An Introduction to Graph Neural Networks (GNNs) for Molecules

Hosein Fooladi

09.12.2025

Contact:

hosein.fooladi@univie.ac.at

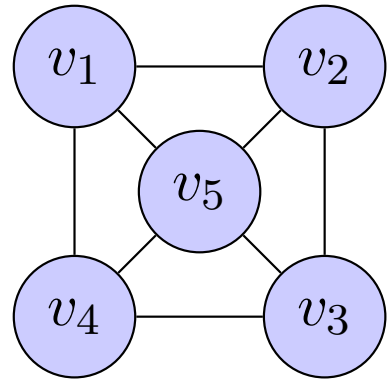
https://hfooladi.github.io

Course Outline

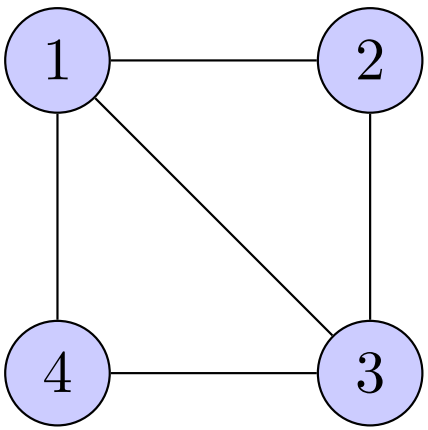
- What is a graph?
- Representing molecules as graphs
- Different tasks with GNNs
- Message passing on molecular graphs
- Introduction to PyTorch Geometric
- Developing a GCN from scratch
- Hands-on implementation

What is a Graph?

- A graph $G = (V, E)$ consists of:
  - Nodes/vertices V
  - Edges E connecting nodes
- Graphs represent relational data
- Natural representation for molecules



Adjacency Matrix



Adjacency matrix $A \in \{0,1\}^{n \times n}$:

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

- $A_{ij} = 1$ if nodes i and j are connected
- $A_{ij} = 0$ otherwise
- Can also have weighted edges where $A_{ij} \in \mathbb{R}$

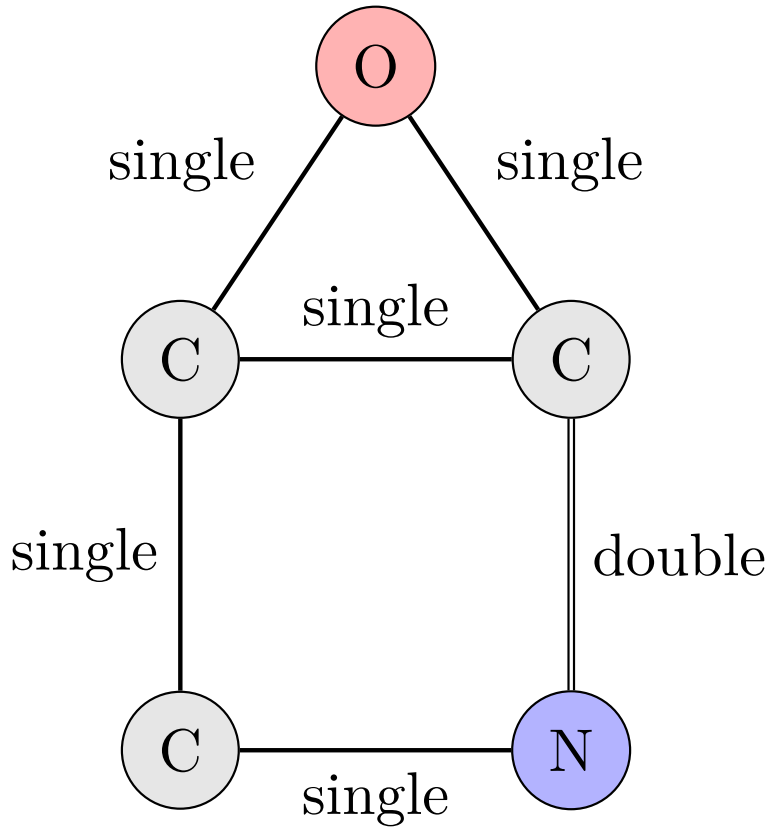
Node and Edge Features

Node Features:

- Each node v_i has features $\mathbf{x}_i$
- Can be categorical or continuous
- For molecules: atom type, charge, hybridization, etc.

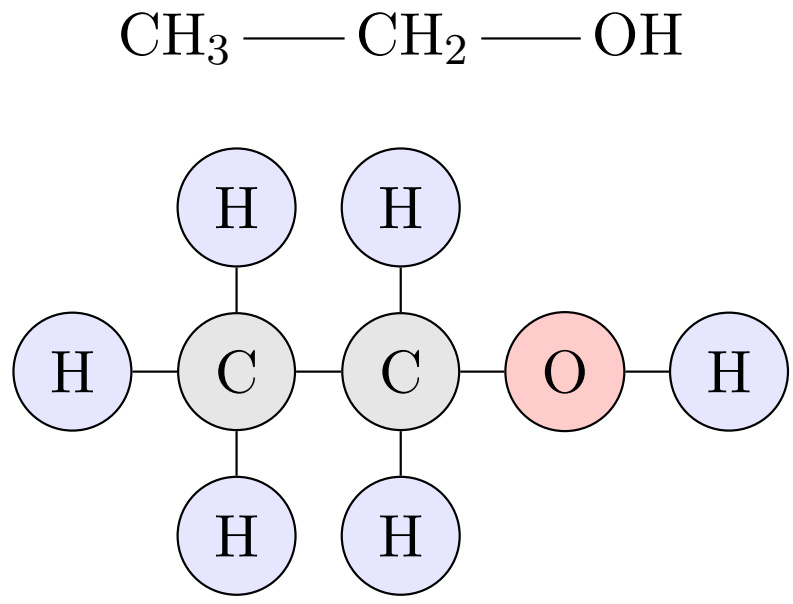
Edge Features:

- Each edge (v_i, v_j) has features $\mathbf{e}_{ij}$
- For molecules: bond type, distance, etc.



Representing a Molecule as a Graph

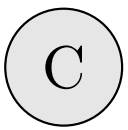
- Nodes = Atoms
- Edges = Bonds
- Node features = Atom properties
- Edge features = Bond properties



Node Features for Molecules

Common atom (node) features:

- Atomic number (one-hot or embedding)
- Atom type (C, H, O, N, etc.)
- Formal charge
- Hybridization state (sp, sp^2, sp^3)
- Number of hydrogens attached
- Is in aromatic ring?
- Chirality
- Partial charge

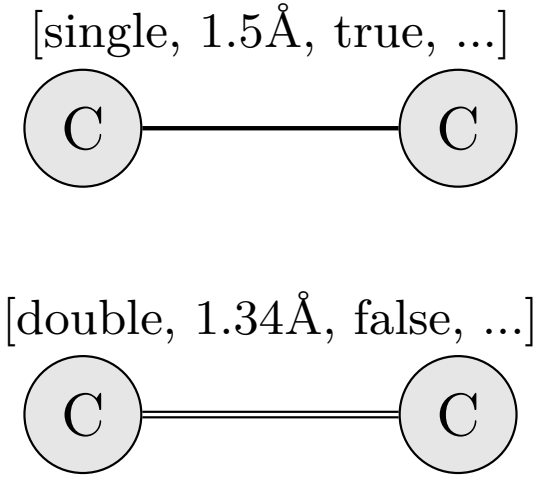


Feature vector: [6, 0, sp^3, 3, false, ...]

