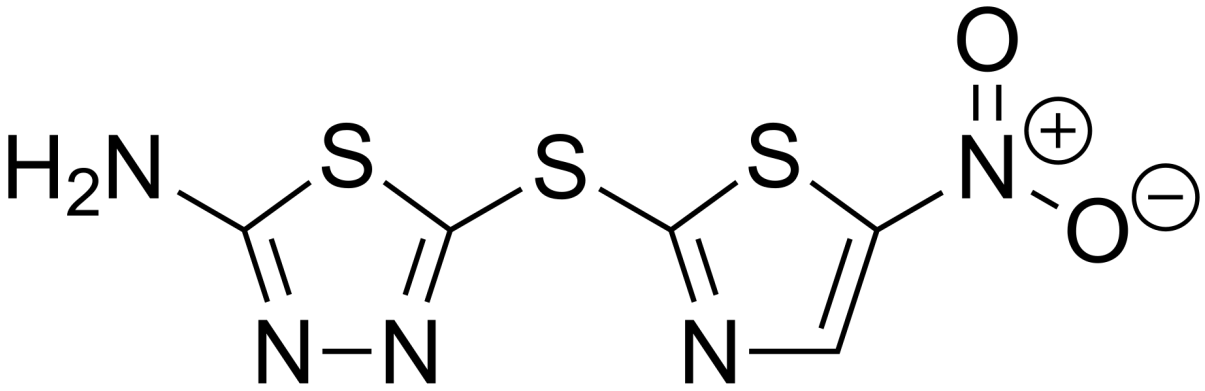
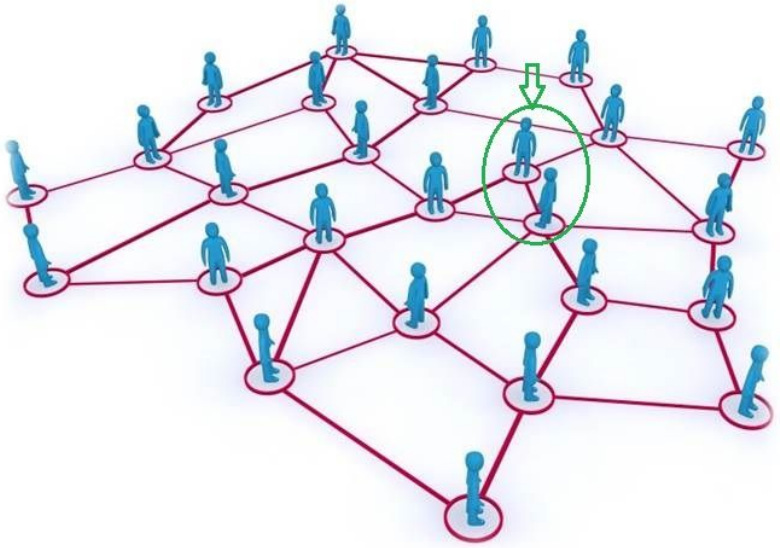
# Introduction to Graph Neural Networks

