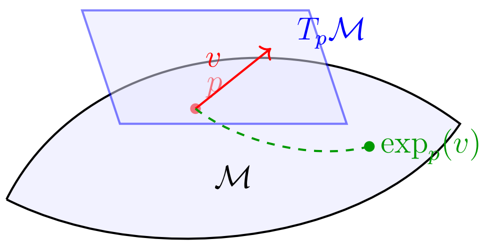

Figure 6.1: Tangent space, tangent vector, and exponential map.

6.2 Differential Operators on Manifolds

Definition 6.5 (Laplace–Beltrami operator)**.** The **Laplace–Beltrami operator** generalizes the Laplacian to a Riemannian manifold:

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

where g^{ij} are the components of the inverse metric tensor and $|g| = \det(g_{ij})$.

Theorem 6.6 (Properties of $\Delta_{\mathcal{M}}$)**.** *1. $\Delta_{\mathcal{M}}$ is **self-adjoint** and **negative semi-definite** (with the convention $\Delta f = -\operatorname{div}(\nabla f)$ the sign is reversed).*

*2. It is **intrinsic**: independent of the coordinate system.*

3. It generalizes the graph Laplacian: on a graph, $\mathbf{L}$ is a discretization of $-\Delta_{\mathcal{M}}$.

4. It admits a spectral decomposition: $\Delta_{\mathcal{M}}\phi_k = -\lambda_k \phi_k$ with $0 = \lambda_0 \le \lambda_1 \le \cdots$

Proposition 6.7 (Heat equation on $\mathcal{M}$)**.** The heat equation on a manifold reads:

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

Its solution is $u(p,t) = \int_{\mathcal{M}} K_t(p,q)\, u_0(q)\, dq$ where $K_t(p,q) = \sum_k e^{-\lambda_k t} \phi_k(p) \phi_k(q)$ is the **heat kernel**.

6.3 Convolutions on Manifolds

6.3.1 Challenges

On a manifold, classical convolution poses several challenges:

- No **translation group structure** (except for $\mathbb{R}^n$ and the torus).
- No **global coordinate system**.
- **Parallel transport** is needed to compare vectors at different points.

6.3.2 Spectral Approach: MoNet and Variants

Definition 6.8 (MoNet — Monti et al., 2017)**.** MoNet (*Mixture Model Networks*) defines parametric filters in local pseudo-polar coordinates:

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

where $\mathbf{c}(v,u) = (\rho_{vu}, \theta_{vu})$ are pseudo-polar coordinates of u in the local frame of v, and $w_j(\mathbf{c}) = \exp\big(-\frac{1}{2}(\mathbf{c}-\boldsymbol{\mu}_j)^\top \boldsymbol{\Sigma}_j^{-1}(\mathbf{c}-\boldsymbol{\mu}_j)\big)$ are learned Gaussian kernels.

6.3.3 Convolution via Spectral Descriptors

Definition 6.9 (Diffusion kernel)**.** The **diffusion kernel** of a manifold is defined by:

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

It defines an inner product in a reproducing kernel Hilbert space (RKHS) and provides intrinsic shape descriptors.

Definition 6.10 (HKS — Heat Kernel Signature)**.** The **Heat Kernel Signature** is an intrinsic descriptor of local geometry:

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

It encodes multi-scale information: small t captures local geometry, large t captures global geometry.

Computing HKS on a mesh

```python
import torch
import numpy as np

def compute_hks(eigenvalues, eigenvectors, time_scales):
    """Compute Heat Kernel Signature.
    Args:
        eigenvalues: (K,) Laplacian eigenvalues
        eigenvectors: (N, K) eigenvectors
        time_scales: (T,) time scales
    Returns:
        hks: (N, T) HKS descriptors
    """
    exp_vals = torch.exp(
        -eigenvalues.unsqueeze(1) * time_scales.unsqueeze(0)
    )
    phi_sq = eigenvectors ** 2
    hks = phi_sq @ exp_vals  # (N, T)
    return hks

eigenvalues = torch.linspace(0, 10, 100)
eigenvectors = torch.randn(500, 100)
eigenvectors, _ = torch.linalg.qr(eigenvectors)
time_scales = torch.logspace(-2, 2, 16)

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

6.4 Convolution via Parallel Transport

Definition 6.11 (Parallel transport)**. Parallel transport** along a curve $\gamma : [0,1] \to \mathcal{M}$ is a linear map $\Gamma_\gamma : T_{\gamma(0)}\mathcal{M} \to T_{\gamma(1)}\mathcal{M}$ that transports a tangent vector along γ while preserving the inner product and satisfying:

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

where ∇ is the Levi-Civita connection.

Definition 6.12 (Gauge Equivariant CNN — de Haan et al., 2021)**.** A **Gauge Equivariant CNN** defines convolutions on a manifold using **local frames** (gauges) and parallel transport:

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

The network is equivariant to gauge changes: the choice of local frame does not affect the output.

6.5 Learning on Specific Manifolds

6.5.1 Convolutions on the Sphere

Definition 6.13 (Spherical CNN — Cohen et al., 2018)**. Spherical CNNs** define convolutions on S^2 using spherical harmonics as a spectral basis:

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

where $\hat{f}_\ell^m$ and $\hat{\psi}_\ell$ are coefficients in the spherical harmonics basis. For a **zonal** filter (rotationally symmetric), $\hat{\psi}_\ell$ depends only on ℓ.

Theorem 6.14 (SO(3) equivariance)**.** *Spherical convolution is rotation-equivariant: if* $R \in \mathrm{SO}(3)$ *and* $L_R f(\hat{\mathbf{r}}) = f(R^{-1}\hat{\mathbf{r}})$*, then:*

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

6.5.2 Learning on Lie Groups

Definition 6.15 (LieConv — Finzi et al., 2020)**.** LieConv defines convolutions directly on Lie groups using a **continuous kernel** parameterized in the Lie algebra:

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

In practice, the integral is approximated by Monte Carlo on a sampled neighborhood.

6.6 Applications: 3D Shape Analysis

Definition 6.16 (Shape correspondence)**.** The **shape correspondence** problem seeks a map $T : \mathcal{M}_1 \to \mathcal{M}_2$ between two surfaces that preserves geometric structure. Spectral methods use **functional maps**:

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

where $\mathbf{\Phi}_i$ are the truncated eigenfunction matrices and T_f is the functional correspondence operator.

Functional maps for shape correspondence