Edge Features for Molecules

Common bond (edge) features:

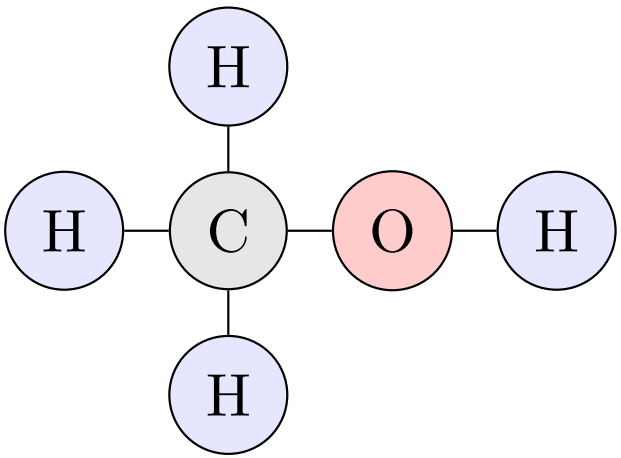
- Bond type (single, double, triple, aromatic)
- Bond distance
- Is in ring?
- Conjugation
- Stereochemistry (cis/trans)



Methanol Example

Methanol (CH_3OH)

CH_3 —— OH



Graph Representation:

Nodes:

- C: [6, 0, sp^3, 3, 0]
- O: [8, 0, sp^3, 1, 0]
- H_1-H_4: [1, 0, s, 0, 0]

Adjacency Matrix:

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Node ordering: C, O, H_1, H_2, H_3, H_4

Real-World Molecular Features

Node Features (21 dimensions per atom in practice):

Feature Group	**Dimensions**
Atom type (one-hot) (C, O, N, H, F, P, S, Cl, Br, I, Other)	11
Formal charge	1
Aromaticity	1
Ring membership	1
Degree (number of bonds)	1
Total hydrogens	1
Radical electrons	1
Hybridization (one-hot) (sp, sp^2, sp^3, Other)	4
Total	**21**

Real-World Molecular Features

Edge Features (6 dimensions per bond in practice):

- Bond type (one-hot, 4D)
  - Single, double, triple, aromatic
- Conjugation (1D)
- Ring membership (1D)

Why this matters:

- Chemical significance drives feature choice
- More features = better predictions
- But: curse of dimensionality

Hands-On Session

Let's put theory into practice!

- Open your Python notebook
- We'll learn how to represent graph in Python
- We'll learn graph representation in PyTorch Geometric
- Resources: `https://github.com/HFooladi/GNNs-For-Chemists/blob/main/notebooks/01_GNN_representation.ipynb`

Different Tasks with Graph Neural Networks

Node-level tasks:

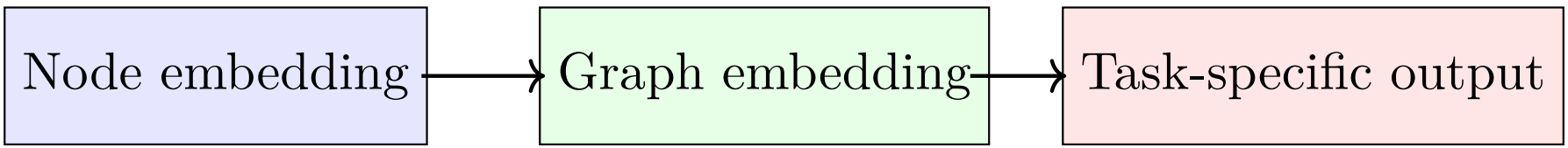
- Atom property prediction
- Reaction site prediction
- Partial charge prediction

Edge-level tasks:

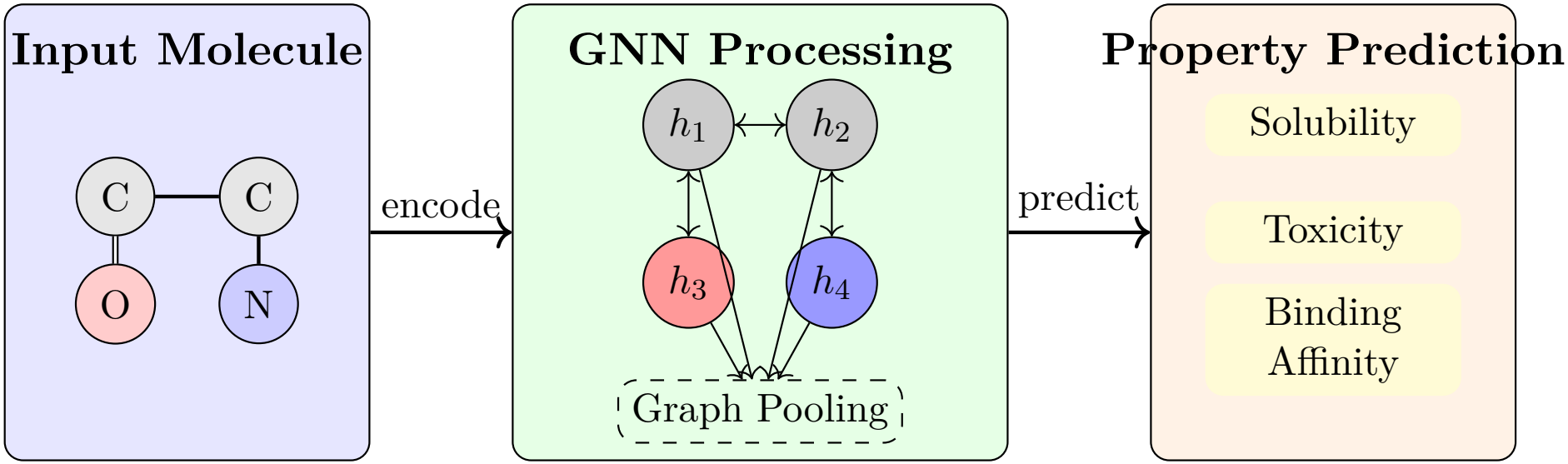
- Bond property prediction
- Link prediction (e.g., potential bonds)

Graph-level tasks:

- Molecular property prediction
  - Solubility
  - Toxicity
  - Drug efficacy
  - Binding affinity
- Molecule generation
- Molecule classification



Graph-Level Property Prediction



Process:

- Encode molecular structure
- Message passing between atoms
- Pool features → molecule representation
- Predict properties

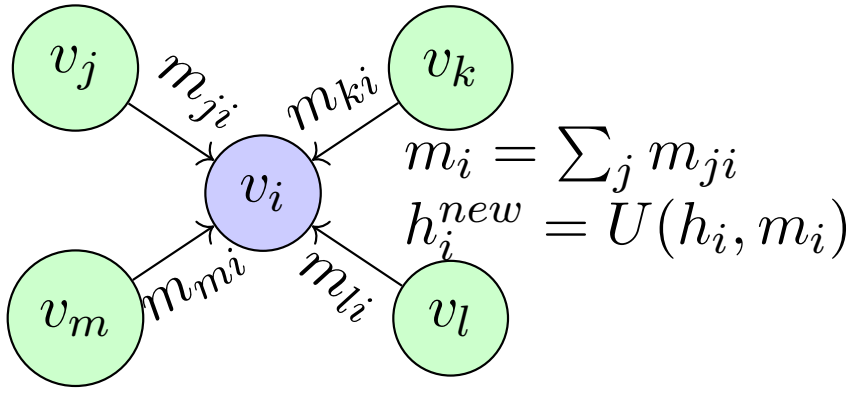
Applications:

- Binding affinity prediction
- Virtual screening
- Toxicity screening
- Reaction yield prediction

Message Passing Neural Networks

Message Passing Framework:

1. Each node sends messages to its neighbors
2. Each node aggregates incoming messages
3. Each node updates its features based on aggregated messages



Mathematical formulation:

$$m_i^{(l+1)} = \sum_{j \in \mathcal{N}(i)} M(h_i^{(l)}, h_j^{(l)}, e_{ij}) \quad (1)$$

$$h_i^{(l+1)} = U(h_i^{(l)}, m_i^{(l+1)}) \quad (2)$$

where M is the message function and U is the update function.

