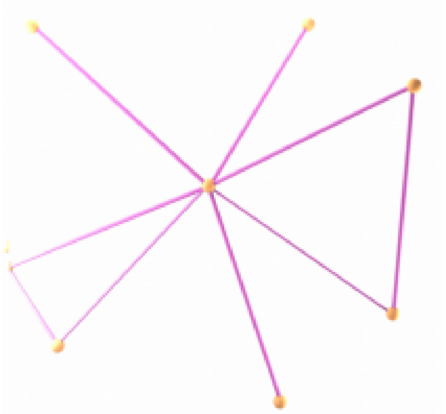


Graph Neural Networks

Mustafa Hajij

Outline

- Introduction to graphs and where to find them
- Machine learning tasks on graphs
- Introduction to graph neural networks (GNNs)
- The computational graph of a GNN
- Convolutional Graph Neural Networks
- Introduction to Pytorch Geometric
- Node Classification with Pytroch Geometric
- Graph Attention Networks.
- The over smoothing problem
- Graph Representation learning
- Graph Classification with Pytorch Geometric

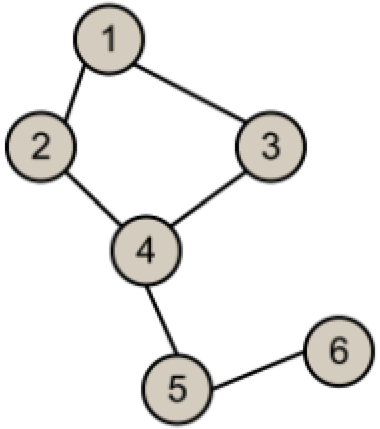
Graphs

A **graph** is an ordered pair (V,E) where,

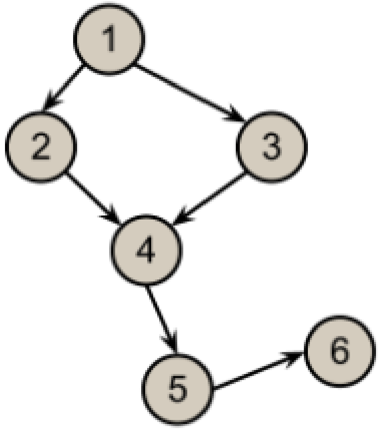
- V is the *vertex set (also node set)* whose elements are the vertices, or *nodes* of the graph.

- E is the *edge set* whose elements are the edges, or connections between vertices, of the graph. If the graph is undirected, individual edges are unordered pairs.

- If the graph is directed, edges are ordered pairs

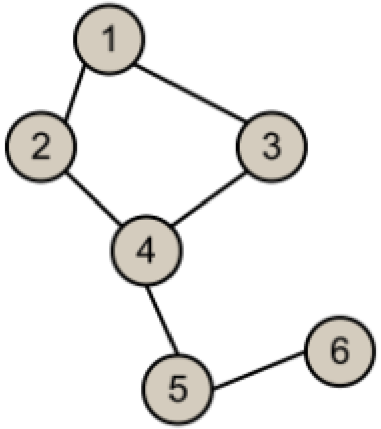


undirected



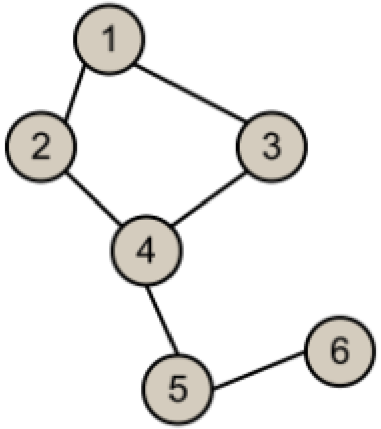
directed

Graphs representation: adjacency matrix

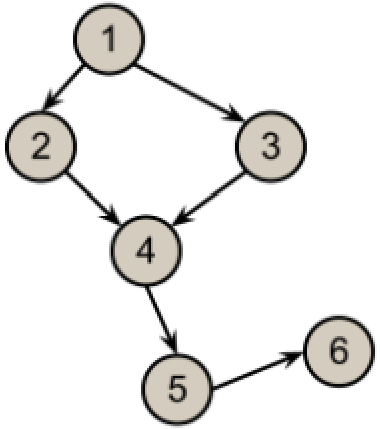


	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	0	1	0	0
3	1	0	0	1	0	0
4	0	1	1	0	1	0
5	0	0	0	1	0	1
6	0	0	0	0	1	0

Graphs representation: adjacency matrix



	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	0	1	0	0
3	1	0	0	1	0	0
4	0	1	1	0	1	0
5	0	0	0	1	0	1
6	0	0	0	0	1	0



	1	2	3	4	5	6
1	0	1	1	0	0	0
2	-1	0	0	1	0	0
3	-1	0	0	1	0	0
4	0	-1	-1	0	1	0
5	0	0	0	-1	0	1
6	0	0	0	0	-1	0

Graphs and where to find them

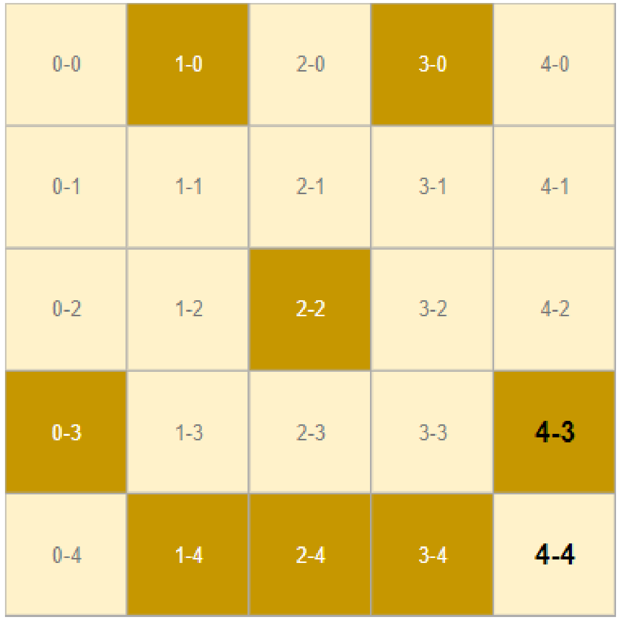
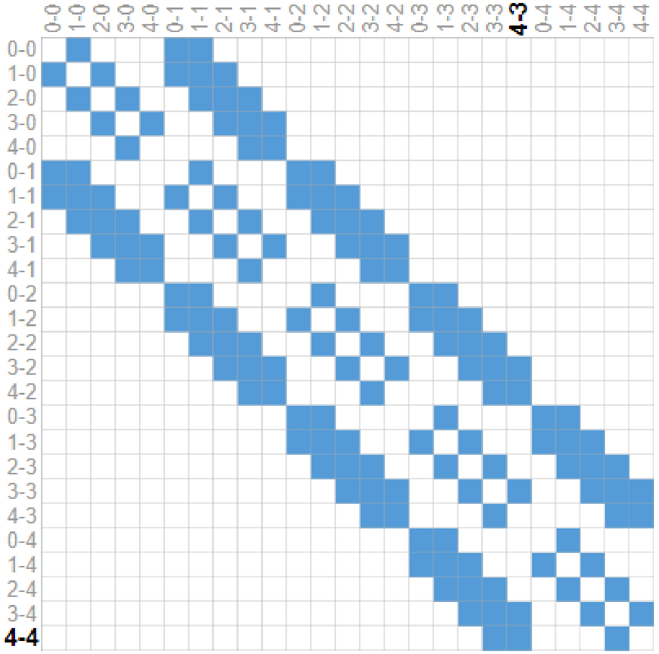
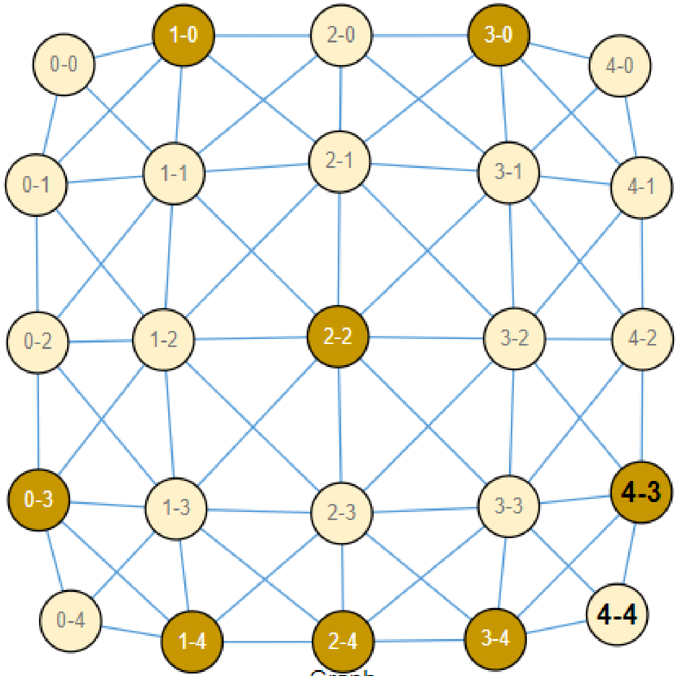