```python
import torch

def functional_map(phi_1, phi_2, descriptors_1, descriptors_2, k=30):
    """Compute a functional map.

    Args:
        phi_1, phi_2: (N, k) truncated eigenfunctions
        descriptors_1, descriptors_2: (N, d) descriptors (HKS, etc.)
        k: number of eigenfunctions
    Returns:
        C: (k, k) functional map matrix
    """
    A1 = phi_1[:, :k].T @ descriptors_1  # (k, d)
    A2 = phi_2[:, :k].T @ descriptors_2  # (k, d)
    C, _ = torch.linalg.lstsq(A1.T, A2.T)
    C = C.T  # (k, k)
    return C

N = 1000
k = 30
phi_1 = torch.randn(N, k)
```

```python
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"Functional map: {C.shape}")  # (30, 30)
```

6.7 Optimization on Manifolds

Definition 6.17 (Riemannian gradient)**.** The **Riemannian gradient** of a function $f : \mathcal{M} \to \mathbb{R}$ at p is the unique vector $\operatorname{grad} f(p) \in T_p\mathcal{M}$ such that:

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

Riemannian gradient descent

1. Compute the Euclidean gradient $\nabla f(\mathbf{x})$.
2. Project onto the tangent space: $\xi = \operatorname{Proj}_{T_\mathbf{x}\mathcal{M}}(\nabla f)$.
3. Perform a retraction step: $\mathbf{x}' = \operatorname{Retr}_\mathbf{x}(-\eta\,\xi)$.

For $\mathrm{SO}(n)$, the retraction is $\operatorname{Retr}_Q(\xi) = \operatorname{qr}(Q + \xi)$.

Optimization on the sphere with geoopt