Message passing allows the model to:

- Capture local chemical environment
- Learn hierarchical representations
- Handle molecules of different sizes

What is Equivariance?

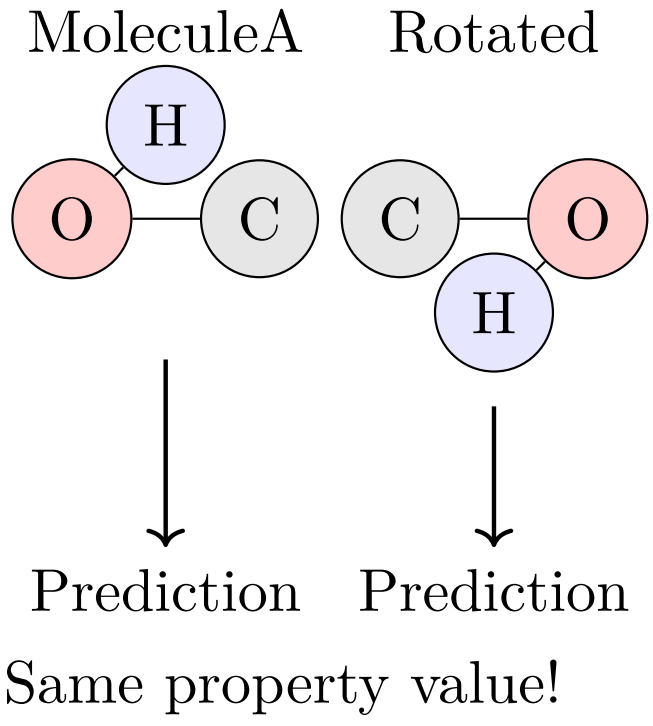
Equivariance means that transformations of input lead to predictable transformations of output.

For molecules:

- Rotation/translation of a molecule shouldn't change its properties
- Permutation equivariance

Why it matters:

- Ensures consistent predictions regardless of orientation
- Reduces required training data
- More physically meaningful representations



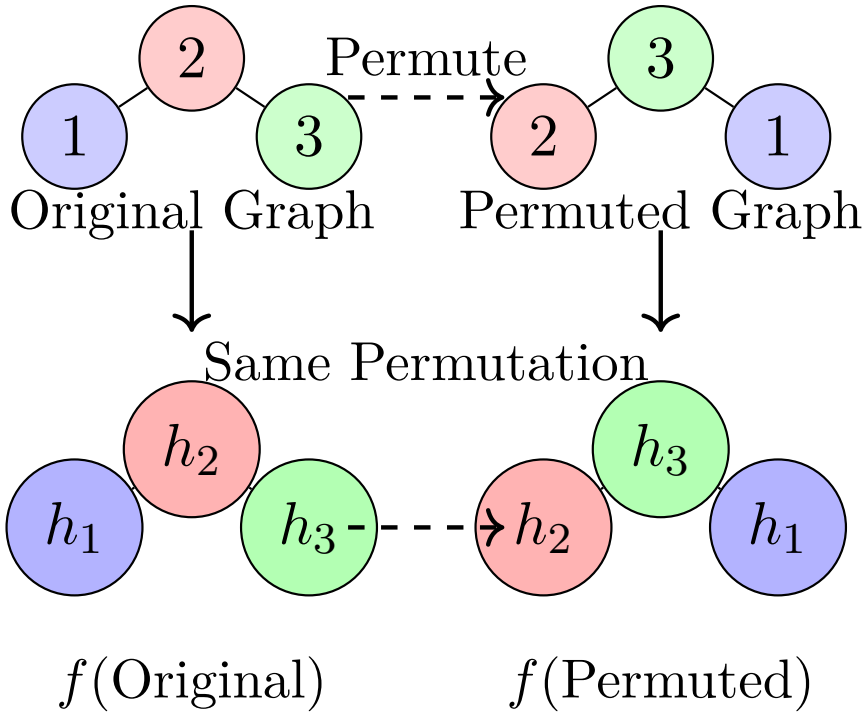
Permutation Equivariance in GNNs

Permutation Equivariance:

- Reordering nodes before applying a GNN should give the same result as applying the GNN first and then reordering nodes
- Critical for molecules where node ordering is arbitrary

Mathematically: for a GNN function f and permutation P:

$$f(PX, PAP^T) = Pf(X, A)$$



Same representation, different order

Hands-On Session

Let's put theory into practice!

- Open your Python notebook
- We'll learn how to implement message passing
- Resources: `https://github.com/HFooladi/GNNs-For-Chemists/blob/main/notebooks/02_GNN_message_passing.ipynb`

Introduction to PyTorch Geometric (PyG)

- PyTorch Geometric (PyG) is a library for deep learning on irregular structures (graphs, point-clouds, etc.)
- Built on top of PyTorch
- Provides tools for working with graphs
- Efficient implementations of many GNN architectures

```
# Installing PyTorch Geometric
pip install torch-geometric

# Basic imports
import torch
from torch_geometric.data import Data
from torch_geometric.nn import GCNConv
```

Downloading and Loading Datasets

```
from torch_geometric.datasets import MoleculeNet

# Download BACE dataset (biological activity prediction)
dataset = MoleculeNet(root='data/molecule_datasets', name='BACE')

print(f'Dataset size: {len(dataset)}')
print(f'Number of features: {dataset.num_features}')

# Look at the first graph
data = dataset[0]

# Splitting into train, validation, test
from torch_geometric.loader import DataLoader
from sklearn.model_selection import train_test_split

train_idx, test_idx = train_test_split(
    range(len(dataset)), test_size=0.2, random_state=42)
train_idx, val_idx = train_test_split(
    train_idx, test_size=0.25, random_state=42)
```

Exploring the Dataset Features

```
# Explore a single molecule
data = dataset[0]
print("Node features shape:", data.x.shape)
print("Edge indices shape:", data.edge_index.shape)

# Visualize the molecule
from rdkit import Chem
from rdkit.Chem import Draw

smiles = data.smiles
mol = Chem.MolFromSmiles(smiles)
img = Draw.MolToImage(mol, size=(300, 300))
plt.imshow(img)
plt.axis('off')

# Analyzing feature distributions
node_features = []
for data in dataset[:100]:
    node_features.append(data.x.mean(dim=0))
average_features = torch.stack(node_features).mean(dim=0)
plt.bar(range(len(average_features)), average_features.numpy())
```

