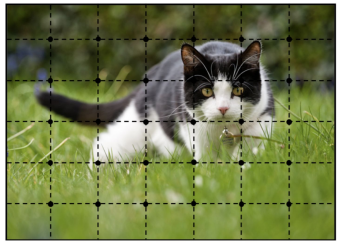
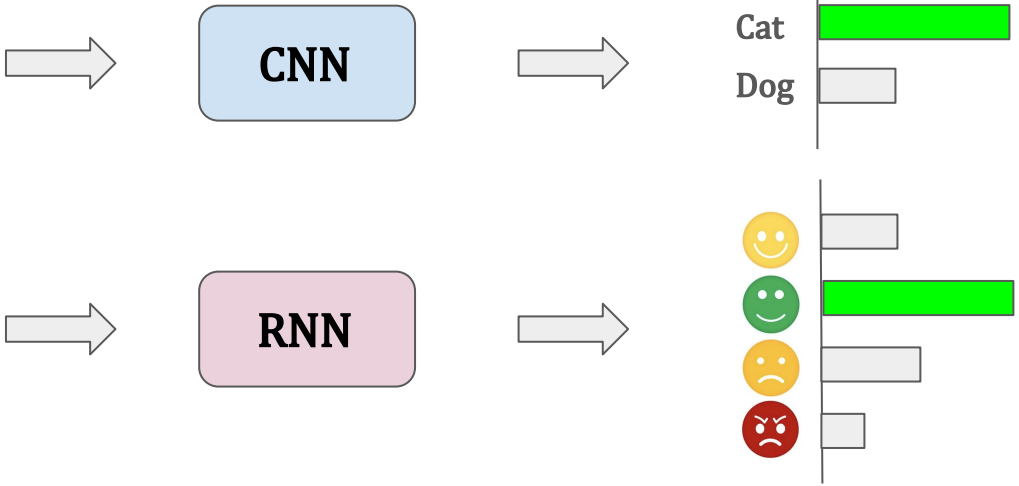
APS360: Applied Fundamentals of Deep Learning

Week 11: Graph Neural Networks

Motivation

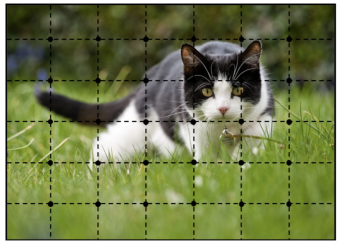
Euclidean



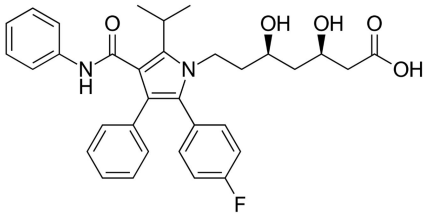


Motivation

Euclidean



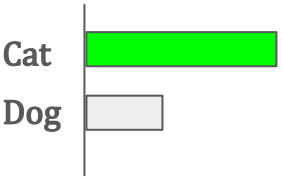
Non-Euclidean

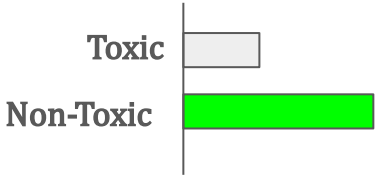


CNN

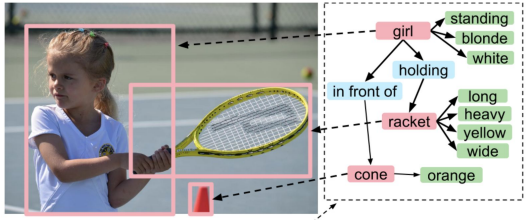
RNN

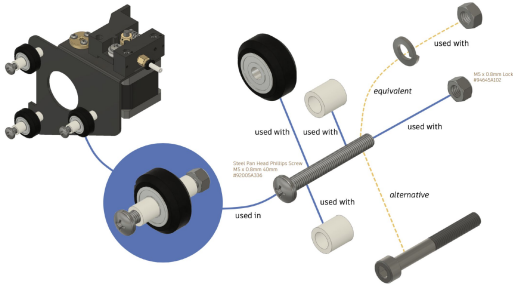
GNN

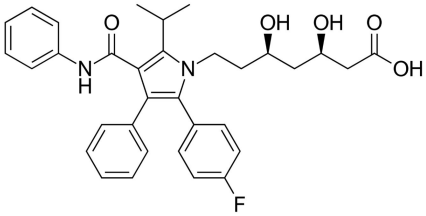


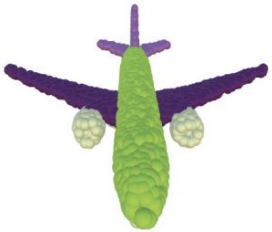


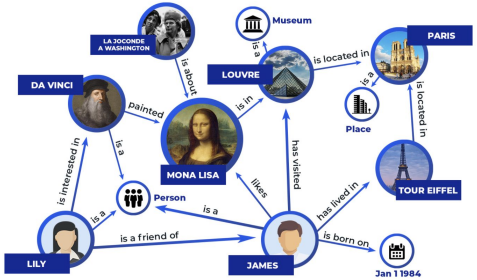
Graphs are everywhere!!!