```python
import torch
import geoopt

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

# Objective: minimize distance to a target point
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"Final distance: {(x - target).norm():.6f}")
print(f"On sphere: {x.norm():.6f}")  # should be ~1
```

Exercises

Exercise 6.1 (Laplace–Beltrami on the sphere)**.**

1. Show that spherical harmonics Y_ℓ^m are eigenfunctions of Δ_{S^2} with eigenvalues $\lambda_\ell = \ell(\ell+1)$.
2. Compute the first spherical harmonics and visualize them.

Exercise 6.2 (Heat kernel)**.**

1. Analytically compute the heat kernel on the circle S^1.
2. Implement numerical computation of the heat kernel on a triangular mesh.
3. Visualize heat diffusion at different scales t.

Exercise 6.3 (Parallel transport)**.**

1. Compute parallel transport along a meridian of the sphere.
2. Show that parallel transport around a spherical triangle induces a rotation by an angle equal to the spherical excess.
3. Implement discrete parallel transport on a mesh.

Exercise 6.4 (Shape correspondence)**.** Implement a shape correspondence pipeline using:

1. HKS descriptors as node features.
2. A GNN to learn equivariant descriptors.
3. Functional maps for correspondence.

CHAPTER 6. LEARNING ON MANIFOLDS

Chapter 7

Equivariant and Invariant Networks

Convolutional neural networks owe their success to a precise mathematical property: translation equivariance. A cat remains a cat, whether it sits on the left or right of the image. But what happens when the relevant symmetries are not translations? In molecular chemistry, a molecule's properties are invariant under rotation and translation of all its atoms. In particle physics, interactions obey gauge symmetries. Taco Cohen and Max Welling, in 2016, pioneered *equivariant networks* by showing how to build neural network layers that respect the symmetries of a given group. This approach, formalized by the geometric framework of Bronstein et al., unifies CNNs, GNNs, and transformers as special cases of a single principle: the network's structure must reflect the problem's symmetries.

7.1 General Formalism

7.1.1 Equivariance Review

Definition 7.1 (General equivariance)**.** Let G be a group acting on spaces $\mathcal{X}$ and $\mathcal{Y}$ via representations $\rho_{\mathcal{X}}$ and $\rho_{\mathcal{Y}}$. A function $f : \mathcal{X} \to \mathcal{Y}$ is **G-equivariant** if:

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

Proposition 7.2 (Composition of equivariant maps)**.** If $f : \mathcal{X} \to \mathcal{Y}$ and $g : \mathcal{Y} \to \mathcal{Z}$ are equivariant, then $g \circ f : \mathcal{X} \to \mathcal{Z}$ is equivariant. This justifies stacking equivariant layers in a deep network.

7.1.2 Constructing Equivariant Layers

Theorem 7.3 (Characterization of equivariant linear layers)**.** *The space of G-equivariant linear maps $T : V \to W$ is:*

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

Its dimension is given by the inner product of characters: $\dim \mathrm{Hom}_G(V, W) = \langle \chi_V, \chi_W \rangle_G$.

CHAPTER 7. EQUIVARIANT AND INVARIANT NETWORKS

Equivariance constraint = weight sharing

Imposing equivariance amounts to **constraining the weight matrix** of the linear layer. The larger the symmetry group, the stronger the constraint, and the fewer free parameters.

Group	Constraint	Parameters
Trivial $\{e\}$	None	n^2
Translation $\mathbb{Z}^d$	Toeplitz (convolution)	k^d
Permutation S_n	Doubly stochastic	2
SO(3)	Clebsch–Gordan	$\sum_k n_k m_k$

7.2 SE(3)-Equivariant Networks

7.2.1 Motivation: Physics and Chemistry

Many physical problems possess SE(3) symmetry (rotations + translations):

- Molecular energy prediction.
- Protein dynamics.
- Fluid simulation.

7.2.2 Type-ℓ Representations

Definition 7.4 (Type-ℓ tensor fields)**.** A **type-ℓ tensor field** is a function $f : \mathbb{R}^3 \to \mathbb{R}^{2\ell+1}$ that transforms according to the irreducible representation D^ℓ under rotation:

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

- $\ell = 0$: scalars (1D)—energy, charge.
- $\ell = 1$: vectors (3D)—forces, velocities.
- $\ell = 2$: order-2 traceless tensors (5D)—stresses.

7.2.3 Tensor Field Networks (TFN)

Definition 7.5 (TFN — Thomas et al., 2018)**. Tensor Field Networks** define SE(3)-equivariant layers using tensor products of irreducible representations. For a node v at position $\mathbf{r}_v$:

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

where:

- $\hat{\mathbf{r}}_{vu} = (\mathbf{r}_v - \mathbf{r}_u)/r_{vu}$ is the normalized direction.
- $Y^{\ell_2}(\hat{\mathbf{r}})$ are spherical harmonics.

- $\otimes_{\mathrm{CG}}$ is the tensor product with Clebsch–Gordan coefficients.
- $W^{\ell_1 \ell_2 \ell}(r)$ is a radial kernel (scalar, hence invariant).

Theorem 7.6 (TFN equivariance)**.** *TFN layers are* SE(3)*-equivariant: for* $R \in \mathrm{SO}(3)$ *and* $\mathbf{t} \in \mathbb{R}^3$*:*

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

The proof relies on the transformation property of spherical harmonics and the Clebsch–Gordan decomposition.

Equivariant network with e3nn

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

model = Network(
    irreps_in="5x0e",
    irreps_hidden="32x0e + 8x1o + 4x2e",
    irreps_out="1x0e",  # scalar output (energy)
    irreps_node_attr="1x0e",
    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,
)

positions = torch.randn(20, 3)
node_features = torch.randn(20, 5)
node_attrs = torch.ones(20, 1)

# Verify equivariance
R = o3.rand_matrix()
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"Scalar invariance: "
      f"{torch.allclose(out_original, out_rotated, atol=1e-4)}")
```

7.3 EGNN: Simplified Equivariance

Definition 7.7 (EGNN — Satorras et al., 2021)**. E(n) Equivariant Graph Neural Networks** (EGNN) achieve E(n) equivariance **without** irreducible representations:

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

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

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

where $d_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|$ and ϕ_e, ϕ_h, ϕ_x are MLPs.

Proposition 7.8 (EGNN equivariance)**.** 1. **Scalar features $\mathbf{h}_i$** are **invariant** under E(n).

2. **Coordinates $\mathbf{x}_i$** are **equivariant** under E(n).

3. The proof relies on d_{ij}^2 and $\mathbf{x}_i - \mathbf{x}_j$ transforming correctly.

EGNN implementation

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

class EGNNLayer(nn.Module):
    """E(n) equivariant layer."""
    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
        diff = x[row] - x[col]
        dist_sq = (diff ** 2).sum(dim=-1, keepdim=True)
        m = self.phi_e(torch.cat([h[row], h[col], dist_sq], dim=-1))
        agg = torch.zeros_like(h)
        agg.index_add_(0, row, m)
        h_new = h + self.phi_h(torch.cat([h, agg], dim=-1))
        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

# Equivariance test
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)

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 Invariant Networks

Definition 7.9 (Fundamental invariants)**.** To build an E(n)-invariant network, one can use **fundamental invariants**:

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 between planes defined by (i, j, k) and (j, k, l).

Theorem 7.10 (Completeness of invariants)**.** *Interatomic distances $\{d_{ij}\}$ determine the geometry up to reflection. Adding dihedral angles resolves the reflection ambiguity. Every* SE(3) *invariant can be expressed as a function of distances and angles.*

Definition 7.11 (SchNet — Schütt et al., 2018)**.** SchNet uses interatomic distances as the sole geometric information:

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

where $\mathrm{RBF}(d) = [\exp(-\gamma_1(d-\mu_1)^2), \ldots, \exp(-\gamma_K(d-\mu_K)^2)]$ is a radial basis function expansion.