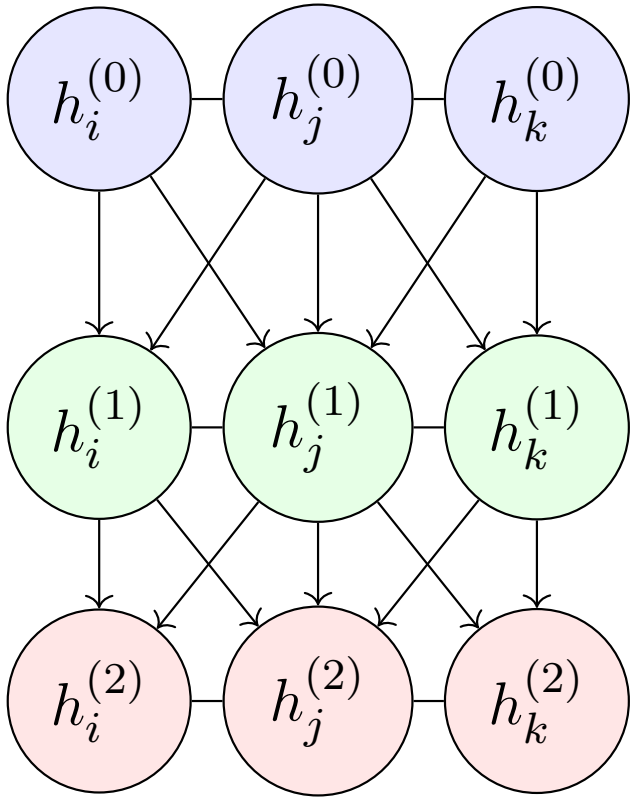
Graph Convolutional Network (GCN) Layer

GCN update rule:

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \quad (3)$$

where:

- $\tilde{A} = A + I$ (adjacency matrix with self-loops)
- $\tilde{D}$ is the degree matrix of $\tilde{A}$
- $H^{(l)}$ is the node feature matrix at layer l
- $W^{(l)}$ is the learnable weight matrix
- σ is a non-linear activation function



Implementing a GCN Layer

```
from torch_geometric.nn import MessagePassing
from torch_geometric.utils import add_self_loops, degree

class GCNConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super().__init__(aggr='add')
        self.lin = nn.Linear(in_channels, out_channels)

    def forward(self, x, edge_index):
        # Add self-loops
        edge_index, _ = add_self_loops(edge_index, num_nodes=x.
            size(0))
        x = self.lin(x)

        # Compute normalization
        row, col = edge_index
        deg = degree(row, x.size(0), dtype=x.dtype)
        deg_inv_sqrt = deg.pow(-0.5)
        norm = deg_inv_sqrt[row] * deg_inv_sqrt[col]

        return self.propagate(edge_index, x=x, norm=norm)

    def message(self, x_j, norm):
        return norm.view(-1, 1) * x_j
```

Building a Complete GNN Model

```
class GCN(torch.nn.Module):
    def __init__(self, num_node_features, num_classes):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(num_node_features, 64)
        self.conv2 = GCNConv(64, 64)
        self.conv3 = GCNConv(64, 64)
        self.lin = nn.Linear(64, num_classes)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index

        # Graph Conv layers
        x = F.relu(self.conv1(x, edge_index))
        x = F.dropout(x, p=0.1, training=self.training)

        x = F.relu(self.conv2(x, edge_index))
        x = F.dropout(x, p=0.1, training=self.training)

        x = F.relu(self.conv3(x, edge_index))

        # Global pooling
        x = global_mean_pool(x, data.batch)

        return self.lin(x)
```

Global Pooling: Aggregating Node Features

After GCN layers, we need **one vector per graph** for graph-level predictions:

Mean Pooling

$$h_G = \frac{1}{|V|} \sum_{v \in V} h_v$$

✓ Balanced representation
× Averages out extremes

Max Pooling

$$h_G = \max_{v \in V} h_v$$

✓ Captures key features
× Loses global context

Sum Pooling

$$h_G = \sum_{v \in V} h_v$$

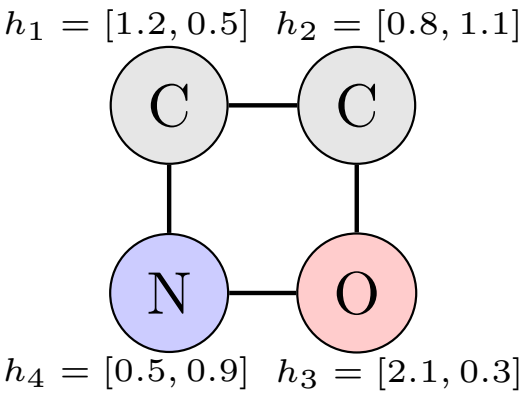
✓ Preserves size info
× Size-dependent

Key Question

Which pooling method should you use? It depends on your molecular property!

Pooling Methods: Visual Comparison

Example: Small molecular graph with 4 nodes



After Pooling:

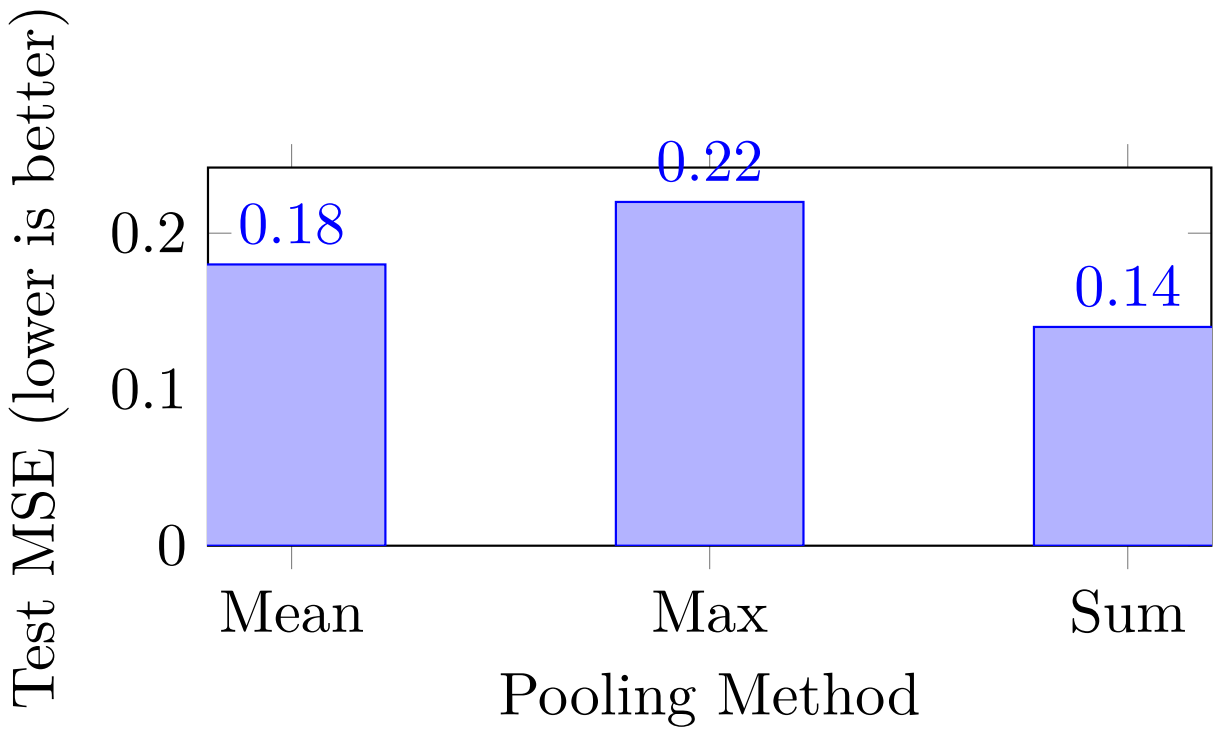
Method	Result	Dim 1, Dim 2
Mean	Average	[1.15, 0.70]
Max	Maximum	[2.1, 1.1]
Sum	Total	[4.6, 2.8]