Image Pixels



Adjacency Matrix

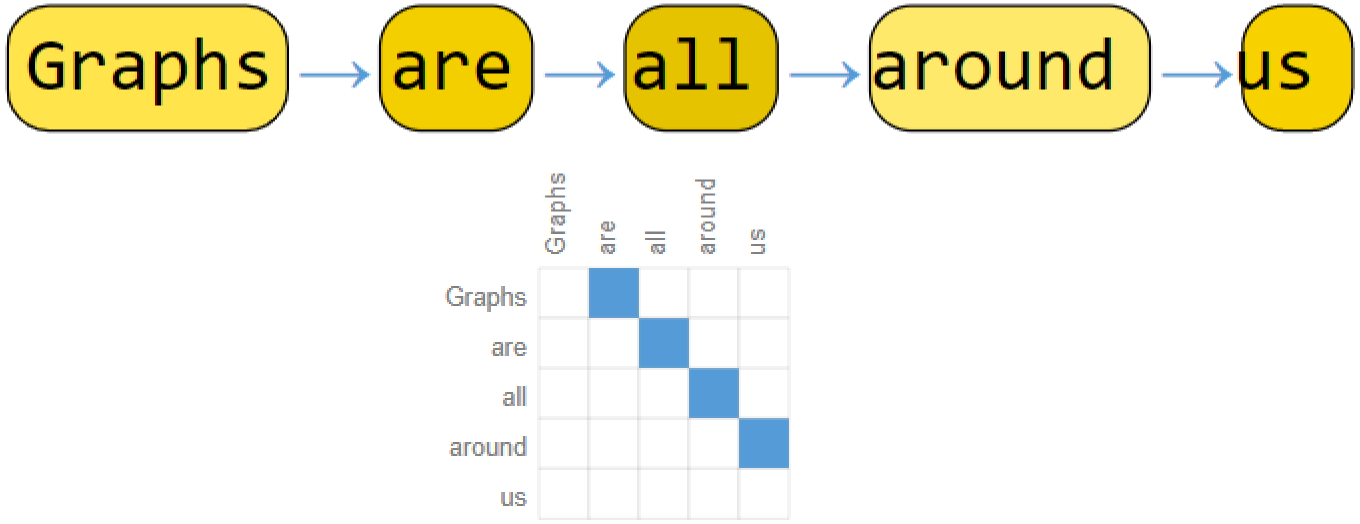


Graph

Images as graphs

Image source

Graphs and where to find them

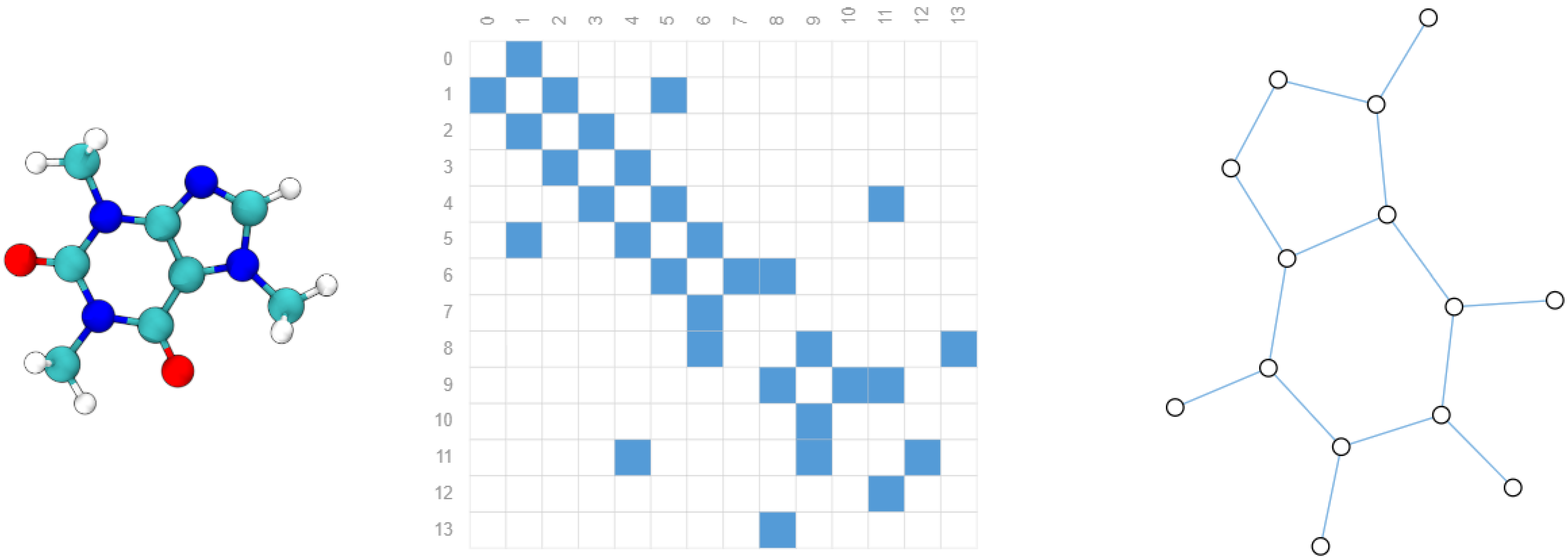


	Graphs	are	all	around	us
Graphs		■			
are			■		
all				■	
around					■
us					

Text as graphs

Image source

Graphs and where to find them

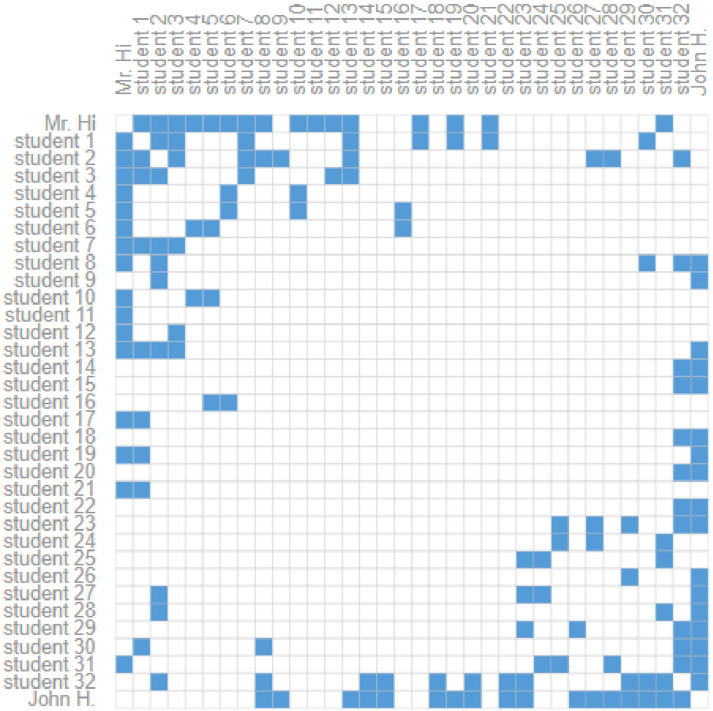


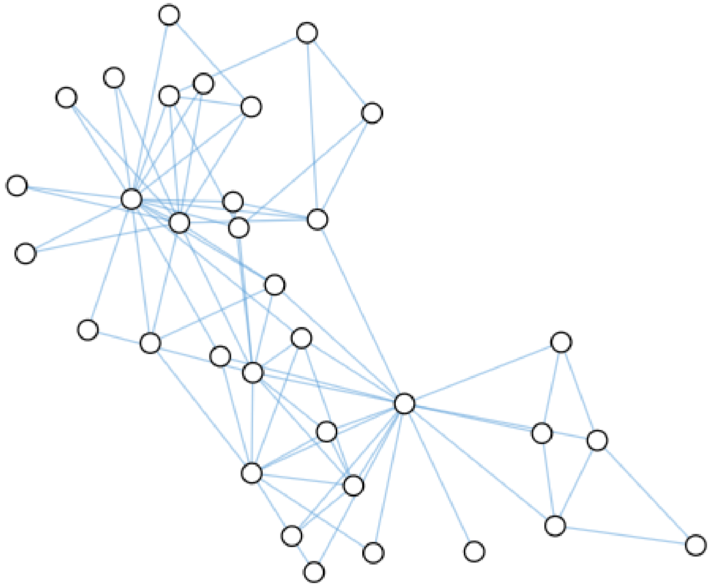
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0		■												
1	■		■			■								
2		■		■										
3			■		■									
4				■		■						■		
5		■			■		■							
6						■		■	■					
7							■							
8							■			■				■
9									■		■	■		
10										■				
11					■					■			■	
12												■		
13									■					

(Left) 3d representation of the Caffeine molecule (Center) Adjacency matrix of the bonds in the molecule (Right) Graph representation of the molecule.