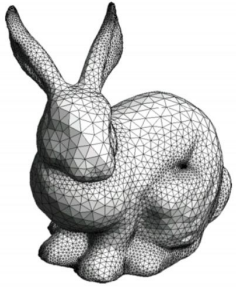


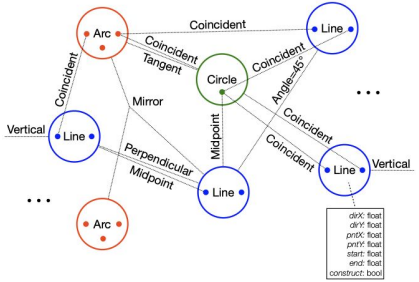


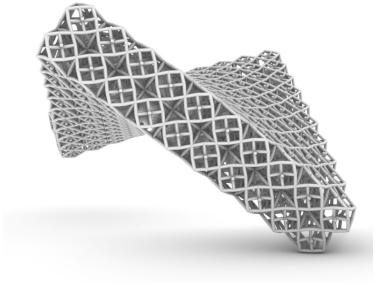










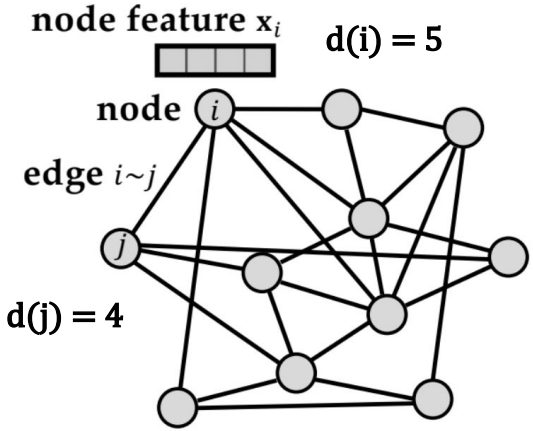


TORONTO

Graphs

A graph **G=(V, E, X)** is a data-structure that encodes **pair-wise interactions** or **relations** among **concepts** and **objects** :

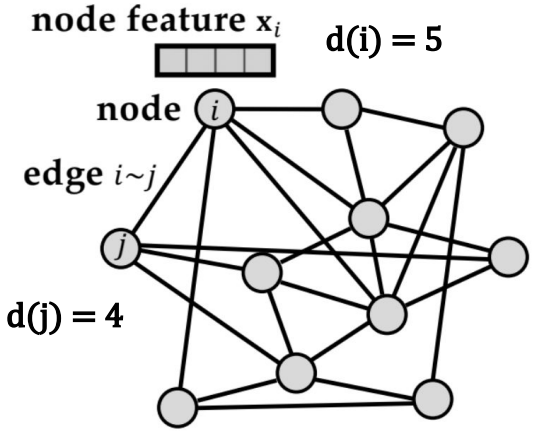
- **V** is set of nodes representing concepts or objects
- **E⊆VxV** is a set of edges connecting nodes and representing relations or interactions among them
- **X** encodes the node features of each node



Graphs

A graph **G=(V, E, X)** is a data-structure that encodes **pair-wise interactions** or **relations** among **concepts** and **objects** :

- **V** is set of nodes representing concepts or objects
- **E⊆VxV** is a set of edges connecting nodes and representing relations or interactions among them
- **X** encodes the node features of each node



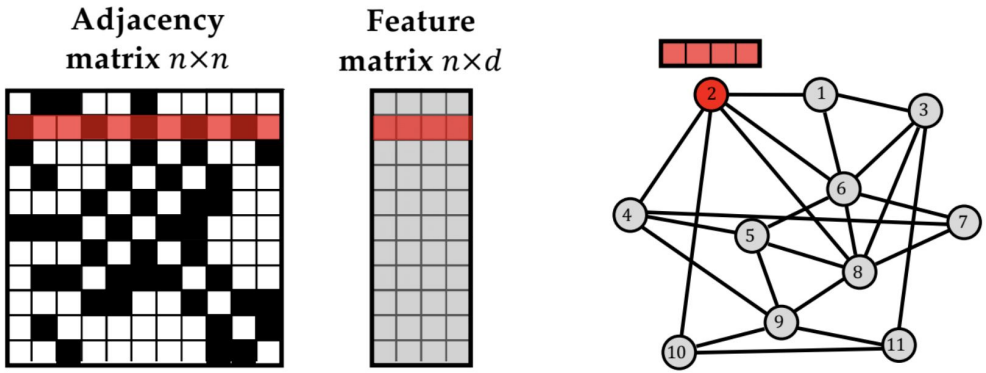
We can represent the edges in an **adjacency matrix A** :

$$a_{ij} = \begin{cases} 1, & (i,j) \in \mathcal{E} \\ 0, & (i,j) \notin \mathcal{E} \end{cases}$$

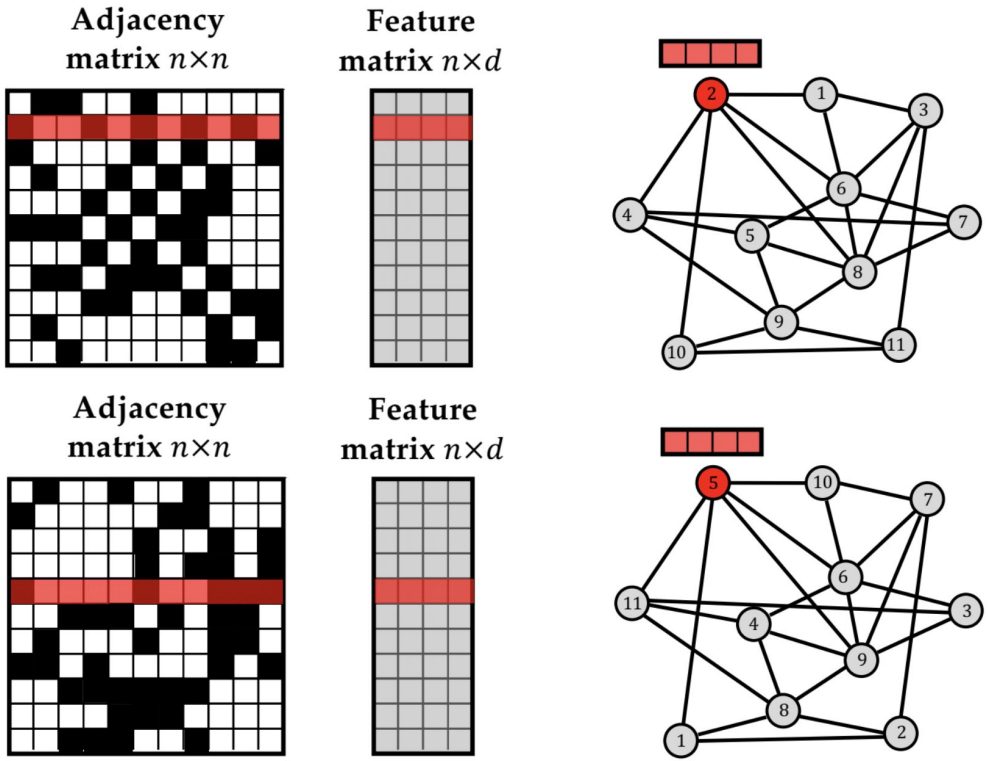
Degree of a node is number of edges connecting to that node