TAML 2025

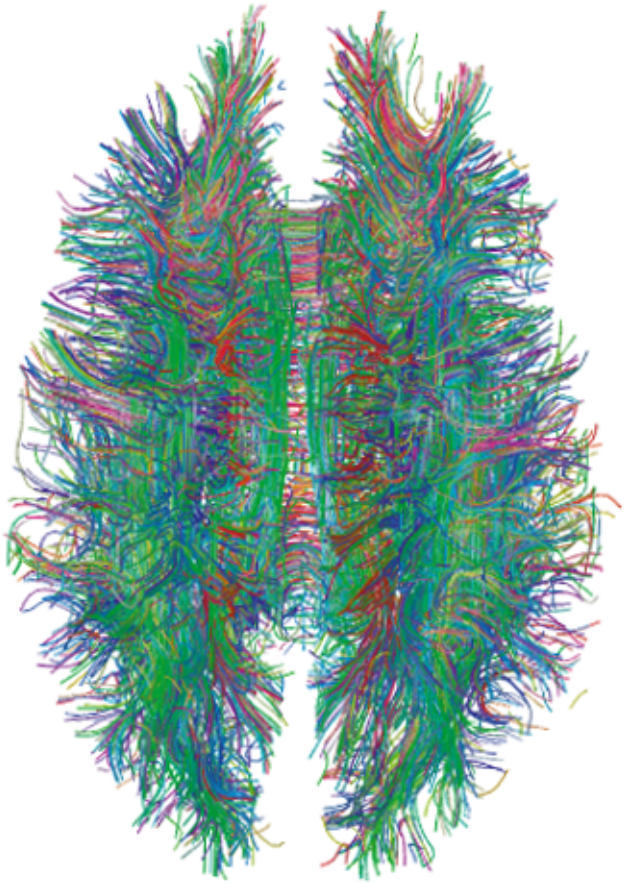
# Where can we see graphs?




Halicin (SU-3327) – antibiotic



Social networks recommendations

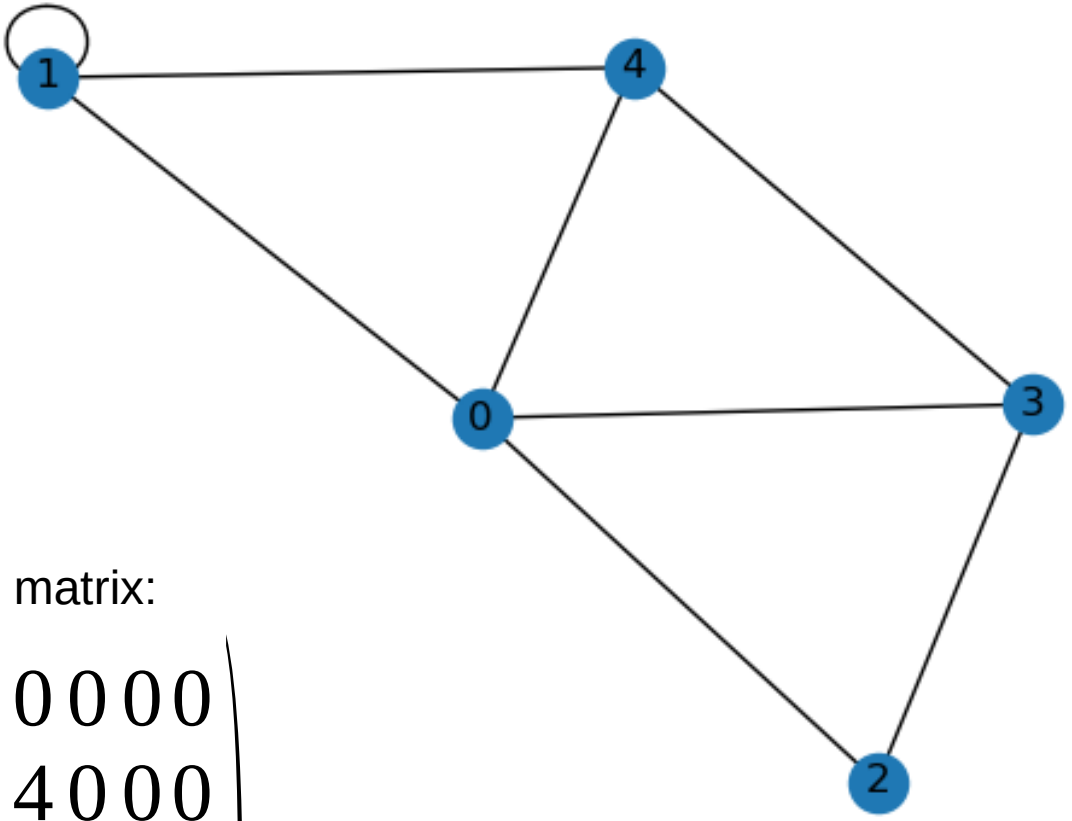


Connectome

# How to describe graphs?

Adjacency matrix:

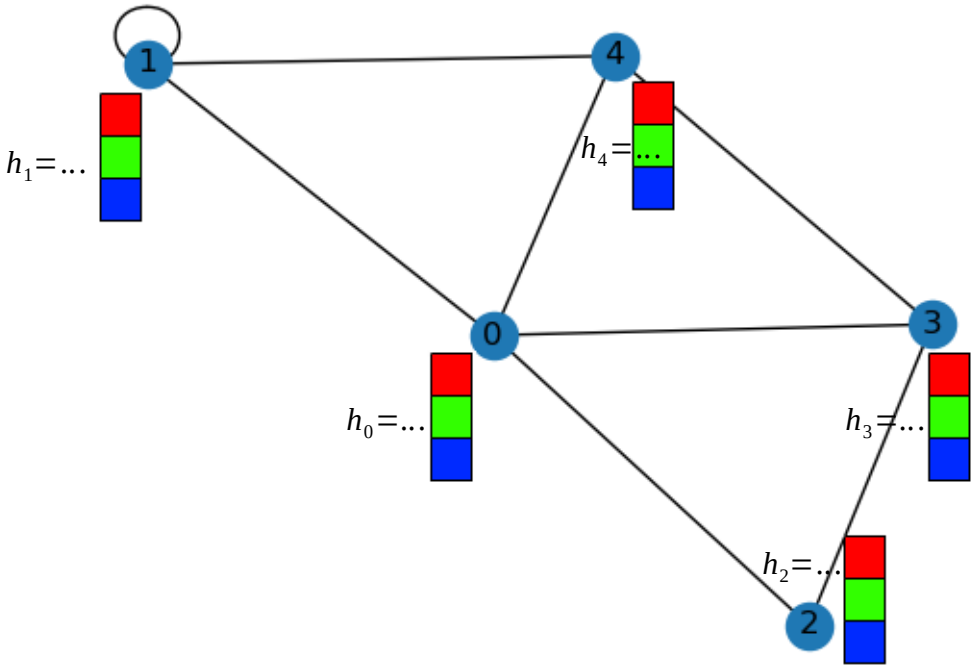
$$A = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$




Degree matrix:

$$D = \begin{pmatrix} 4 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 3 \end{pmatrix}$$