Image source

Other examples

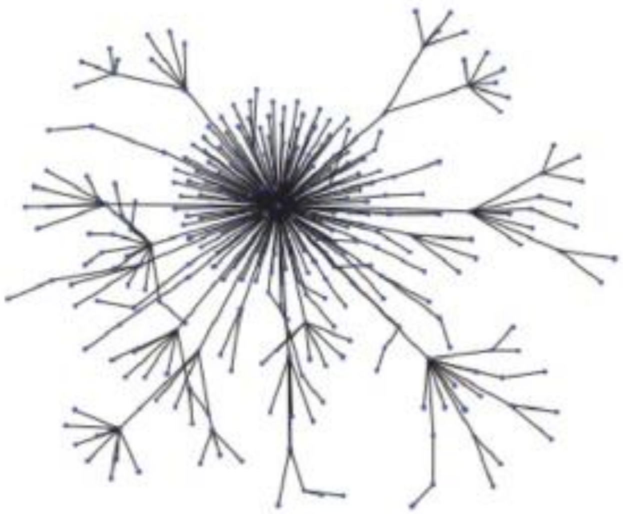




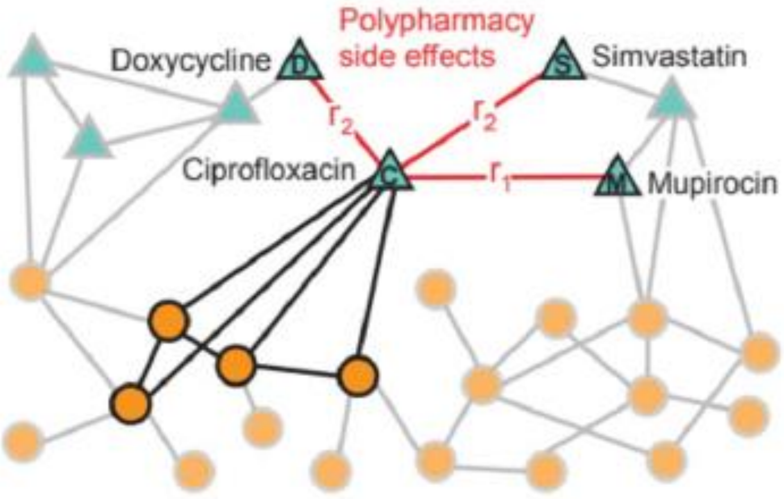
Social networks as graphs

Image source

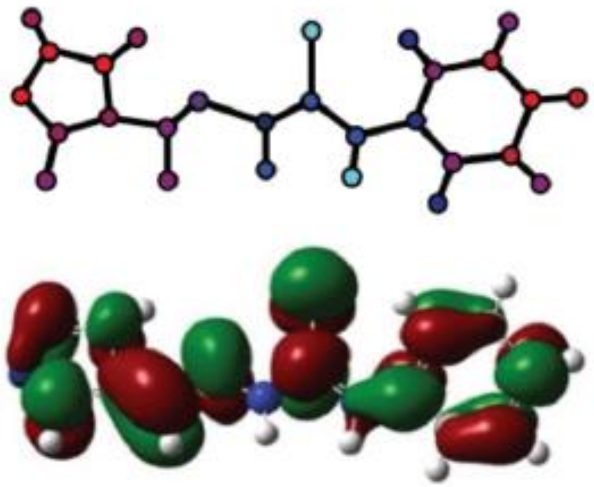
Other examples



Fake news detection



Polypharmacy

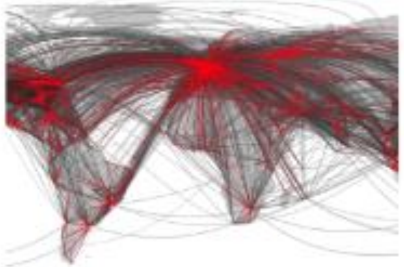


Chemistry

Image source

Other examples

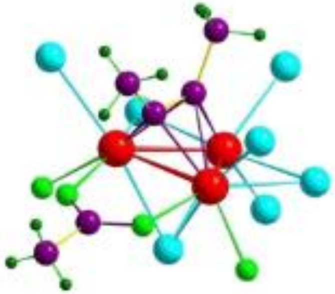
Social Graphs



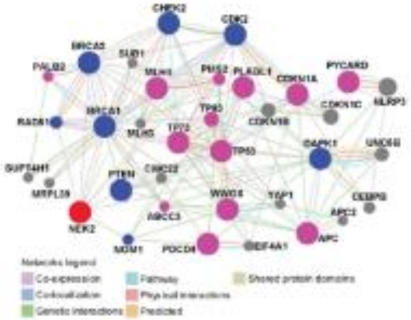
Transportation Graphs

Brain Graphs

Web Graphs



Molecular Graphs

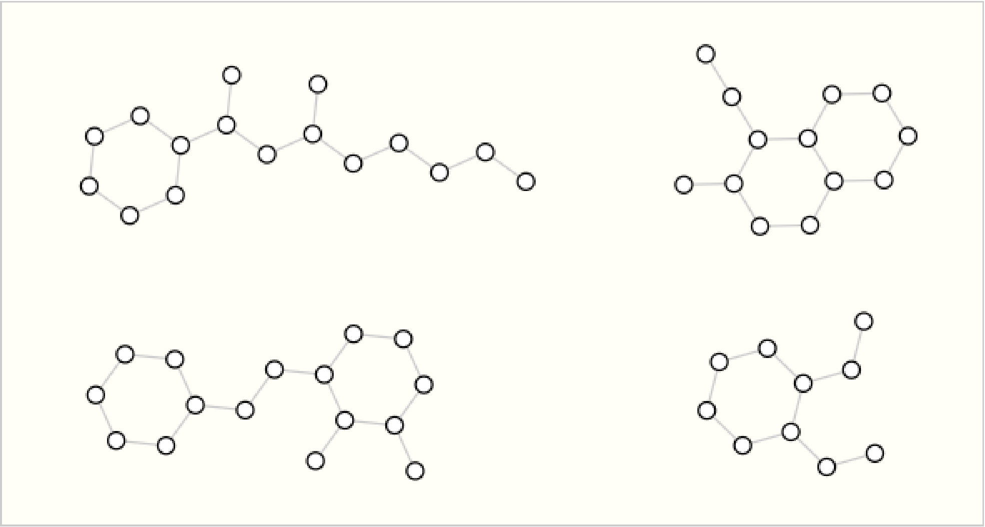


Gene Graphs

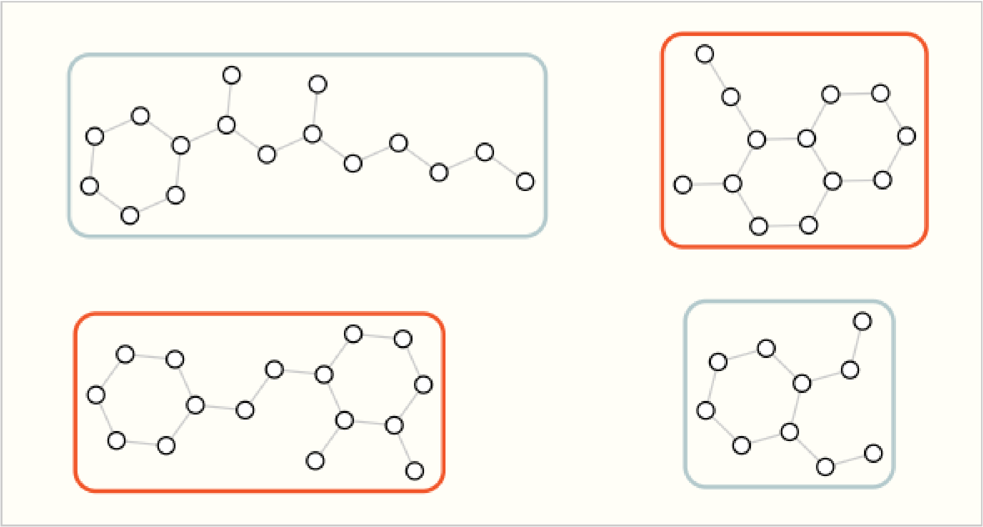
Protein-protein interactions , Citation networks as graphs, Machine learning models, etc.

ML on graphs: Graph level task

In a graph-level task, our goal is to predict the property of an entire graph.