$$d(i) = \sum_j a_{ij}$$

Graphs are order-invariant



Graphs are order-invariant

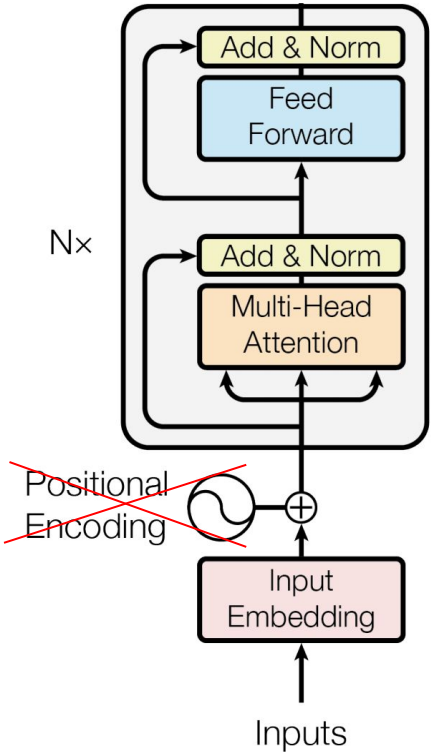


Same graph with arbitrary ordering of nodes

There are n! of such permutations

Transformers & Graphs

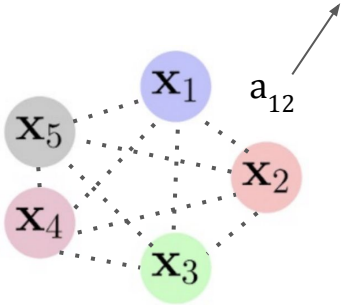
What happens if we omit the Positional Encoding from Transformers?

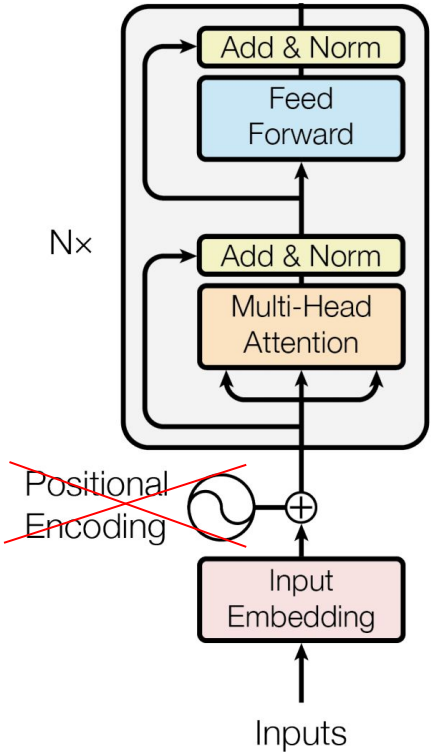


Graphs

- The transformer learns an NxN attention matrix which represent pairwise importance scores
- This means Transformer creates a fully-connected graph over the input and learns the edge weights

Attention score learned between X_1 and X_2

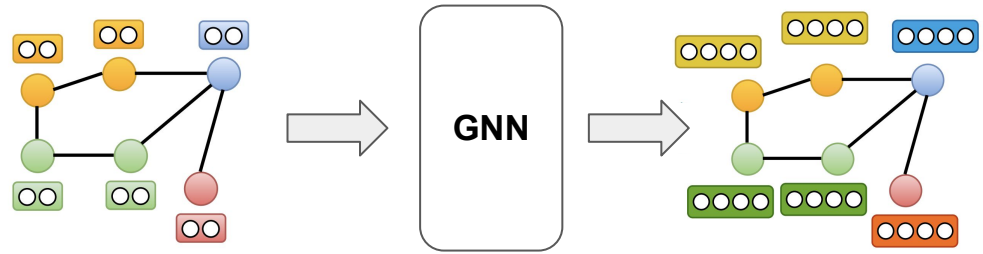




Graph Neural Networks (GNNs)

Message-Passing

GNNs learn embeddings over graphs through **Message-Passing**



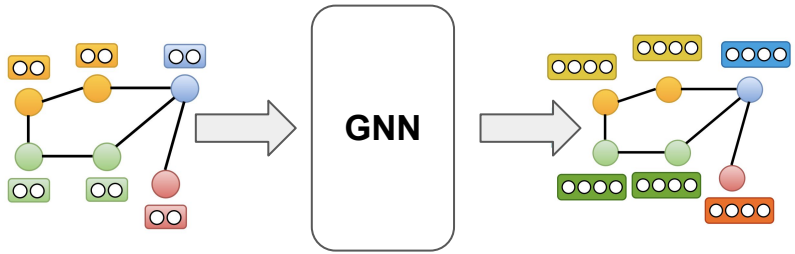
Message-Passing

For each node in graph:

1. **Aggregate** embeddings of its neighbor nodes
2. **Combine** the aggregated embedding with the node embedding
3. **Update** the node embedding

$$h_v^{(k)} = \text{COMBINE}\left(h_v^{(k-1)}, \text{AGGREGATE}\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)\right)$$

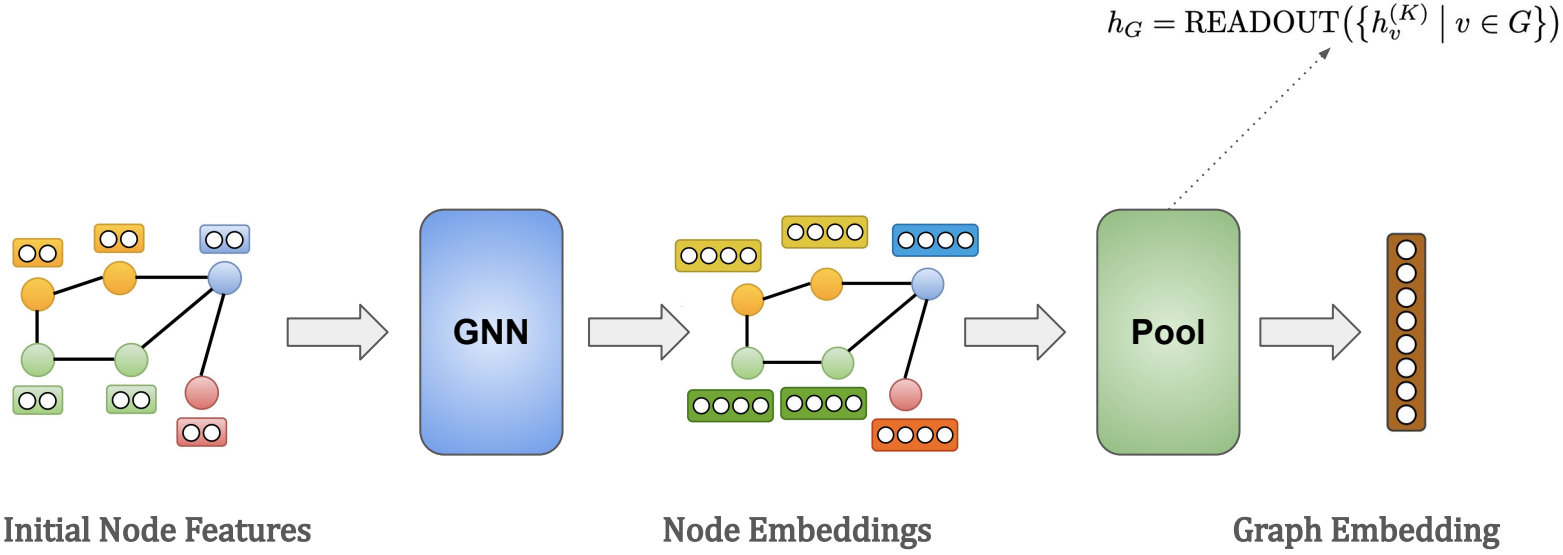
Aggregate function must be an order invariant function: sum, mean, max, attention, etc.



Read-out (graph pooling) function

Aggregates all node embeddings into a graph embedding

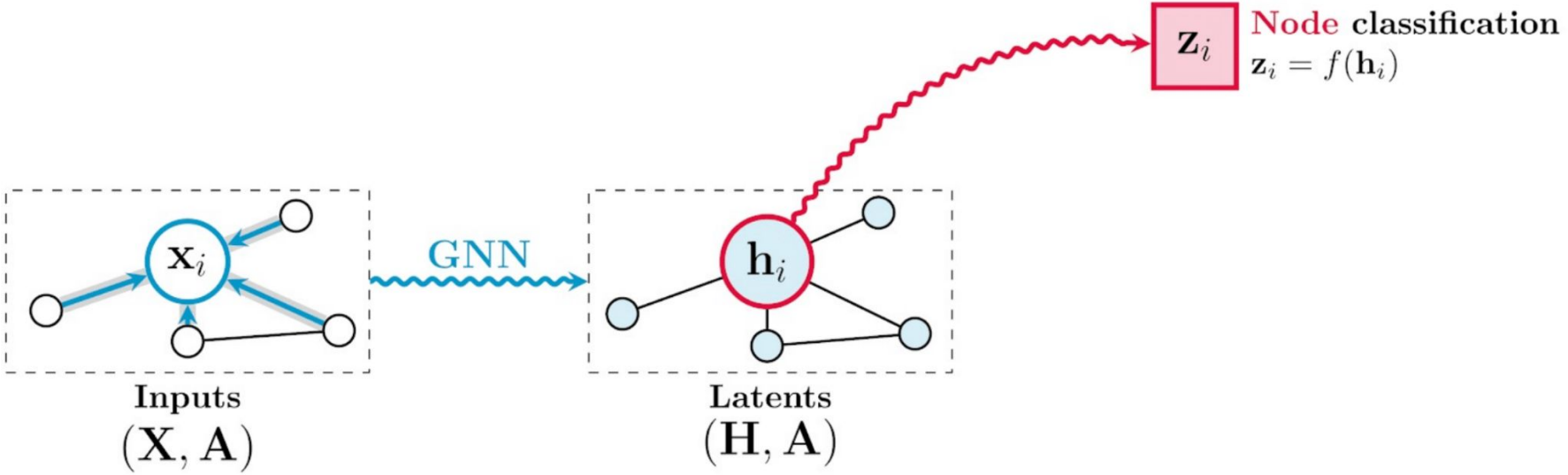
Must be an order invariant function such as sum, mean, max, attention, etc.



What GNNs can do?