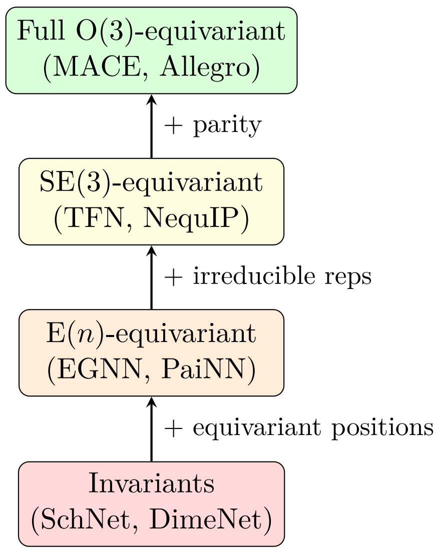

Figure 7.1: Hierarchy of equivariant architectures.

7.5 Hierarchy of Equivariant Models

Exercises

Exercise 7.1 (Equivariance verification)**.** For each of the following models, numerically verify equivariance by applying random rotations, translations, and reflections:

1. SchNet (E(3) invariance).
2. EGNN (E(n) equivariance).
3. A TFN with vector outputs (SE(3) equivariance).

Exercise 7.2 (Tensor products)**.** 1. Compute the tensor product $1 \otimes 1$ for SO(3) and verify $D^1 \otimes D^1 = D^0 \oplus D^1 \oplus D^2$.

2. Implement with `e3nn` and verify.
3. Build an equivariant layer that takes two vectors and produces a scalar (dot product), a vector (cross product), and an order-2 tensor.

Exercise 7.3 (EGNN from scratch)**.** Implement a complete EGNN for molecular energy prediction on the QM9 dataset.

1. Load QM9 from PyTorch Geometric.
2. Implement 4 EGNN layers.
3. Train on the U_0 property (internal energy).
4. Verify E(3) invariance of predictions.

Exercise 7.4 (Invariant vs. equivariant comparison)**.** Compare SchNet (invariant) and EGNN (equivariant) on molecular force prediction (a vectorial quantity).

1. Why can SchNet not directly predict forces?

2. How are forces obtained from an energy model?
3. Does EGNN directly predict equivariant forces?

CHAPTER 7. EQUIVARIANT AND INVARIANT NETWORKS

Chapter 8

Point Cloud Representations

In 2017, Charles Qi and collaborators at Stanford published PointNet, a neural network architecture capable of processing 3D point clouds directly — without converting them to voxels or meshes. The key idea is elegant: since a point cloud is an *unordered* set, the architecture must be *permutation invariant*. PointNet achieves this by applying a shared function to each point and then aggregating via a global max. PointNet++, Dynamic Graph CNN (DGCNN), and transformer-based methods subsequently extended this approach by capturing local structures and neighbourhood relationships. This chapter explores these architectures, from theoretical foundations (Zaheer's theorem on set functions) to applications in autonomous driving, robotics, and 3D reconstruction.

Intuition

A 3D point cloud is an **unordered** set of points $\{x_1, \dots, x_n\} \subset \mathbb{R}^3$. Unlike images (regular grids) or graphs (combinatorial structure), it has no canonical connectivity. The challenge is to design architectures that are **permutation invariant** and, ideally, **equivariant** with respect to geometric transformations (rotations, translations).

8.1 The permutation invariance problem

Definition 8.1 (Permutation-invariant function)**.** A function $f : \mathbb{R}^{n\times d} \to \mathbb{R}^k$ is **permutation invariant** if for every permutation matrix $\Pi \in \{0,1\}^{n\times n}$:

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

Theorem 8.2 (Characterization — Zaheer et al. 2017)**.** *Any continuous permutation-invariant function $f : 2^{\mathbb{R}^d} \to \mathbb{R}$ can be written as:*

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

for continuous functions $\phi : \mathbb{R}^d \to \mathbb{R}^q$ and $\rho : \mathbb{R}^q \to \mathbb{R}$ (provided q is sufficiently large).

Remark 8.3. This theorem underpins the **Deep Sets** architecture and, by extension, **PointNet**. The key operation is sum (or max) aggregation, which guarantees permutation invariance.

8.2 PointNet

Definition 8.4 (PointNet architecture — Qi et al. 2017)**. PointNet** directly processes a point cloud of n points $X \in \mathbb{R}^{n\times 3}$:

1. **Spatial transformer**: a mini-network (T-Net) predicts a matrix $T \in \mathbb{R}^{3\times 3}$ to align the cloud: $X' = X \cdot T$.
2. **Shared MLP**: each point is transformed independently by an MLP $\phi : \mathbb{R}^3 \to \mathbb{R}^{1024}$: $h_i = \phi(x'_i)$.
3. **Symmetric aggregation**: global max-pooling produces a global feature vector: $g = \max_{i=1}^{n} h_i$.
4. **Classification/segmentation head**: a final MLP on g (or $[g; h_i]$ for segmentation).

PointNet — fundamental equation

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

where ϕ is a pointwise MLP and ρ a global MLP.

Proposition 8.5 (Universal approximation of PointNet)**.** PointNet can approximate any continuous permutation-invariant function on compact point clouds, in the sense of Theorem 8.2.

Limitations of PointNet

PointNet processes each point **independently** before aggregation: it does not capture local structures (neighborhoods). Two point clouds with the same points but different local geometries may yield the same global vector.