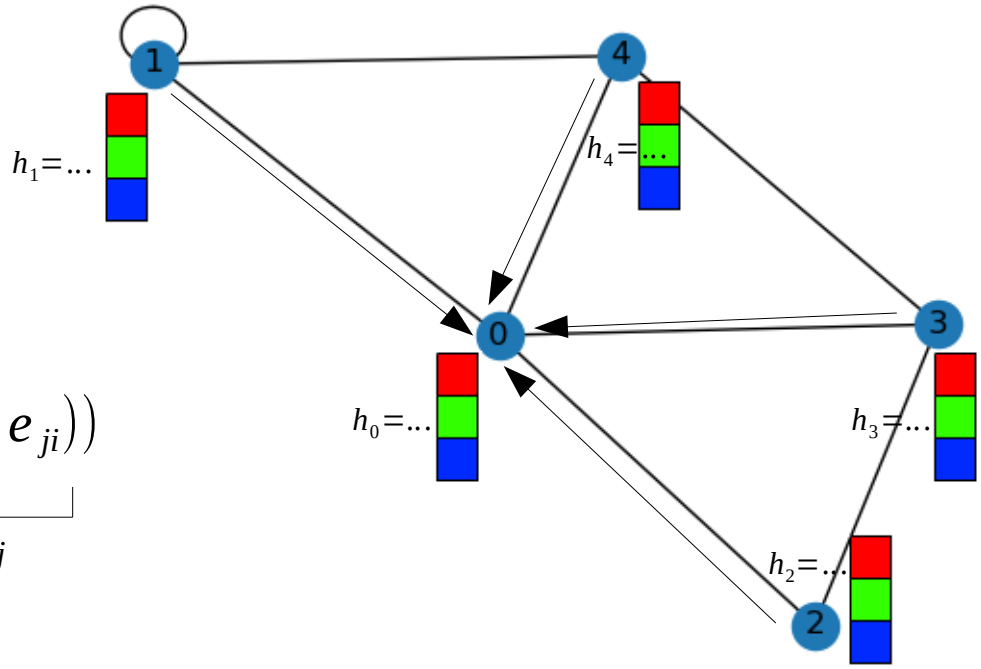
# Input data for GNN




Let's say we have an unweighted and undirected graph, each vertex has embedding (atom type, neuron activation count, most liked music genres, colour channels, etc...)

# Message passing

$$\overrightarrow{h_0^{new}} = f_v\left(\vec{h}_0, \underbrace{\frac{1}{N}\sum_{i \in neibhours}}_{\text{aggregation}} \underbrace{f_e\left(\vec{h}_0, \vec{h}_i, e_{ji}\right)}_{m_{0j}}\right)$$



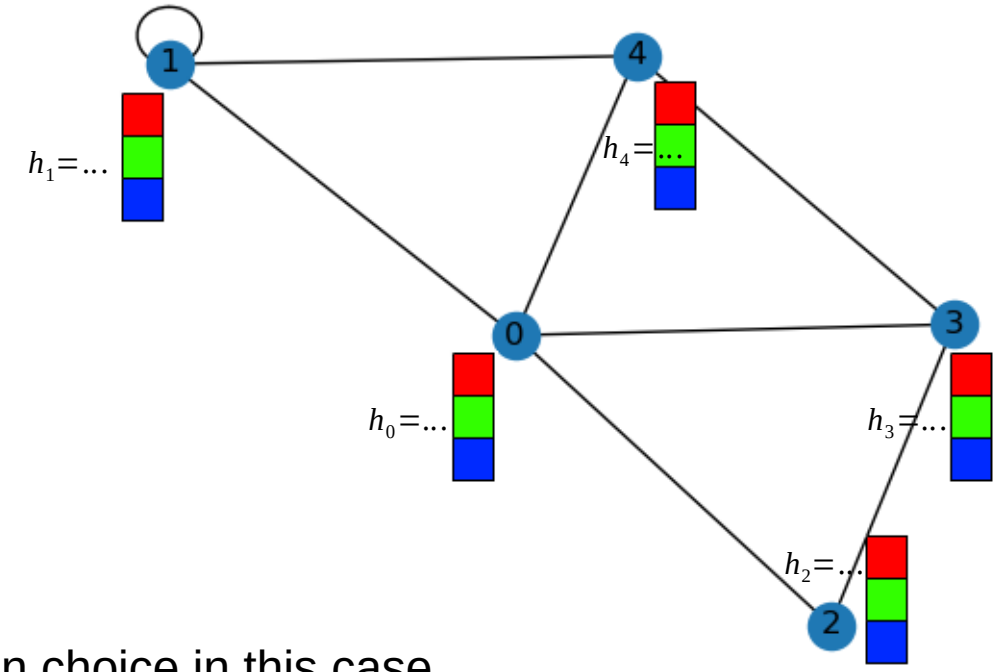

Where N is normalization for considering different number of edges:

There are a few choices of normalization.