We can use them to predict node classes

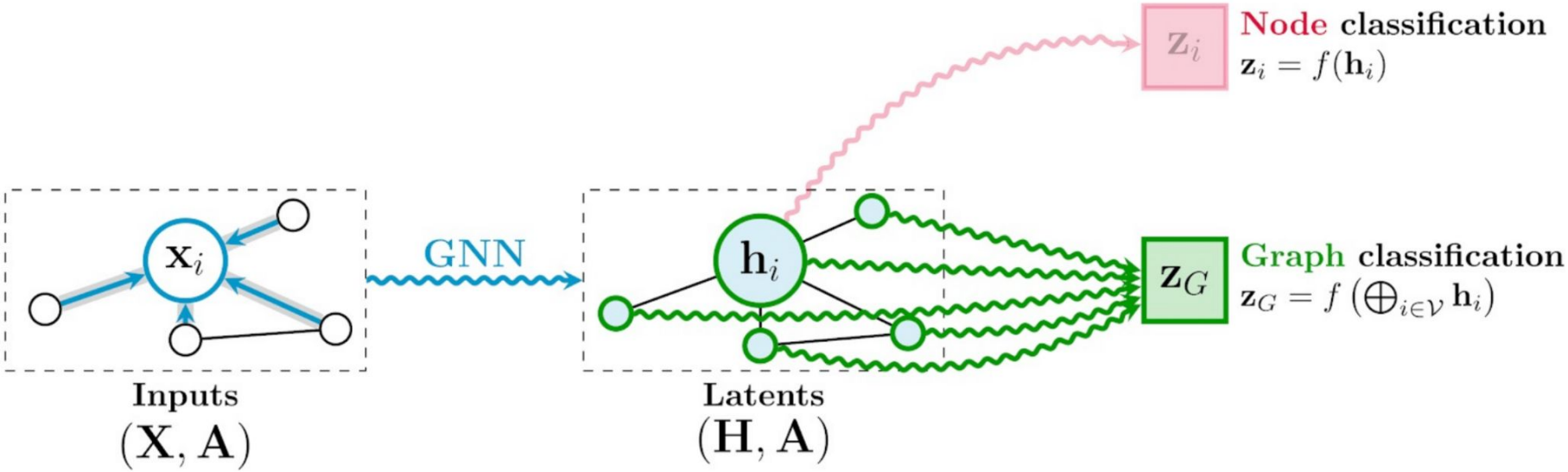
Example: Predicting Atom types in a molecule structure



What GNNs can do?

We can use them to predict graph classes

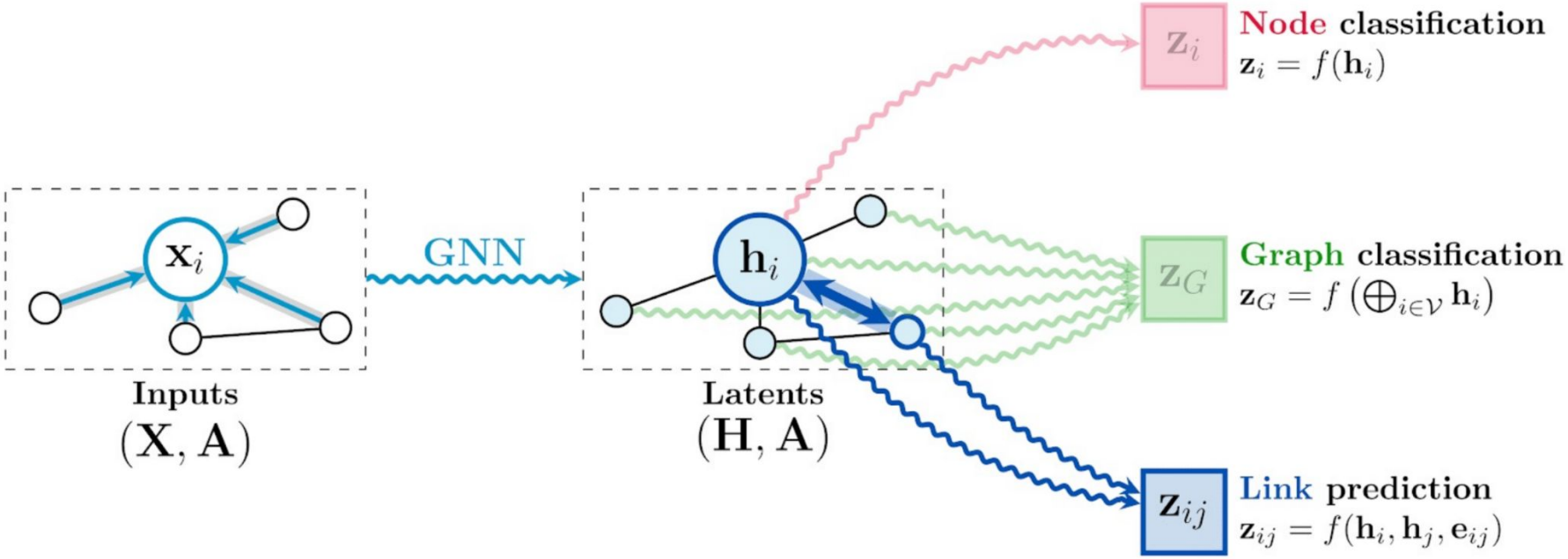
Example: Predicting if a molecule structure is toxic



What GNNs can do?

We can use them to predict links between nodes

Example: Predicting if there should be a bond between two atoms

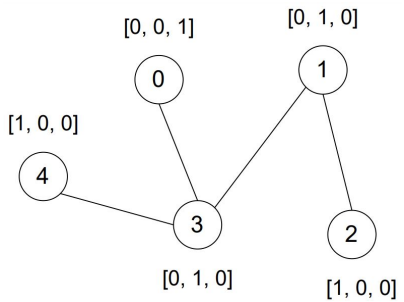


Graph Convolutional Networks (GCNs)

GCNs

A layer of a GNN is a nonlinear function over node features and adjacency matrix.

$$\mathbf{H} = \text{ReLU}\left(\mathbf{AXW} + b\right)$$



$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad X = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

This is already a strong model, but has two limitations.