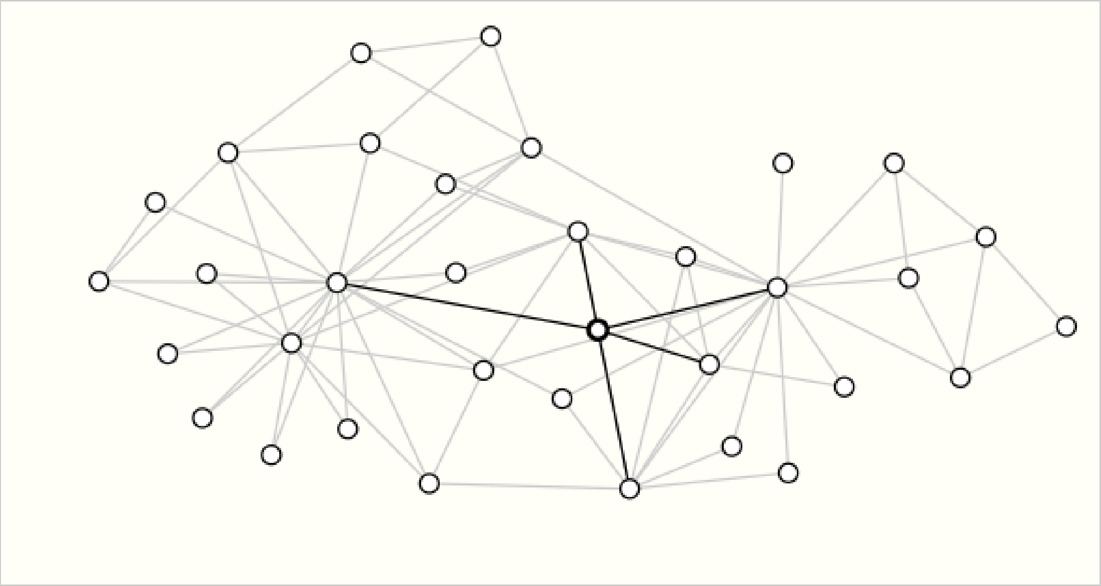
Input: graphs



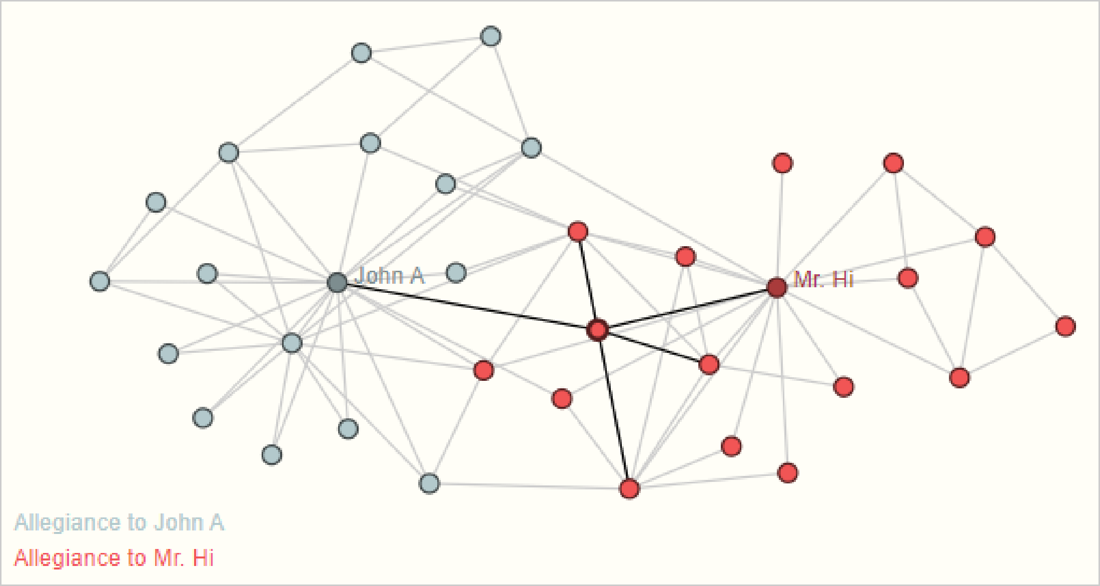
Output: labels for each graph, (e.g., "does the graph contain two rings?")

ML on graphs: Node-level task

In a graph-level task, our goal is to predict the property of an entire graph.



→



Input: graph with unlabled nodes

Output: graph node labels

Problem setup

Assume we have a graph $\boldsymbol{G}$:

- V is the vertex set
- $\boldsymbol{A}$ is the adjacency matrix (assume binary)