$f_v, f_e$ – trainable functions (small fully connected neural networks)

# GCNConv

$$\overrightarrow{h_j^{new}} = f_v\left(\vec{h}_j, \frac{1}{N}\sum_{i \in neibhours} f_e\left(\vec{h}_j, \vec{h}_i, e_{ji}\right)\right)$$




Matrix form example for GCNConv layer:

$$H^{new} = (D+1)^{1/2} \cdot (A+1) \cdot (D+1)^{1/2} \cdot H \cdot W$$

$N = \sqrt{(deg(i)+1) \cdot (deg(j)+1)}$ – normalization choice in this case

$$f_e(h_j, h_i, e_{ji}) = e_{ji} h_i$$

$f_v(h_j, message_j) = W^T \cdot message_j$ – W is trainable

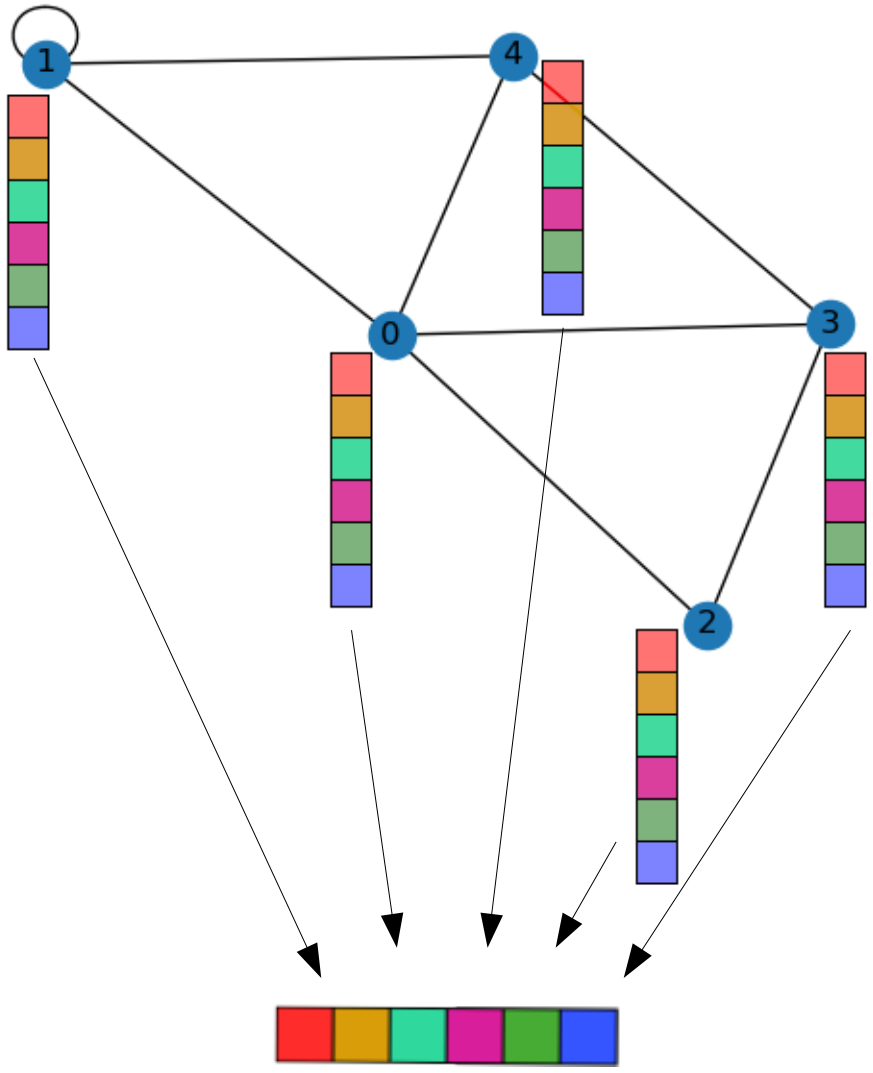
arxiv.org/abs/1609.02907 – Semi-Supervised Classification with Graph Convolutional Networks

https://pytorch-geometric.readthedocs.io/en/2.6.0/generated/torch_geometric.nn.conv.GCNConv.html