GCNs

Limitation 1 : Multiplication with A means that, for every node, we sum up all the feature vectors of all neighboring nodes but not the node itself.

Fix: Add self-loops (add the identity matrix to A)

$$\mathbf{A} = \mathbf{A} + \mathbf{I}$$

GCNs

Limitation 1 : Multiplication with A means that, for every node, we sum up all the feature vectors of all neighboring nodes but not the node itself.

Fix: Add self-loops (add the identity matrix to A)

$$\mathbf{A} = \mathbf{A} + \mathbf{I}$$

Limitation 2 : A is not normalized and therefore the multiplication with A will completely change the scale of the feature vectors

Fix: Symmetrically normalize A using diagonal degree matrix D such that all rows sum to one.

$$\mathbf{A} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$$

GCNs

Limitation 1 : Multiplication with A means that, for every node, we sum up all the feature vectors of all neighboring nodes but not the node itself.

Fix: Add self-loops (add the identity matrix to A)

$$\mathbf{A} = \mathbf{A} + \mathbf{I}$$

Limitation 2 : A is not normalized and therefore the multiplication with A will completely change the scale of the feature vectors

Fix: Symmetrically normalize A using diagonal degree matrix D such that all rows sum to one.

$$\mathbf{A} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$$

With these fixes, we define a **GCN layer** as follows:

$$\mathbf{H} = \mathrm{ReLU}\left(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\mathbf{X}\mathbf{W} + b\right)$$

Going deeper with GNNs

A GCN layer updates the node embeddings based on the features of the immediate neighbors (recall multiplication with A)

We can influence the embeddings from further neighborhood by stacking GCN layers

This is analogous to increasing the receptive field in CNNS

$$\mathbf{H}^{(0)} = \mathbf{X}$$

$$\mathbf{H}^{(1)} = \mathrm{ReLU}\left(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\mathbf{H}^{(0)}\mathbf{W}^{(1)} + b^{(1)}\right)$$

$$\mathbf{H}^{(2)} = \mathrm{ReLU}\left(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\mathbf{H}^{(1)}\mathbf{W}^{(2)} + b^{(2)}\right)$$

$$\cdots$$

$$\mathbf{H}^{(l)} = \mathrm{ReLU}\left(\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\mathbf{H}^{(l-1)}\mathbf{W}^{(l)} + b^{(l)}\right)$$

Node embeddings in layer 2 are computed based on contributions from 1-hop and 2-hop neighbors

Graph Attention Networks (GAT)

Idea: Instead of using node degree, learn an attention score between two nodes, i.e., learn the contribution weight of neighbor nodes

Graph Attention Networks (GAT)

Idea: Instead of using node degree, learn an attention score between two nodes, i.e., learn the contribution weight of neighbor nodes

1. Use a shared neural network to compute an attention score between two nodes.

$$e_{ij} = \mathrm{NN}(h_i, h_j)$$

Graph Attention Networks (GAT)

Idea: Instead of using node degree, learn an attention score between two nodes, i.e., learn the contribution weight of neighbor nodes

1. Use a shared neural network to compute an attention score between two nodes.

$$e_{ij} = \mathrm{NN}(h_i, h_j)$$

2. Normalize the attention scores

$$\alpha_{ij} = \mathrm{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})}$$

Graph Attention Networks (GAT)

Idea: Instead of using node degree, learn an attention score between two nodes, i.e., learn the contribution weight of neighbor nodes

1. Use a shared neural network to compute an attention score between two nodes.

$$e_{ij} = \text{NN}(h_i, h_j)$$

2. Normalize the attention scores

$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})}$$

3. Update the node embeddings based on the attention score

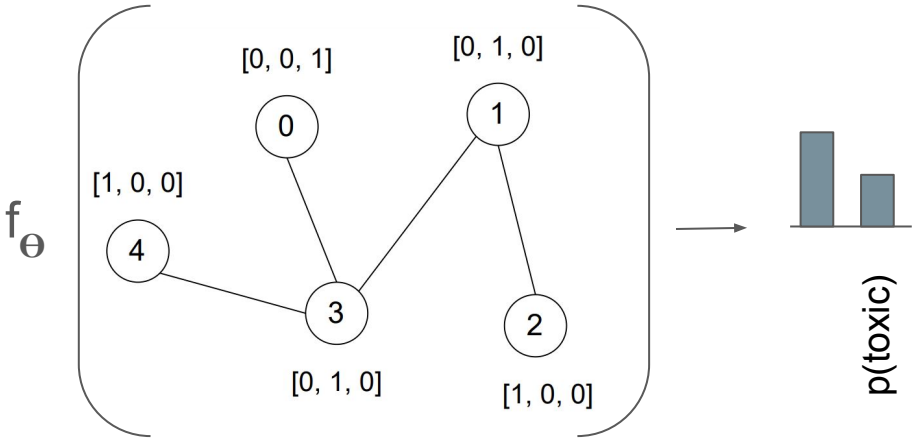
$$h_i = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W} h_j\right)$$

PyTorch Implementation

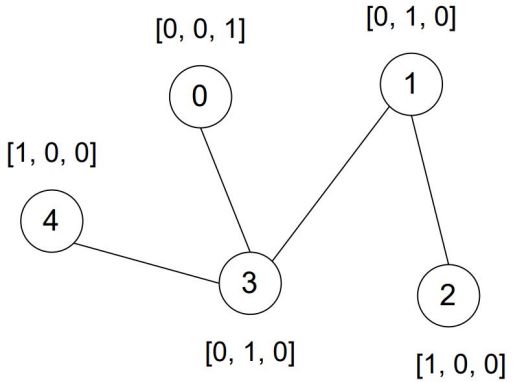
p(non-toxic)