Observations:

- Mean: Balanced, normalized by size
- Max: Captures oxygen's high feature in dim 1
- Sum: Keeps magnitude, reflects mol size

Experimental Results: ESOL Solubility Dataset

Question: Which pooling method works best for predicting water solubility?



Result

Sum pooling wins! Why? Water solubility correlates with molecular size. Larger molecules have more surface area and functional groups affecting solubility.

When to Use Each Pooling Method

Mean Pooling

- **Best for**: General molecular properties
- **Examples**:
  - Toxicity classification
  - LogP (lipophilicity)
  - Drug-likeness scores
- Size-independent properties

Max Pooling

- **Best for**: Key substructure detection
- **Examples**:
  - Active site identification
  - Pharmacophore matching
  - Toxicophore detection
- Properties dominated by one feature

Sum Pooling

- **Best for**: Size-dependent properties
- **Examples**:
  - Solubility
  - Molecular weight prediction
  - Molar refractivity
  - Surface area estimation
- Properties that scale with molecule size

Pro Tip

Try all three and use validation set to choose! Some papers use **concatenation**: $[h_{mean}, h_{max}, h_{sum}]$

Loss Function and Optimization

Common Loss Functions:

- Binary classification: Binary Cross Entropy
- Multi-class: Cross Entropy
- Regression: Mean Squared Error

Optimization:

- Adam optimizer is commonly used
- Learning rate scheduling can help
- Early stopping based on validation

```
# Binary classification
criterion =
    nn.BCEWithLogitsLoss()

# Regression
criterion = nn.MSELoss()

# Optimizer
optimizer = torch.optim.Adam(
    model.parameters(),
    lr=0.01,
    weight_decay=5e-4
)

# Learning rate scheduler
scheduler =
    torch.optim.lr_scheduler.Reduc
    optimizer,
    mode='min',
    factor=0.5,
```

Training Loop

```python
def train(model, loader, optimizer, criterion):
    model.train()
    total_loss = 0
    for data in loader:
        optimizer.zero_grad()
        out = model(data)
        loss = criterion(out, data.y)
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * data.num_graphs
    return total_loss / len(loader.dataset)

def evaluate(model, loader, criterion):
    model.eval()
    total_loss = 0
    with torch.no_grad():
        for data in loader:
            out = model(data)
            loss = criterion(out, data.y)
            total_loss += loss.item() * data.num_graphs
    return total_loss / len(loader.dataset)

# Training process
for epoch in range(1, 101):
    train_loss = train(model, train_loader, optimizer, criterion)
```

Evaluation on Test Data

Evaluation Metrics:

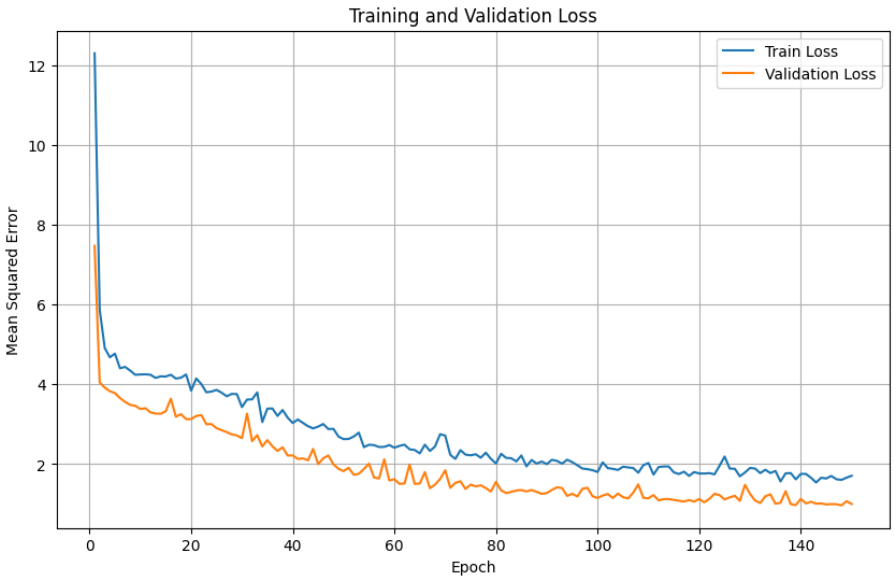
- Classification: Accuracy, F1-score, AUC-ROC
- Regression: RMSE, MAE, R^2

Evaluation Process:

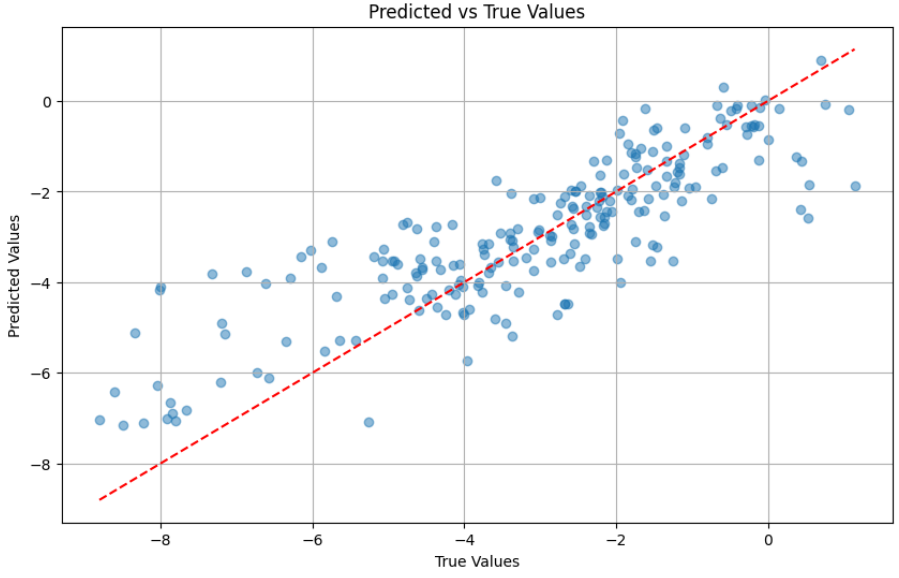
1. Load the best model
2. Run predictions on test set
3. Calculate metrics

```
from sklearn.metrics import
    roc_auc_score

# Load best model
model.load_state_dict(
    torch.load('best_model.pt'))

# Evaluate on test set
model.eval()
preds, targets = [], []

with torch.no_grad():
    for data in test_loader:
        pred = torch.sigmoid(model(
            data))
        preds.append(pred)
        targets.append(data.y)

preds = torch.cat(preds, dim=0).numpy
    ()
targets = torch.cat(targets, dim=0).
    numpy()

# Calculate AUC-ROC
auc = roc_auc_score(targets, preds)
```