8.3 PointNet++

Definition 8.6 (PointNet++ architecture — Qi et al. 2017)**. PointNet++** introduces a hierarchy of local processing:

1. **Sampling**: select n' centers via farthest point sampling (FPS).
2. **Grouping**: for each center c_j, select the K nearest neighbors (or ball query of radius r).
3. **Local PointNet**: apply PointNet on each group, producing a feature $g_j \in \mathbb{R}^{d'}$.
4. **Iteration**: repeat on the centers with their features, creating a multi-scale pyramid.

Definition 8.7 (Set Abstraction Layer)**.** The **Set Abstraction** (SA) layer combines the three operations:

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

where $\mathcal{N}(c_j)$ is the neighborhood of center c_j.

Intuition

PointNet++ is to point clouds what CNNs are to images: it applies local operations hierarchically. The difference is the absence of a grid: neighborhoods are determined dynamically.

8.4 Point cloud convolutions

Definition 8.8 (Continuous convolution on point clouds)**.** A **continuous convolution** on a point cloud is:

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

where $g : \mathbb{R}^3 \to \mathbb{R}^{d' \times d}$ is a continuous filter and $h_j \in \mathbb{R}^d$ are the features of point x_j.

Example 8.9 (Convolution-based architectures)**.**

- **PointConv** (Wu et al. 2019): g is parameterized by an MLP applied to relative position and local density.
- **KPConv** (Thomas et al. 2019): g is defined by fixed kernel points $\{y_k\}_{k=1}^K$ with correlation functions: $g(x) = \sum_k W_k \cdot \max(0, 1 - \|x - y_k\| / \sigma)$.
- **DGCNN** (Wang et al. 2019): convolution on a dynamic k-nearest neighbor graph in feature space.

8.5 Rotation equivariance

Definition 8.10 (SO(3) equivariance)**.** A function $f : \mathbb{R}^{n\times 3} \to \mathbb{R}^{n\times d}$ is SO(3)**-equivariant** if for every rotation $R \in \mathrm{SO}(3)$:

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

where $\rho(R)$ is a representation of R acting on the output space.

Theorem 8.11 (Equivariant convolutions — Thomas et al. 2018)**.** *Equivariant filters under* SO(3) *are expressed in the spherical harmonics basis:*

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

where $Y^{(\ell)}$ are spherical harmonics of degree ℓ, $R^{(\ell)}$ is a learned radial function, and $\hat{x} = x/\|x\|$.

Remark 8.12. Equivariant architectures (Tensor Field Networks, SE(3)-Transformers, EGNN) guarantee that predictions transform correctly under rotation. This eliminates the need for data augmentation with random rotations.

8.6 Applications

Example 8.13 (3D object classification)**.** On the **ModelNet40** benchmark (12,311 models, 40 classes):

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

Example 8.14 (Semantic scene segmentation)**.** For LiDAR point cloud segmentation (S3DIS, ScanNet), PointNet++ and KPConv achieve competitive performance. Segmentation assigns each point a semantic label (floor, wall, furniture, etc.).

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

class PointNetPPLayer(torch.nn.Module):
    """A Set Abstraction layer from 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)
        # Local PointNet
        x_out = self.conv(x, (pos, pos[idx]), edge_index)
        return x_out, pos[idx], batch[idx]
```

8.7 Exercises

Exercise 8.1. Show that global max-pooling is permutation invariant. Give a counterexample showing that concatenation is not.

Exercise 8.2. Compute the number of parameters in PointNet for a cloud of $n = 1024$ points in 3D, with a pointwise MLP $[3, 64, 128, 1024]$ and a classifier $[1024, 512, 256, 40]$.

Exercise 8.3 (Equivariance)**.** Show that PointNet with T-Net is **approximately** rotation invariant but not exactly equivariant. What property is missing?

Exercise 8.4 (Implementation)**.** Implement a simple PointNet in PyTorch and train it on ModelNet10. Compare with and without the T-Net.

Exercise 8.5 (Point cloud convolution)**.** Explain why KPConv uses fixed kernel points rather than an MLP to define the filter. What advantage does this provide in terms of efficiency and interpretability?

Chapter 9

Applications

Geometric deep learning is not a theoretical exercise: it solves concrete problems that classical methods could not address. In chemistry, GNNs predict molecular properties with accuracy rivalling ab initio quantum mechanics, but a thousand times faster. In particle physics, they reconstruct trajectories in the detectors at CERN's LHC. In biology, AlphaFold (Jumper et al., 2021) uses geometric transformers to predict the 3D structure of proteins, solving a problem that had been open for fifty years. In social networks, GNNs detect communities and predict links. This chapter surveys these applications, showing how geometric principles translate into practical performance.

Intuition

Geometric deep learning finds applications in numerous domains where data possesses a natural non-Euclidean structure: molecules (graphs), social networks (graphs), surfaces (manifolds), physics simulations (meshes and particles). This chapter presents the most impactful applications.

9.1 Drug discovery

Definition 9.1 (Molecular graph representation)**.** A molecule is represented as a graph $G = (V, E)$ where:

- Nodes $v \in V$ represent atoms, with features h_v (atomic type, charge, hybridization, etc.).
- Edges $e \in E$ represent chemical bonds, with features h_e (bond type, distance, etc.).

Example 9.2 (Molecular property prediction)**.** The **MoleculeNet** benchmark (Wu et al. 2018) includes several tasks:

1. **ESOL**: aqueous solubility prediction (regression).
2. **BBBP**: blood-brain barrier penetration (binary classification).
3. **Tox21**: toxicity across 12 biological targets (multi-task classification).

GNNs (SchNet, DimeNet, GemNet) achieve the best published results on these benchmarks thanks to their ability to encode molecular structure.

Definition 9.3 (Message passing for molecules)**.** For a molecular graph, SchNet's message passing (Schütt et al. 2017) integrates interatomic distances:

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

where $x_i \in \mathbb{R}^3$ are atomic positions. The continuous distance filter makes the model invariant under rotation and translation.