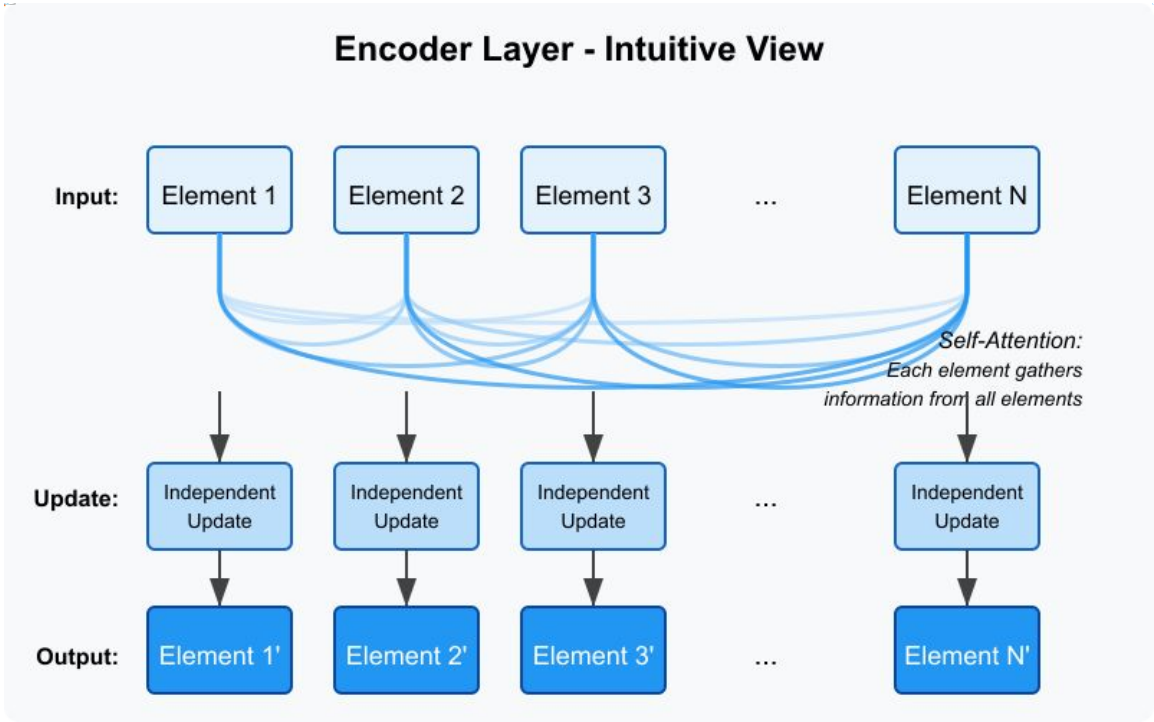
# Global pooling (depends on task)

No need for marking each node (for example node classification)

After performing the convolutions, we need to pool the graph data for postprocessing