Implementation

Suppose we want to classify the following molecule using a GCN to a toxic/non-toxic where node features represent the atom type (Carbon, Hydrogen, Nitrogen)

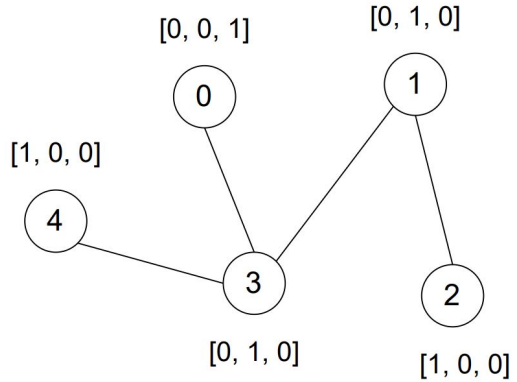


Dense Implementation



$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad X = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Dense Implementation



$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad X = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

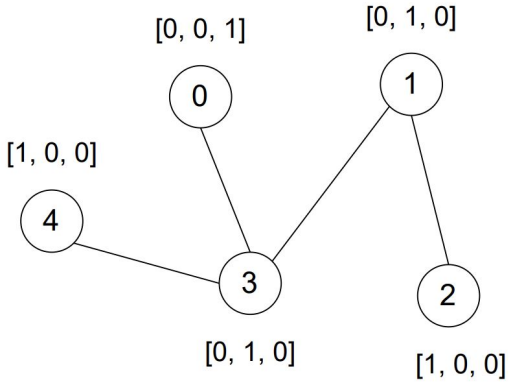
Normalize Adjacency

$$D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$$

```
class DenseGCN(nn.Module):
    def __init__(self):
        super(DenseGNN, self).__init__()
        self.fc1 = nn.Linear(3, 32)
        self.fc2 = nn.Linear(32, 64)
        self.fc3 = nn.Linear(64, 1)

    def forward(self, A, x):
        A = A + torch.eye(A.shape[0])
        D = torch.diag(torch.sum(A, dim=0))
        D = torch.pow(D, -0.5)
        A = torch.mm(torch.mm(D, A), D)
        x = torch.mm(A, x)
        x = F.relu(self.fc1(x))
        x = torch.mm(A, x)
        x = F.relu(self.fc2(x))
        g = torch.sum(x, dim=0)
        y = torch.Sigmoid(self.fc3(g))
        return y
```

Dense Implementation



$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad X = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Node embeddings

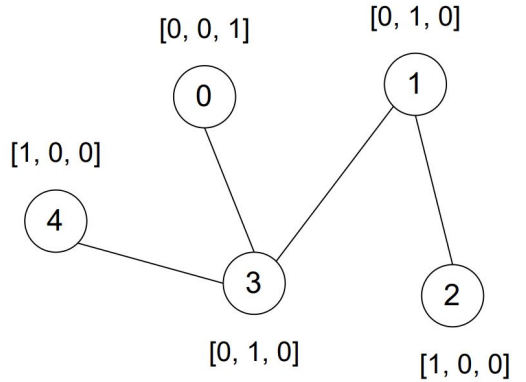
Graph embedding

Graph-level prediction

```
class DenseGCN(nn.Module):
    def __init__(self):
        super(DenseGNN, self).__init__()
        self.fc1 = nn.Linear(3, 32)
        self.fc2 = nn.Linear(32, 64)
        self.fc3 = nn.Linear(64, 1)

    def forward(self, A, x):
        A = A + torch.eye(A.shape[0])
        D = torch.diag(torch.sum(A, dim=0))
        D = torch.pow(D, -0.5)
        A = torch.mm(torch.mm(D, A), D)
        x = torch.mm(A, x)
        x = F.relu(self.fc1(x))
        x = torch.mm(A, x)
        x = F.relu(self.fc2(x))
        g = torch.sum(x, dim=0)
        y = torch.Sigmoid(self.fc3(g))
        return y
```

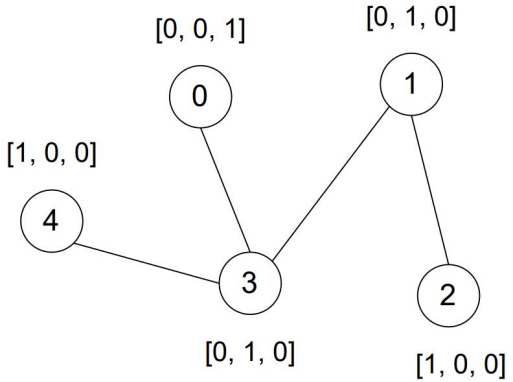
Dense Implementation



$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad X = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

- Most graphs are sparse
- Dense implementation uses N^2 space for adjacency
- We can represent this graph with only 4 edges where dense implementation represents it with 25
- Implementing sparse operations are supported by PyTorch but are not very straightforward
- We can use **PyTorch Geometric (PYG)**