Visualizing Results



Loss Curves



Prediction vs Actual

The Oversmoothing Problem

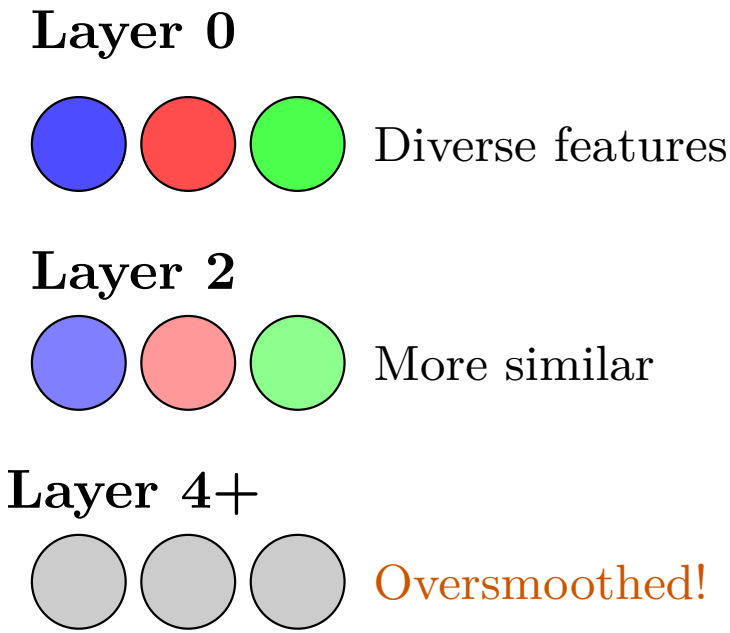
A critical limitation of deep GNNs

What is oversmoothing?

- Node features become increasingly similar with network depth
- Loss of discriminative power
- Performance degrades after 3-4 layers

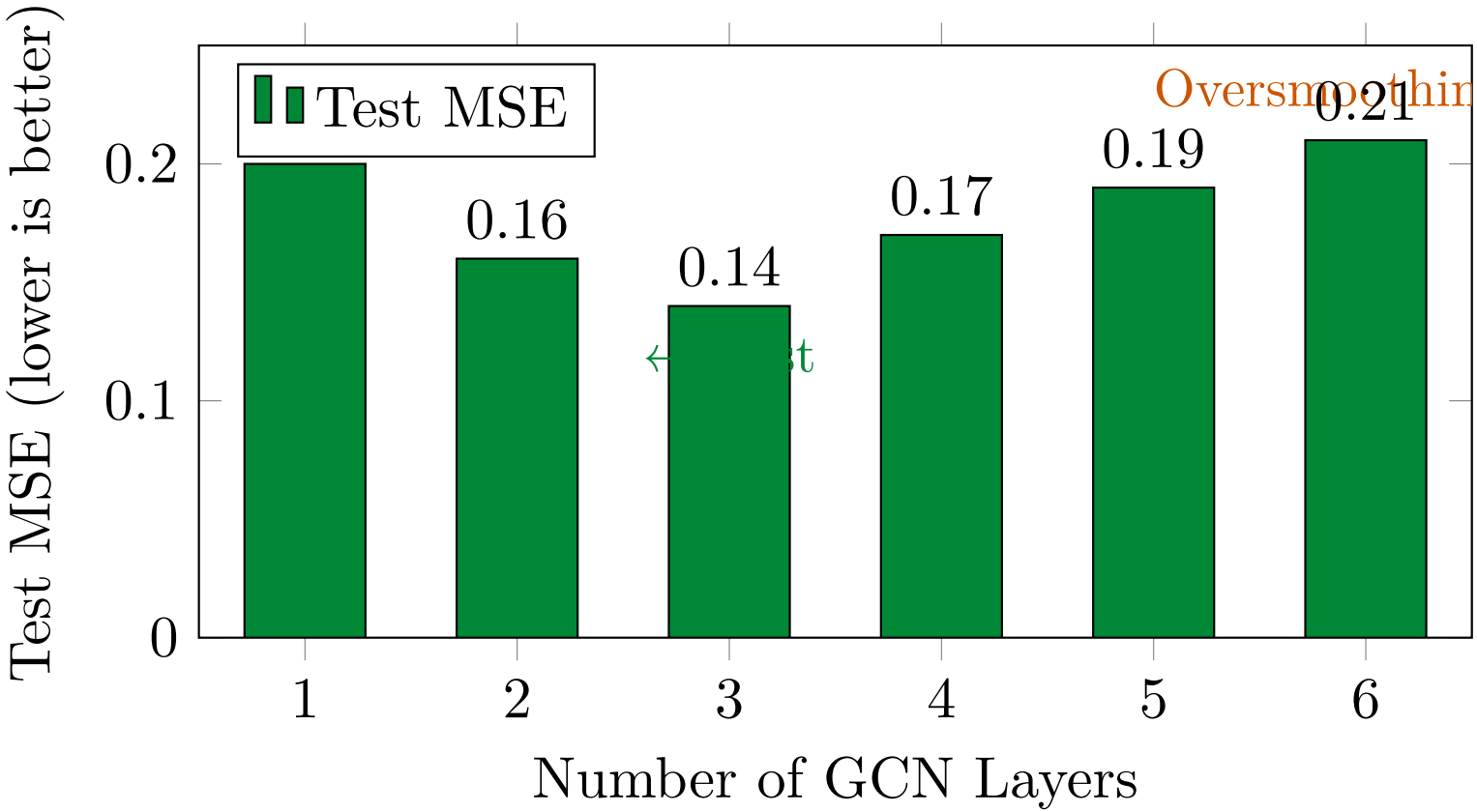
Why does it happen?

- Repeated neighborhood averaging
- Information from distant nodes overwhelms local features
- All nodes converge to global average



Experimental Evidence: Effect of Network Depth

ESOL solubility prediction: How does test performance change with depth?



Key Observation

3 layers achieve optimal performance. Deeper networks (4-6 layers) perform **worse** due to oversmoothing.

Mathematical Perspective on Oversmoothing

Why do features converge?