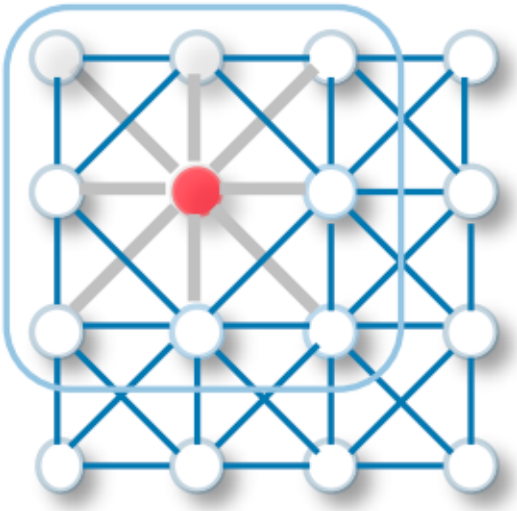
# Connection with Transformers



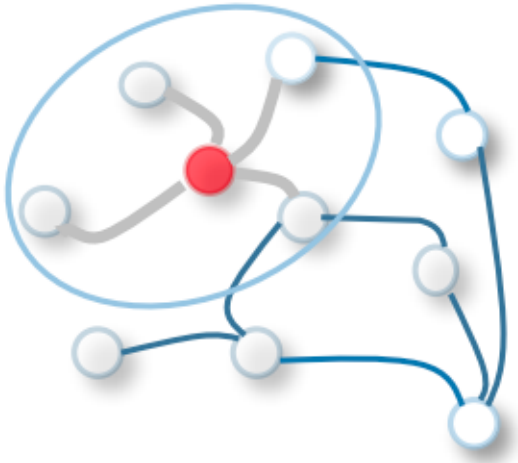

Transformer could be considered as a special case of GNN with some features:
- each vertex is connected with all (fully connected graph)
- special message passing protocol (attention)
- necessary global pooling for making output for entire sequence (depends on task)

See Ivan Kharuk's talk

# Connection with CNN



CNN



GNN

CNN could be considered as a special case of GNN with some features:
- each vertex connected with 8 neighbours (for Conv2D with kernel_size=3)
- special message passing protocol (similar with GCNConv but different message weight for each neighbour)
- after each layer we delete vertexes on the boundaries
- usually final pooling doesn’t aggregate whole 2D lattice, just reshaping for fully connected network

# PyTorch Geometric

# Input data preparation

Note: before global pooling we can work with batch of graphs like one graph with special adjacency matrix:

$$A_{batch} = \begin{pmatrix} A_1 & 0 & 0 & 0 \\ 0 & A_2 & 0 & 0 \\ 0 & 0 & A_3 & 0 \\ 0 & 0 & 0 & \dots \end{pmatrix}$$

Don’t worry about RAM if you use sparse torch tensors

https://pytorch-geometric.readthedocs.io/en/2.6.0/get_started/introduction.html#mini-batches

# Input data preparation

- Inputs:

  data: (|V|, len(embedding))

  edges: (2, |E|)

  batch_indexes: 1D array length |V|

# Message passing in code

$$\vec{h}_j^{new} = f_v\left(\vec{h}_j, \frac{1}{N}\sum_{i \in neibhours} f_e\left(\vec{h}_j, \vec{h}_i, e_{ji}\right)\right)$$

```
class MyGraphLayer(MessagePassing):
    def __init__(self, in_channels, hidden_channels, out_channels, aggr = 'mean'):
        super(MyGraphLayer, self).__init__(aggr=aggr)
        self.in_channels = in_channels
        self.hidden_channels = hidden_channels
        self.out_channels = out_channels
        self.messageEncoder = nn.Sequential(nn.Linear(2*in_channels,hidden_channels),
                                            nn.LeakyReLU(),
                                            nn.Linear(hidden_channels,out_channels),
                                            nn.LeakyReLU())
        self.newVertexEncoder = nn.Linear(in_channels+out_channels,out_channels)
    def forward(self, x, edge_index):
        return self.propagate(edge_index, x=x)
    def message(self, x_i, x_j):
        # x_i : "cental" node; x_j : neighbours
        # Make source-target dependent message
        msg = torch.cat([x_i, x_j], dim=-1) # concatenate embeddings like EdgeConv
        msg = self.messageEncoder(msg)
        return msg
    def update(self, aggr_out, x):
        embedding = torch.cat([aggr_out, x], dim=-1) # How to mix information from current vertex and neibhours?
        embedding = self.newVertexEncoder(embedding) # Let nn.Linear think about it
        return embedding
```

# Thank you for your attention!

Resources:
https://youtu.be/-UjytpbqX4A – Graph Neural Networks using Pytorch Geometric by Stanford
https://youtu.be/8owQBFAHw7E – Intro to graph neural networks (ML Tech Talks)
https://pytorch-geometric.readthedocs.io/ – Documentation on PyTorch Geometric