Sparse Implementation



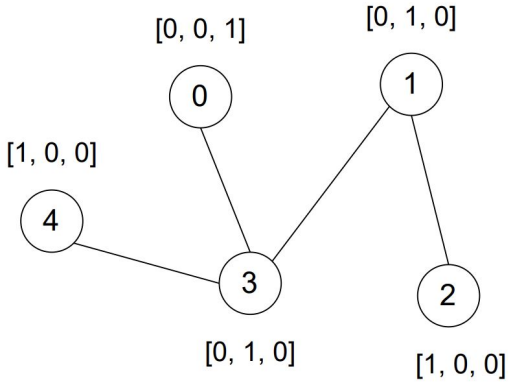
$$A = \begin{bmatrix} 0 & 1 & 1 & 2 & 3 & 3 & 3 & 4 \\ 3 & 2 & 3 & 1 & 0 & 1 & 4 & 3 \end{bmatrix} \quad X = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

```
import torch
from torch_geometric.data import Data

edge_index = torch.tensor([[0, 1, 1, 2, 3, 3, 3, 4],
                           [3, 2, 3, 1, 0, 1, 4, 3]],
                          dtype=torch.long)
x = torch.tensor([[0, 0, 1],
                  [0, 1, 0],
                  [1, 0, 0],
                  [0, 1, 0],
                  [1, 0, 0]], dtype=torch.float)

data = Data(x=x, edge_index=edge_index, y=[1.0])
```

Sparse Implementation



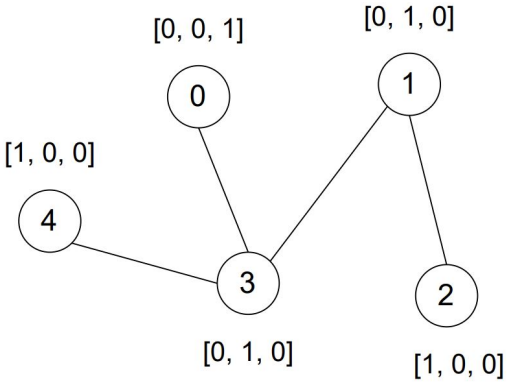
$$A = \begin{bmatrix} 0 & 1 & 1 & 2 & 3 & 3 & 3 & 4 \\ 3 & 2 & 3 & 1 & 0 & 1 & 4 & 3 \end{bmatrix} \quad X = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

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

class SparseGCN(nn.Module):
    def __init__(self):
        super(SparseGCN, self).__init__()
        self.gcn1 = nn.GCNConv(3, 32)
        self.gcn2 = nn.GCNConv(32, 64)
        self.fc = nn.Linear(64, 1)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = F.relu(self.gcn1(x, edge_index))
        x = F.relu(self.gcn2(x, edge_index))
        g = torch.sum(x, dim=0)
        y = torch.Sigmoid(self.fc(g))
        return y
```

Sparse Implementation



$$A = \begin{bmatrix} 0 & 1 & 1 & 2 & 3 & 3 & 3 & 4 \\ 3 & 2 & 3 & 1 & 0 & 1 & 4 & 3 \end{bmatrix} \quad X = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

We can replace it with GATConv.

```
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv

class SparseGCN(nn.Module):
    def __init__(self):
        super(SparseGCN, self).__init__()
        self.gcn1 = nn.GCNConv(3, 32)
        self.gcn2 = nn.GCNConv(32, 64)
        self.fc = nn.Linear(64, 1)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = F.relu(self.gcn1(x, edge_index))
        x = F.relu(self.gcn2(x, edge_index))
        g = torch.sum(x, dim=0)
        y = torch.Sigmoid(self.fc(g))
        return y
```

DataLoader & Dataset

Dense implementation: batching is done by creating a diagonal matrix of adjacency matrices

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & & \\ & \ddots & \\ & & \mathbf{A}_n \end{bmatrix}, \qquad \mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_n \end{bmatrix}, \qquad \mathbf{Y} = \begin{bmatrix} \mathbf{Y}_1 \\ \vdots \\ \mathbf{Y}_n \end{bmatrix}$$

Sparse implementation: uses an index vector which maps each node to its respective graph in the batch

$$\text{batch} = [0 \quad \cdots \quad 0 \quad 1 \quad \cdots \quad n-2 \quad n-1 \quad \cdots \quad n-1]^\top$$

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 2 \\ 2 \end{bmatrix}$$

Nodes 0-2 → Graph 0

Nodes 3-6 → Graph 1

Nodes 7-8 → Graph 2

DataLoader & Dataset

```
from torch_geometric.datasets import TUDataset
from torch_geometric.loader import DataLoader

dataset = TUDataset(root='/tmp/ENZYMES', name='ENZYMES')
loader = DataLoader(dataset, batch_size=32, shuffle=True)

for data in loader:
    print(data)
    break

>>> DataBatch(batch=[1082], edge_index=[2, 4066], x=[1082, 21], y=[32])
```

#Graphs in batch

Node → Graph mapping

#Edges in batch

#Nodes in batch

Node feature dimension

Sparse Implementation

Now that we have a batch of graphs, we need to only sum up node embeddings corresponding to each graph

We can use PYG functions that accept the node embeddings and node-graph mapping

```
from torch_geometric.nn import GCNConv, global_add_pool

class SparseGCN(nn.Module):
    def __init__(self):
        super(SparseGCN, self).__init__()
        self.gcn1 = nn.GCNConv(3, 32)
        self.gcn2 = nn.GCNConv(32, 64)
        self.fc = nn.Linear(64, 1)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = F.relu(self.gcn1(data.x, data.edge_index))
        x = F.relu(self.gcn2(x, data.edge_index))
        g = global_add_pool(x, data.batch)
        y = torch.Sigmoid(self.fc(g))
        return y
```

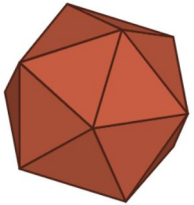
Training

```
from torch_geometric.datasets import TUDataset
from torch_geometric.loader import DataLoader

dataset = TUDataset(root='/tmp/ENZYMES', name='ENZYMES')
loader = DataLoader(dataset, batch_size=32, shuffle=True)

model = SparseGCN()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
criterion = nn.BCELoss()

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



PyTorch geometric

https://pytorch-geometric.readthedocs.io

Questions?