```
import torch
from torch_geometric.nn import SchNet

# SchNet for molecular energy prediction
model = SchNet(
    hidden_channels=128,
    num_filters=128,
    num_interactions=6,
    num_gaussians=50,
    cutoff=10.0,
)

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

9.2 Molecular property prediction

Theorem 9.4 (Universality of equivariant GNNs for molecules)**.** *An* E(3)*-equivariant GNN with tensorial features of sufficient degree can approximate any continuous equivariant function on molecular graphs with atomic positions.*

Remark 9.5. Recent models (DimeNet, SphereNet, PaiNN, MACE) use not only distances but also angles and dihedrals to improve expressiveness.

9.3 Social network analysis

Definition 9.6 (Social network tasks)**.** On a social network modeled as a graph:

1. **Node classification**: predict the interests or community of a user.
2. **Link prediction**: predict future connections between users.
3. **Community detection**: cluster the nodes.

Example 9.7 (Node classification on Cora)**.** The **Cora** dataset contains 2,708 scientific papers (nodes) connected by citations (edges). Each paper must be classified among 7 categories. A 2-layer GCN achieves $\sim 81\%$ accuracy with only 20 labels per class.

Proposition 9.8 (Over-smoothing)**.** For a GCN with L layers, node embeddings converge to a subspace of dimension at most 1 as $L \to \infty$:

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

where d_v is the node degree. This is the **over-smoothing** phenomenon that limits GNN depth.

9.4 Recommender systems

Definition 9.9 (Collaborative filtering on bipartite graphs)**.** A recommender system can be modeled as a bipartite graph $G = (U \cup I, E)$ where U are users, I are items, and $(u, i) \in E$ if user u interacted with item i.

Example 9.10 (PinSage — Ying et al. 2018)**. PinSage** is a GNN deployed by Pinterest on a graph of 3 billion nodes and 18 billion edges. It uses:

- **Neighbor sampling**: importance-weighted random walks.
- **Local aggregation**: similar to GraphSAGE.
- **Mini-batch training**: on sampled subgraphs.

9.5 Physics simulation

Definition 9.11 (Learned simulator via GNN)**.** A learned simulator models a physical system as an interaction graph:

- Nodes represent particles (position, velocity, type).
- Edges represent interactions (nearest neighbors).

The GNN predicts the acceleration of each particle:

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

Example 9.12 (GNS — Sanchez-Gonzalez et al. 2020)**.** The **Graph Network Simulator** (GNS) simulates fluids, granular materials, and deformable solids. It generalizes to unseen initial conditions and different domain geometries.

Theorem 9.13 (Energy conservation by construction)**.** *By parameterizing a GNN as a Hamiltonian H_θ and integrating Hamilton's equations:*

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

one obtains a simulator that conserves energy by construction (Hamiltonian GNN).

9.6 Weather prediction

Example 9.14 (GraphCast — Lam et al. 2023)**. GraphCast** (DeepMind) uses a GNN on an icosahedral mesh of Earth for 10-day weather forecasting. It outperforms ECMWF's HRES numerical model on 90% of metrics, with computation time of **less than one minute** versus hours for physical models.

The architecture uses:

1. An **encoder** from grid to multi-resolution mesh.
2. A **processor** GNN on the mesh (16 message-passing layers).
3. A **decoder** from mesh back to grid.

Remark 9.15. The success of GraphCast illustrates the potential of GNNs for problems defined on non-planar domains (Earth's sphere).

9.7 Exercises

Exercise 9.1. For a benzene molecule (C_6H_6), construct the molecular graph (nodes, edges, features). How many messages are exchanged in one iteration of message passing?

Exercise 9.2. Explain why permutation invariance of nodes is essential for molecular graphs. What would happen if atoms were numbered differently?

Exercise 9.3 (Over-smoothing)**.** For a complete graph K_n with a GCN without bias or nonlinearity, show that features converge to a constant vector after L layers when L is large.

Exercise 9.4 (Implementation)**.** Implement a GNN for solubility prediction on the ESOL dataset with PyTorch Geometric. Compare GCN, GAT, and SchNet.

Exercise 9.5 (Simulation)**.** Implement a simple GNS to simulate N particles interacting via a Lennard-Jones potential. Compare with classical Verlet integration.

Chapter 10

Connections to TDA

Intuition

Topological Data Analysis (TDA) and geometric deep learning share a common concern: exploiting the **structure** of data. TDA provides topological invariants (Betti numbers, persistent homology) that complement features learned by GNNs and improve their **expressiveness**.

10.1 Review of persistent homology

Definition 10.1 (Persistent homology)**.** Given a filtration of simplicial complexes $K_0 \subseteq K_1 \subseteq \cdots \subseteq K_m$, **persistent homology** tracks the evolution of homology classes through the filtration. Each class is born at index b and dies at index $d > b$, yielding a point (b, d) in the **persistence diagram**.

Remark 10.2. For a graph $G = (V, E)$, different filtrations can be constructed:

- **By degree**: $f(v) = \deg(v)$ and $f(e) = \max(f(u), f(v))$.
- **By centrality**: $f(v) = \text{betweenness}(v)$.
- **By learned features**: $f(v) = g_\theta(h_v)$ where g_θ is a neural network.