In each GCN layer, node features are averaged:

$$h_v^{(l+1)} = \sigma\left(\frac{1}{\sqrt{d_v}} \sum_{u \in \mathcal{N}(v)} \frac{h_u^{(l)}}{\sqrt{d_u}} W^{(l)}\right)$$

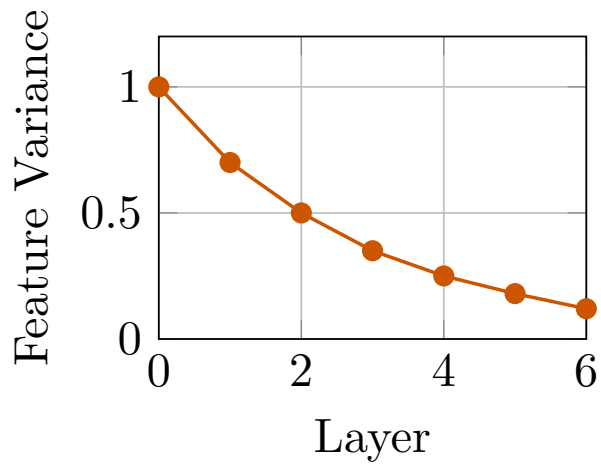
Feature variance decays:

$$\mathrm{Var}(h^{(l)}) \propto \lambda^l$$

where $\lambda < 1$

After many layers:

- Each node "sees" exponentially more neighbors
- Layer k: sees k-hop neighborhood
- In small molecules, this quickly covers entire graph
- Features become weighted average of all nodes

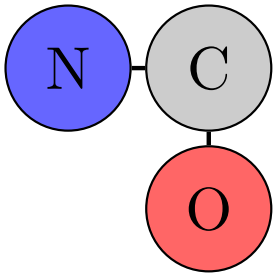


Exponential decay with depth

Chemical Implications of Oversmoothing

Shallow networks (1-3 layers):

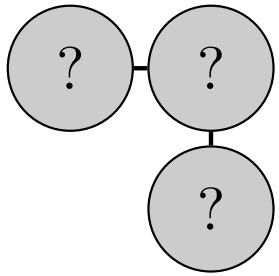
- Preserve local chemical environment
- Atoms retain distinct features
- Can differentiate functional groups
- Good for property prediction



Distinct atom features

Deep networks (5+ layers):

- Atoms become indistinguishable
- Loss of local chemical information
- Cannot identify reactive sites
- Poor discrimination between molecules



Oversmoothed!

The Solution

Skip connections (residual learning) allow deeper networks while preserving feature diversity. We'll cover this next!

Skip Connections: The Solution to Oversmoothing

Residual Learning for GNNs

Key Idea: Add skip/residual connections

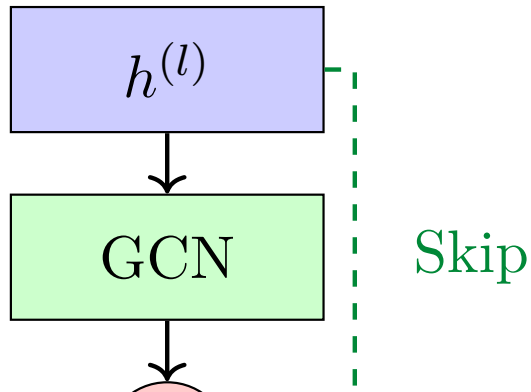
$$h^{(l+1)} = \text{GCN}(h^{(l)}) + h^{(l)}$$

Benefits:

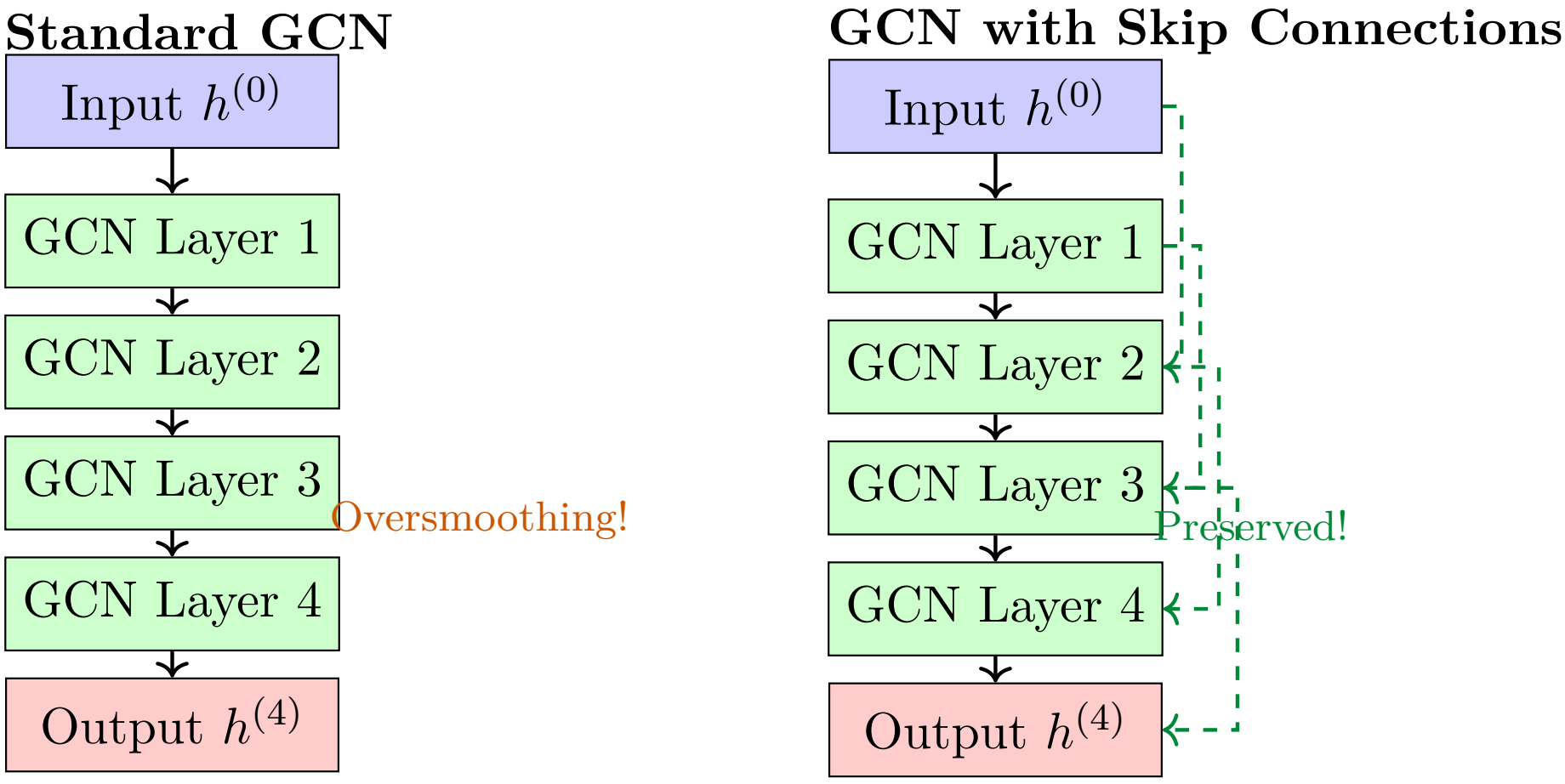
- Preserves information from earlier layers
- Mitigates oversmoothing
- Enables deeper architectures (5-10+ layers)
- Improves gradient flow during training

Intuition:

- Identity mapping preserves features
- Network learns **residual** (difference)
- Easier to optimize: can always keep identity
- Features stay diverse across layers