- $\boldsymbol{X} \in \mathbb{R}^{m \times |V|}$ is a matrix of node features
- v: a node in V; $N(v)$: the set of neighbors of v.

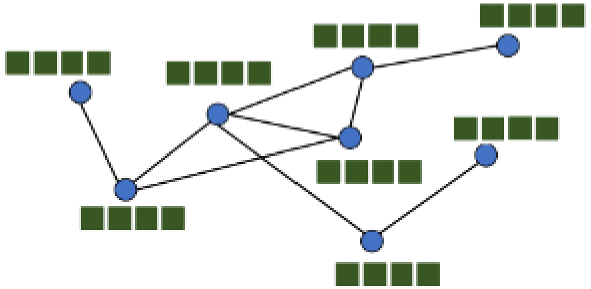
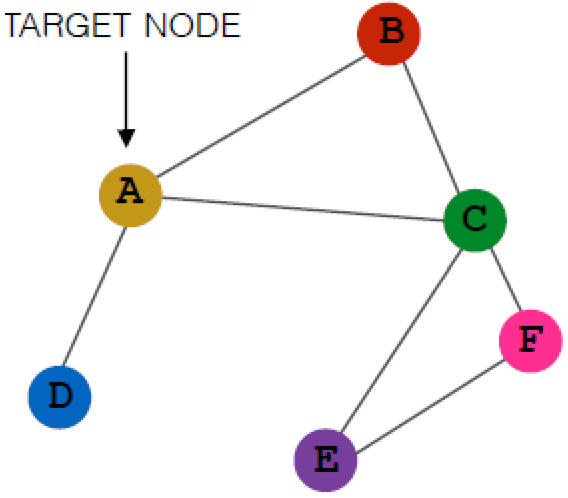
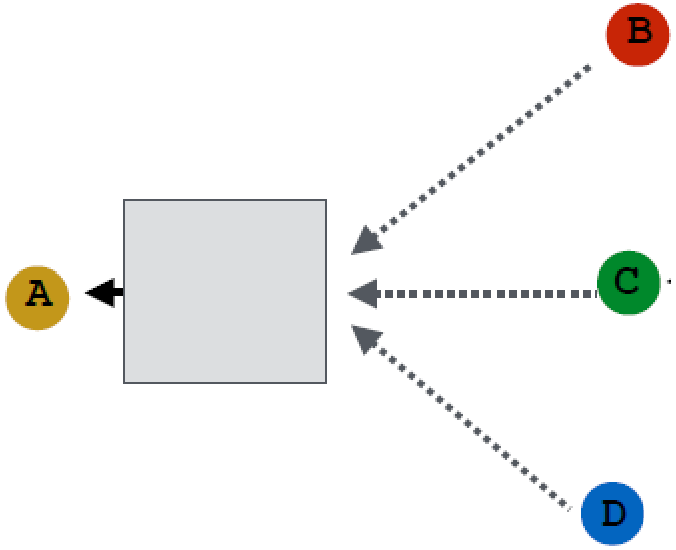


Image source

Graph Neural Networks



INPUT GRAPH

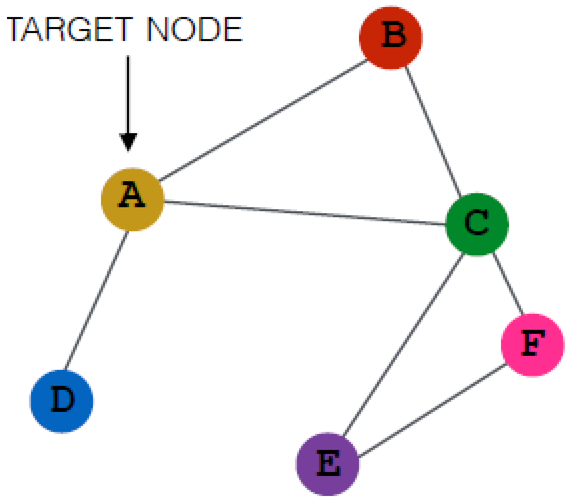


Key idea: Each node defines a computation graph

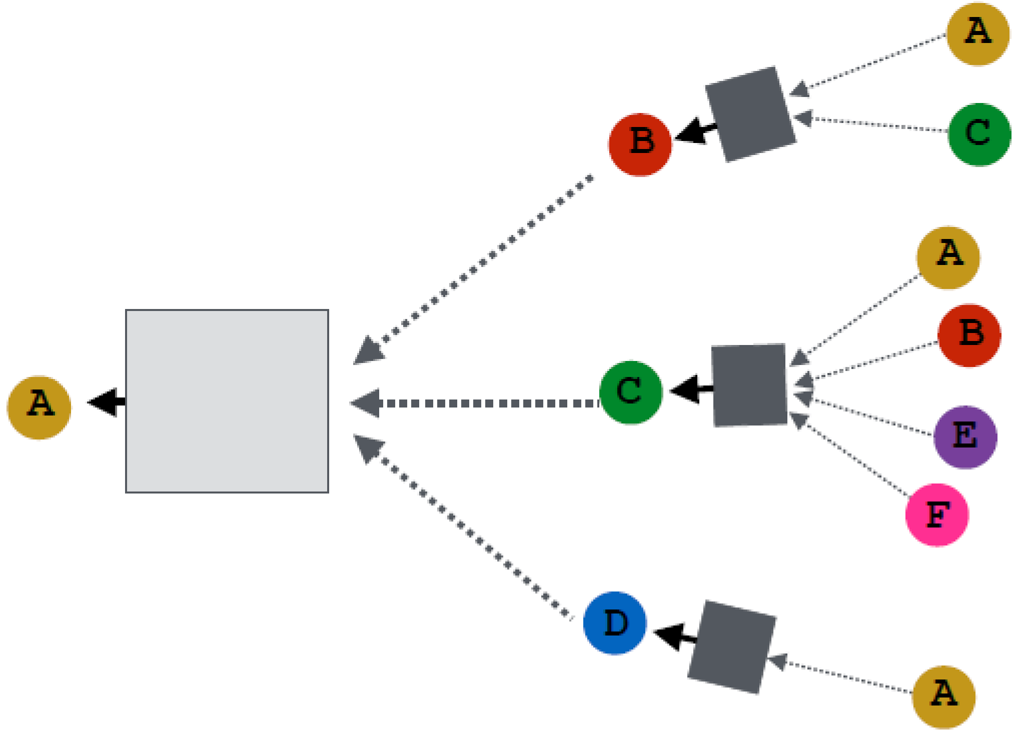
- Each edge in this graph is a transformation/aggregation function