10.2 Topological features in GNNs

Definition 10.3 (Topological features for graphs)**. Topological features** of a graph are extracted from its persistence diagrams:

- H_0: connected components and their stability.
- H_1: cycles and their persistence.

These features are vectorized (landscapes, persistence images) and concatenated with GNN features.

Proposition 10.4 (Complementarity)**.** Topological features capture global graph properties (cycles, cavities) that local message passing cannot detect in a bounded number of iterations: detecting a cycle of length k requires at least $k/2$ message-passing layers.

```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):
    """Compute topological features of a point cloud."""
    rips = RipsComplex(points=pos, max_edge_length=max_edge)
    st = rips.create_simplex_tree(max_dimension=2)
    st.persistence()

    # Diagrams by dimension
    dgm_h0 = st.persistence_intervals_in_dimension(0)
    dgm_h1 = st.persistence_intervals_in_dimension(1)

    # Vectorization via persistence images
    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 Topological message passing

Definition 10.5 (Message passing on simplicial complexes)**. Simplicial message passing** (Bodnar et al. 2021) generalizes graph message passing to higher-dimensional simplicial complexes:

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

where $\mathcal{B}(\sigma)$ are the boundary faces and $\mathcal{C}(\sigma)$ are the coboundary cofaces of simplex σ.

Intuition

In a graph, messages flow along edges (1-simplices). In a simplicial complex, messages also flow along triangles (2-simplices) and higher-dimensional structures. This enables capturing **higher-order** relationships between nodes.

Definition 10.6 (Message passing on cell complexes)**. CW Networks** (Bodnar et al. 2021) extend message passing to cell complexes, which generalize simplicial complexes by allowing cells of arbitrary shape (not just simplices).

10.4 The Weisfeiler-Leman hierarchy and topology

Definition 10.7 (Weisfeiler-Leman test)**.** The **Weisfeiler-Leman** (WL) test of level k is a color refinement algorithm that characterizes graphs: two graphs not distinguished by k-WL are called **k-WL equivalent**.

Theorem 10.8 (GNN expressiveness — Xu et al. 2019, Morris et al. 2019)**.** *A message-passing GNN is* ***at most as powerful*** *as the* 1*-WL test for distinguishing non-isomorphic graphs.*

Theorem 10.9 (Beyond WL via topology — Bodnar et al. 2021)**.** *Message passing on simplicial complexes with topological features (persistent homology) can distinguish graphs that k-WL cannot distinguish, for any fixed k.*

Example 10.10 (Rook graphs)**.** The Rook graphs $R_{3\times3}$ and $R_{4\times4}$ are not distinguished by 3-WL. However, their H_1 persistence diagrams (with degree filtration) differ: $R_{4\times4}$ has more persistent cycles.

10.5 Expressiveness and topology

Proposition 10.11 (Topological enrichment)**.** By adding persistent homology features to GNN node features, the resulting expressiveness is **strictly greater** than that of the GNN alone for graph classification.

Definition 10.12 (Joint learned filtration)**.** In a **joint learned filtration**, filtration values are defined by the GNN itself:

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

where $h_v^{(L)}$ is the final embedding of node v after L message-passing layers. Persistence is computed on this filtration, and gradients backpropagate through the persistence computation (cf. differentiable persistence).

GNN + TDA pipeline

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

10.6 Exercises

Exercise 10.1. For the Petersen graph, compute the H_0 and H_1 persistence diagrams with the degree filtration. All nodes have the same degree: what can you conclude?

Exercise 10.2. Show that two isomorphic graphs have the same persistence diagrams for any filtration defined from structural graph properties.

Exercise 10.3 (Simplicial message passing)**.** Write the message-passing equations for a triangle (v_1, v_2, v_3) in a simplicial complex. What messages does the edge (v_1, v_2) receive?

Exercise 10.4. Give an example of two non-isomorphic graphs with the same persistence diagrams for the degree filtration. Propose a filtration that distinguishes them.

Exercise 10.5 (Project)**.** Implement a GNN enriched with topological features on the ZINC benchmark and compare with a baseline GCN. Measure the improvement in MAE.

Chapter 11

Current Research Directions

Intuition

Geometric deep learning is a rapidly expanding field. This chapter presents the most active research directions and open problems that will shape the next advances in the field.

11.1 Geometric foundation models

Definition 11.1 (Geometric foundation model)**.** A **geometric foundation model** is a large-scale pretrained model on structured data (graphs, molecules, point clouds) that is transferable to multiple downstream tasks via fine-tuning or prompting.

Example 11.2 (Examples of geometric foundation models)**.**

- **GEM** (Fang et al. 2022): foundation model for molecules, pretrained on 20M structures.
- **Uni-Mol** (Zhou et al. 2023): unified 2D/3D molecular representation.
- **Point-MAE** (Pang et al. 2022): masked autoencoder for 3D point clouds.

Remark 11.3. Unlike text foundation models (GPT, Claude), geometric foundation models must respect data **symmetries** (permutation invariance, rotation equivariance). This imposes strong architectural constraints.

Proposition 11.4 (Scaling challenges)**.** Scaling GNNs poses specific problems:

1. **Neighborhood explosion**: k layers imply $\mathcal{O}(d^k)$ neighbors ($d =$ average degree).
2. **Over-smoothing**: representations converge with depth.
3. **Over-squashing**: long-range information is compressed through bottlenecks.
4. **Variable-size graphs**: batching is non-trivial.

11.2 Equivariant transformers

Definition 11.5 (Geometric transformer)**.** A **geometric transformer** is a transformer whose attention mechanism integrates geometric information while respecting symmetries:

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

where $b(x_i, x_j)$ is a geometry-dependent attention bias (distance, angle, relative orientation).

Example 11.6 (Example architectures)**.**

- **SE(3)-Transformer** (Fuchs et al. 2020): SE(3)-equivariant attention with tensor features.
- **Equiformer** (Liao & Smidt, 2023): equivariant transformer with tensor products in attention.
- **GPS** (Rampašek et al. 2022): combines local message passing and global attention.

Theorem 11.7 (Expressiveness of Graph Transformers)**.** *A Graph Transformer with spectral positional encoding and global attention is **strictly more expressive** than the* 1*-WL test. With* SE(3)*-equivariant positional encodings, it can approximate any continuous equivariant function.*

Equivariant attention

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

where $V : \mathbb{R}^d \times \mathbb{R}^3 \to \mathbb{R}^{d'}$ is a value function depending on relative position, and α_{ij} is SE(3)-invariant.

11.3 Diffusion models on manifolds

Definition 11.8 (Riemannian diffusion)**.** A **Riemannian diffusion model** (De Bortoli et al. 2022) generalizes diffusion models (DDPM) to Riemannian manifolds $(\mathcal{M}, g)$:

1. **Forward process**: Riemannian Brownian motion $dX_t = \sqrt{2\beta_t}\, dB_t^{\mathcal{M}}$.
2. **Score matching**: learn the score $\nabla_{\mathcal{M}} \log p_t(x)$ on the manifold.
3. **Reverse process**: integrate the reverse equation on $\mathcal{M}$ with the learned score.

Example 11.9 (Applications of geometric diffusion)**.**

- **3D molecule generation**: diffusion on $\mathbb{R}^{3n}$ with SE(3) equivariance (EDM, Hoogeboom et al. 2022).
- **Conformation generation**: diffusion on torsion angles (toric manifold).
- **Protein generation**: diffusion on $\mathrm{SO}(3)^n$ for residue frames (RFDiffusion, Watson et al. 2023).

Technical challenges

Diffusion on manifolds requires:

- The exponential map $\exp_x : T_x\mathcal{M} \to \mathcal{M}$ for integration steps.
- The Riemannian logarithm $\log_x : \mathcal{M} \to T_x\mathcal{M}$ for the score.
- Parallel transport to move vectors between tangent spaces.

11.4 Geometric self-supervised learning

Definition 11.10 (Self-supervised learning on graphs)**.** **Self-supervised learning** (SSL) on graphs learns representations without labels via auxiliary tasks:

1. **Contrastive**: maximize similarity between two augmented views of the same graph (GraphCL, GCA).
2. **Predictive**: predict masked properties (node features, edges, subgraphs).
3. **Generative**: reconstruct the graph from a compressed representation (GraphMAE).

Proposition 11.11 (Geometric augmentations)**.** Augmentations for graphs include:

- Random node or edge dropout.
- Feature perturbation.
- Random subgraph extraction.
- Heat diffusion (feature smoothing).

For 3D data, add: rotation, translation, Gaussian noise, partial sampling.

Theorem 11.12 (Contrastive SSL guarantees)**.** *Under regularity assumptions on the data distribution and augmentations, contrastive representations are* ***linearly separable*** *for downstream tasks, with an error bound controlled by the alignment and uniformity of the representations.*

11.5 Open problems

Remark 11.13 (Major open problems).

1. **Optimal expressiveness**: what is the minimal GNN architecture to match k-WL for a given k?
2. **Over-squashing**: how to enable long-range information flow without expensive global attention ($\mathcal{O}(n^2)$)?
3. **Out-of-distribution generalization**: do GNNs generalize to graphs with very different structures from training?
4. **Approximate equivariance**: how to handle approximate symmetries (quasi-symmetries) of real-world data?

5. **Interpretability**: how to explain GNN decisions in terms of meaningful substructures?
6. **Scalability**: how to train GNNs on graphs with billions of nodes?

Definition 11.14 (Over-squashing)**. Over-squashing** refers to the phenomenon where information from distant nodes is exponentially compressed when passing through message-passing layers. Formally, the Jacobian:

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

decays exponentially with graph distance if $\lambda_{\max} < 1$.

11.6 Exercises

Exercise 11.1. Explain why Laplacian positional encodings are not unique (sign, ordering). Propose a solution to make them invariant.

Exercise 11.2. For a molecular graph, propose three contrastive augmentations that preserve chemical identity and three that do not. Justify.

Exercise 11.3 (Over-squashing)**.** Compute the bound on $\left\| \frac{\partial h_v^{(L)}}{\partial h_u^{(0)}} \right\|$ for a path graph of length L with a GCN with unit weights and ReLU activation.

Exercise 11.4. Show that Brownian motion on the sphere S^2 converges to the uniform distribution. What is the convergence rate as a function of the first nonzero eigenvalue of the Laplacian?

Exercise 11.5 (Project)**.** Implement a simple equivariant diffusion model to generate 3D point clouds (e.g., shapes from ShapeNet). Evaluate quality using Chamfer distance and Betti numbers.

Bibliography

[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.