Architecture Comparison

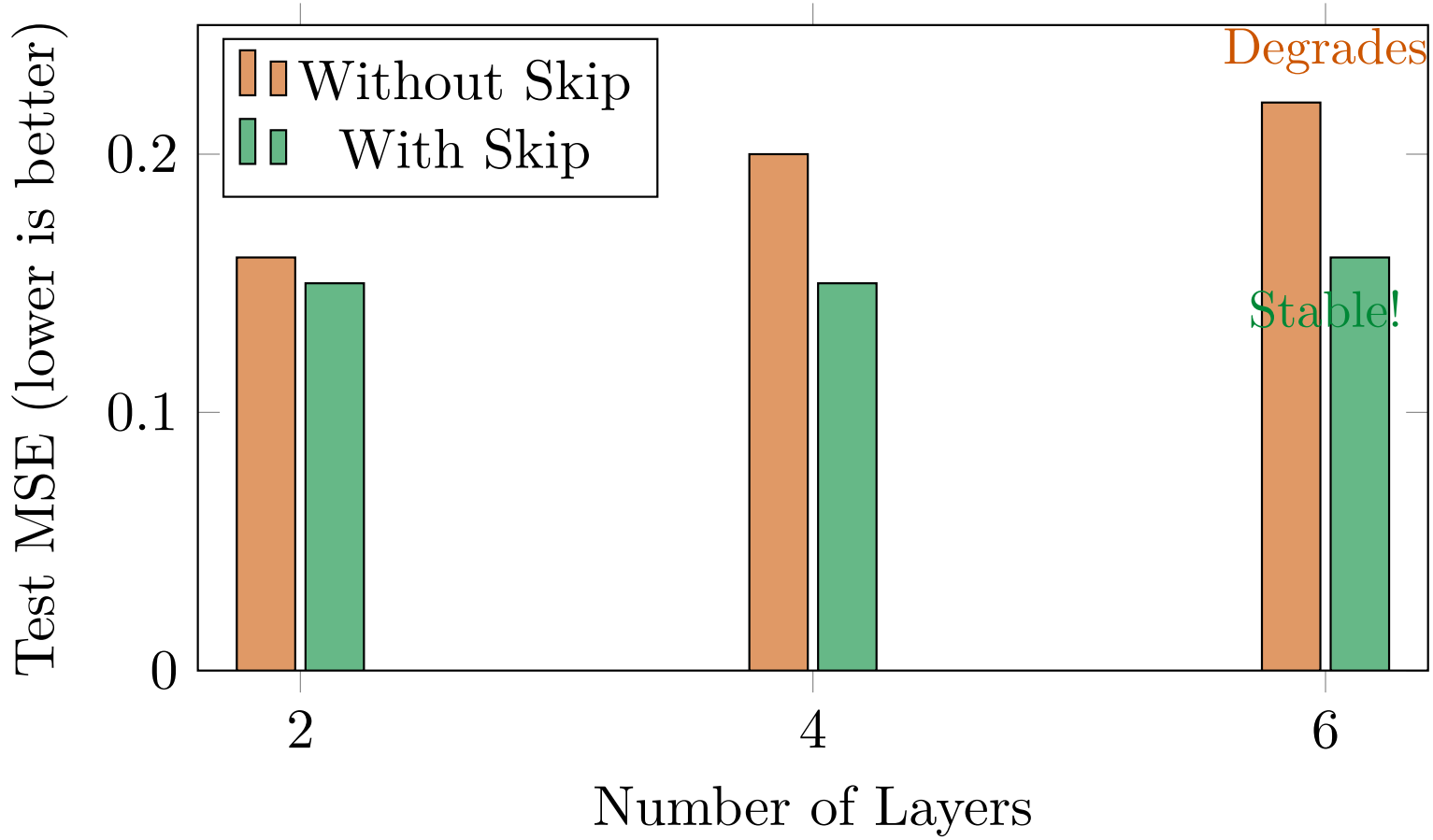


Key Difference

Skip connections allow information to bypass layers, maintaining feature diversity even in deep networks.

Experimental Impact: ESOL Dataset

How much do skip connections help?



Results

- At **2 layers**: Skip connections provide **6%** improvement
- At **4 layers**: Skip connections provide **25%** improvement

Implementation in PyTorch

Adding skip connections is straightforward:

```python
class GCNWithSkip(nn.Module):
    def __init__(self, in_channels,
                 hidden_channels,
                 out_channels, num_layers):
        super().__init__()
        self.convs = nn.ModuleList()

        # First layer
        self.convs.append(GCNConv(
            in_channels, hidden_channels))

        # Hidden layers
        for _ in range(num_layers - 2):
            self.convs.append(GCNConv(
                hidden_channels,
                hidden_channels))

        # Output layer
        self.lin = nn.Linear(
            hidden_channels, out_channels)
```

```python
    def forward(self, x,
                edge_index, batch
                    ):
        # First layer
        x = F.relu(
            self.convs[0](x,
                 edge_index))

        # Hidden layers with skip
        for conv in self.convs
              [1:]:
            identity = x  # Save
            x = F.relu(
                conv(x,
                     edge_index)
                     )
            x = x + identity  #
                  Skip!

        # Pooling and output
        x = global_mean_pool(x,
             batch)
        return self.lin(x)
```

Key Lines

`identity = x` → Save input; `x = x + identity` → Add skip connection

Best Practices and Extensions

When to use skip connections:

- Networks with **¿ 3 layers**
- When you need deeper representations
- Property prediction on large molecules

Combine with:

- Batch normalization
- Dropout (helps regularization)
- Layer normalization

Advanced variants:

- **Dense connections**: Skip from all previous layers
- **Jumping Knowledge (JK) Networks**:

$$h_{final} = \text{Agg}(h^{(0)}, h^{(1)}, \ldots, h^{(L)})$$

- **Highway connections**: Learn gating

$$h^{(l+1)} = g \cdot \text{GCN}(h^{(l)}) + (1-g) \cdot h^{(l)}$$

Recommendation

Start simple: Add skip connections to layers 2+. Monitor validation performance to find optimal depth.

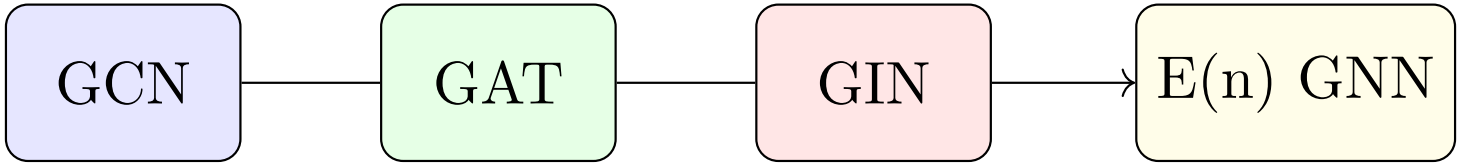
GNN Advanced Concepts

Advanced Architectures:

- GraphSAGE: Scalable inductive representation learning
- Graph Attention Networks (GAT): Attention-based message passing
- Graph Isomorphism Network (GIN): More powerful than GCN
- E(n) Equivariant GNNs: 3D-aware models

Advanced Techniques:

- Virtual nodes: Connect all nodes to improve long-range information flow
- Edge features: Include bond information
- Residual connections: Skip connections in message passing
- Graph normalization: Batch norm for graphs
- Pre-training: Self-supervised learning on molecules



Increasing expressive power

Resources and Further Reading

GitHub Repository:

- `https://github.com/HFooladi/GNNs-For-Chemists`

Tutorials:

- "A Gentle Introduction to Graph Neural Networks"
- "Understanding Convolutions on Graphs"
- PyTorch Geometric Documentation

Research Papers:

- Kipf & Welling, "Semi-Supervised Classification with Graph Convolutional Networks"
- Gilmer et al., "Neural Message Passing for Quantum Chemistry"
- Veličković et al., "Graph Attention Networks"
- Xu et al., "How Powerful are Graph Neural Networks?"

Questions? Let's discuss!