Image source

Graph Neural Networks



INPUT GRAPH



Key idea: Each node defines a computation graph

- Each edge in this graph is a transformation/aggregation function

Image source

Graph Neural Networks

Nodes have embeddings at each layer

- Layer-0 embedding of node u is its input feature, x_u
- Layer-k embedding gets information from nodes that are K hops away

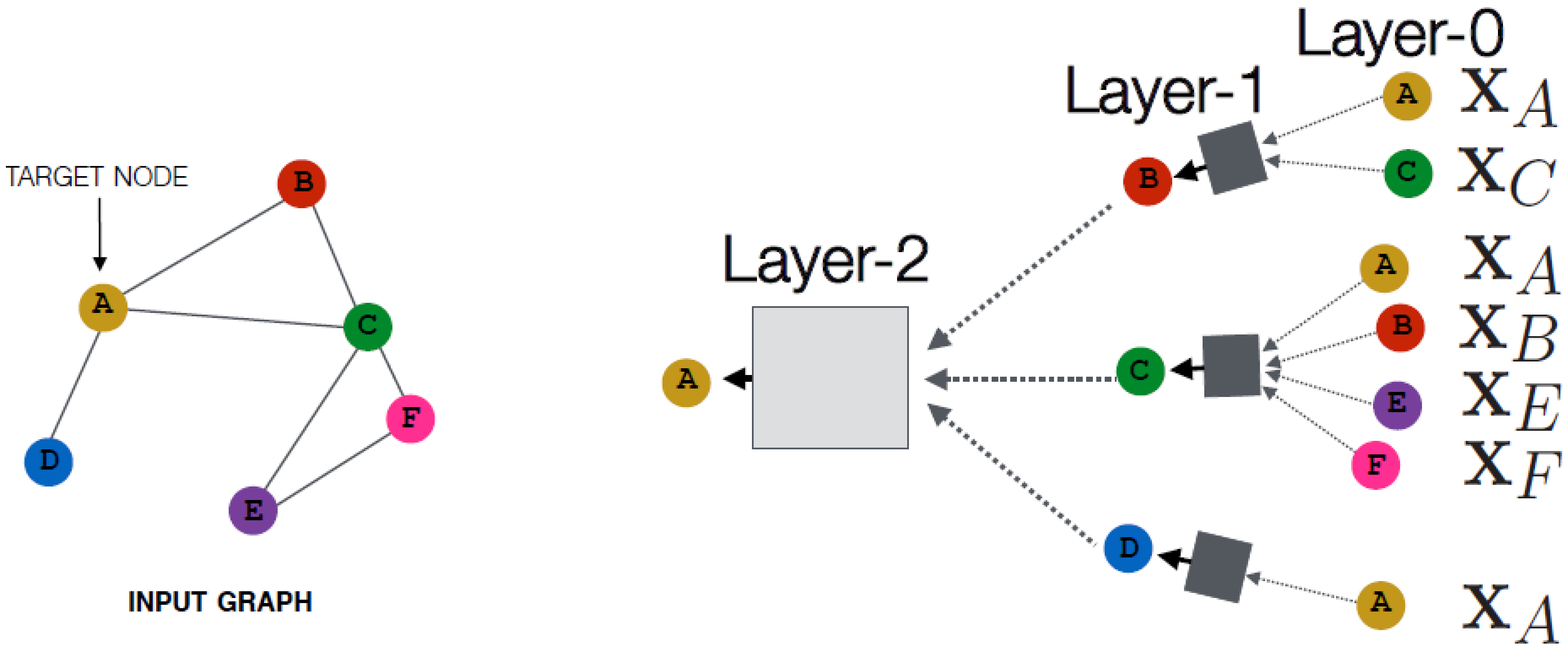
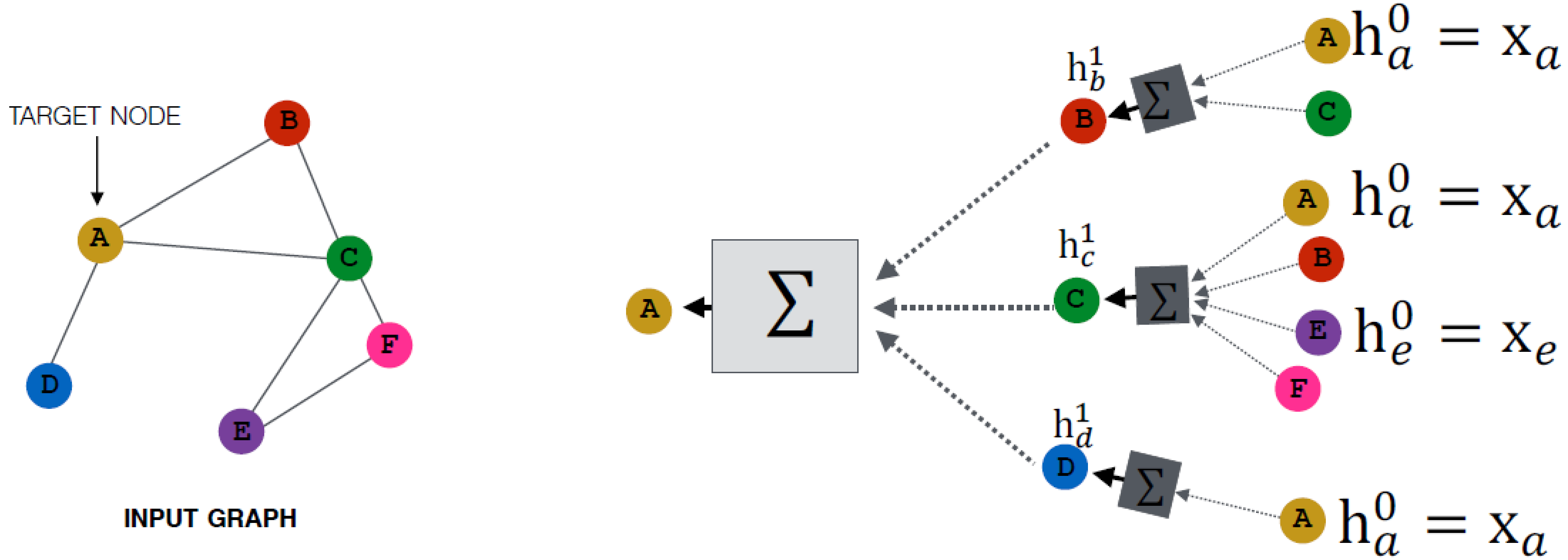


Image source

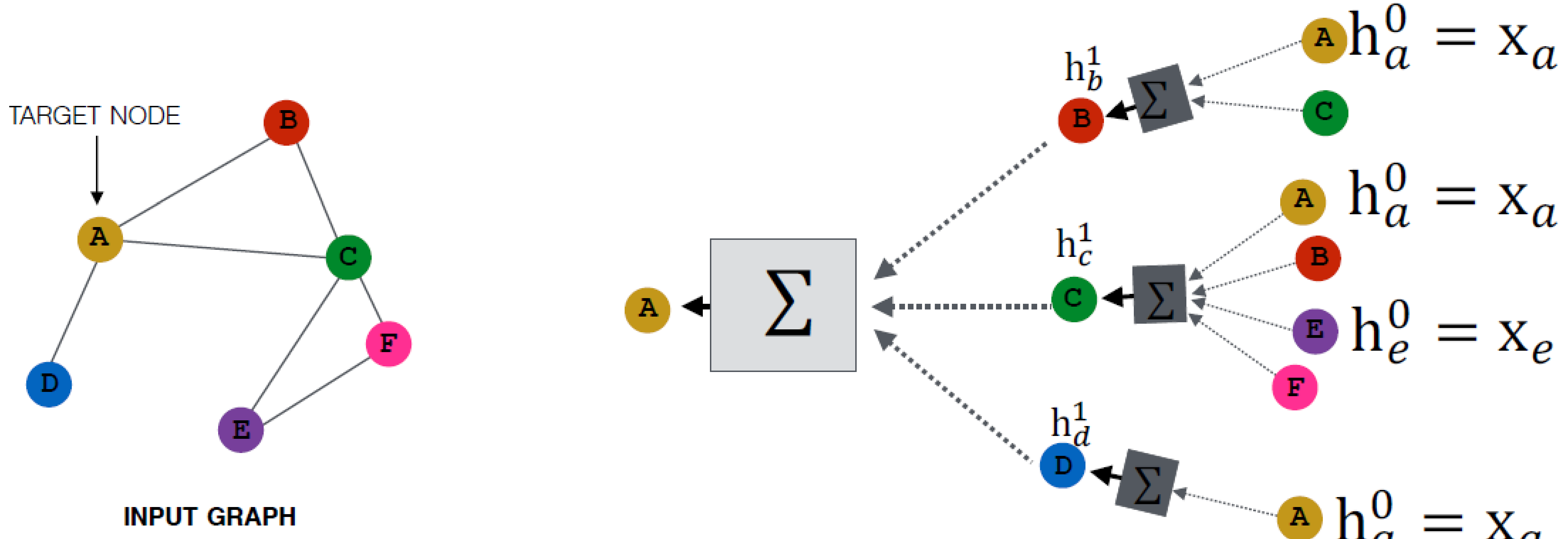
Graph Neural Networks



$$h_v^{(l+1)} = \sigma(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)})$$

Image source

Graph Neural Networks



$$h_v^{(l+1)} = \sigma\left(\boxed{W_l} \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + \boxed{B_l} h_v^{(l)}\right)$$

Learnable parameters

Image source

Graph Neural Networks

The *same* aggregation parameters are shared for all nodes

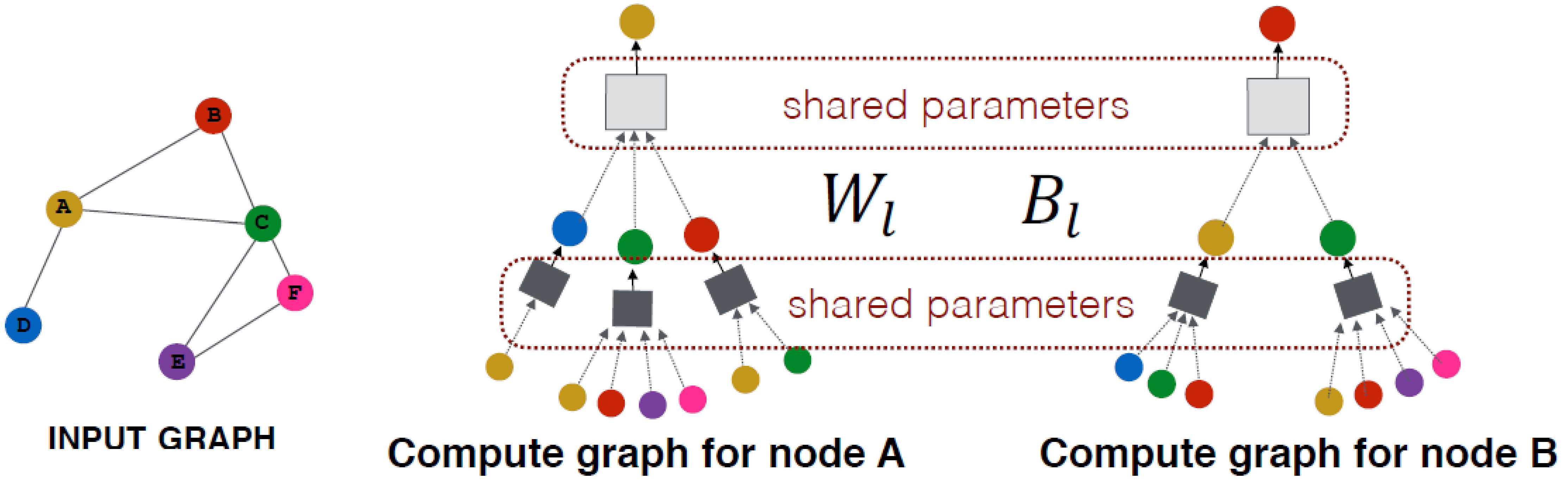
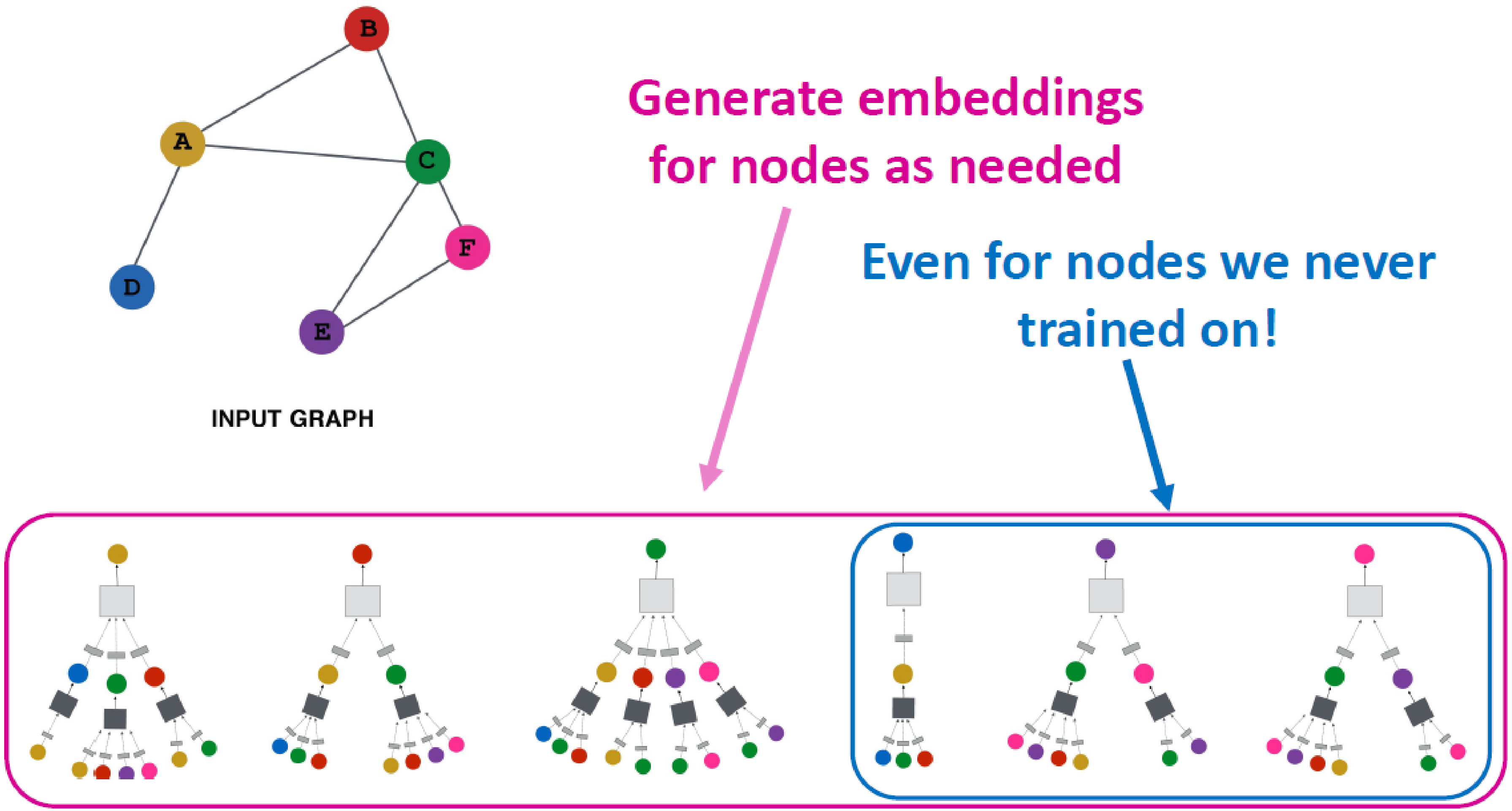


Image source

Graph Neural Networks



INPUT GRAPH

Generate embeddings for nodes as needed

Even for nodes we never trained on!

Image source

Key Benefits

*No manual feature engineering needed

*End-to-end learning results in optimal features.

*Any graph machine learning task: Node-level, link-level, entire graph-level prediction

*Scalable to billion node graphs!

Image source

Graph Convolutional Network

$$h_v^{k+1} = \sigma(W_k \sum_{u \in N(u)} \frac{h_u^k}{|N(v)|} + B_k h_v^k)$$

Key benefit : fast and easy to implement

Image source

Graph Neural Networks

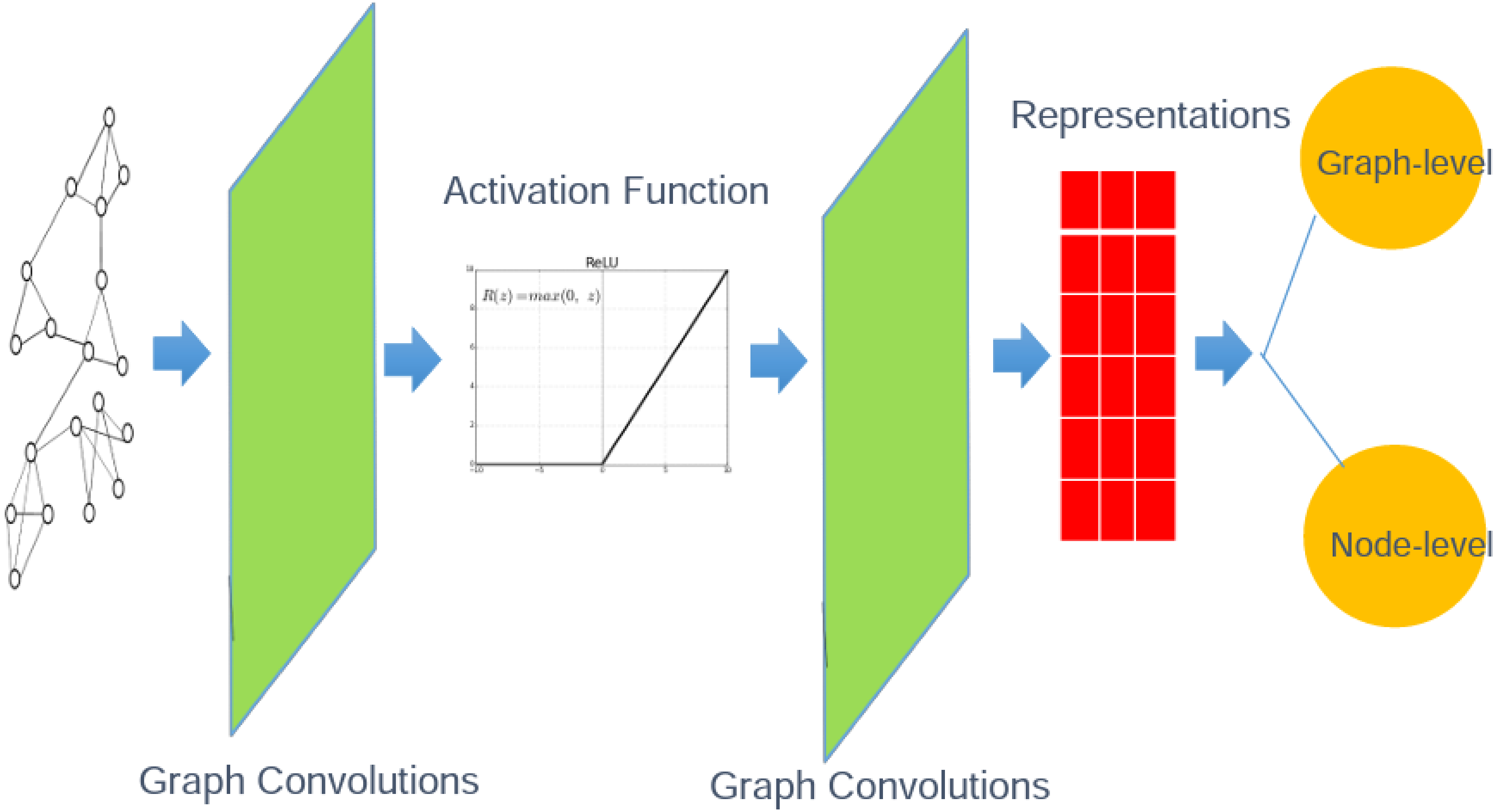


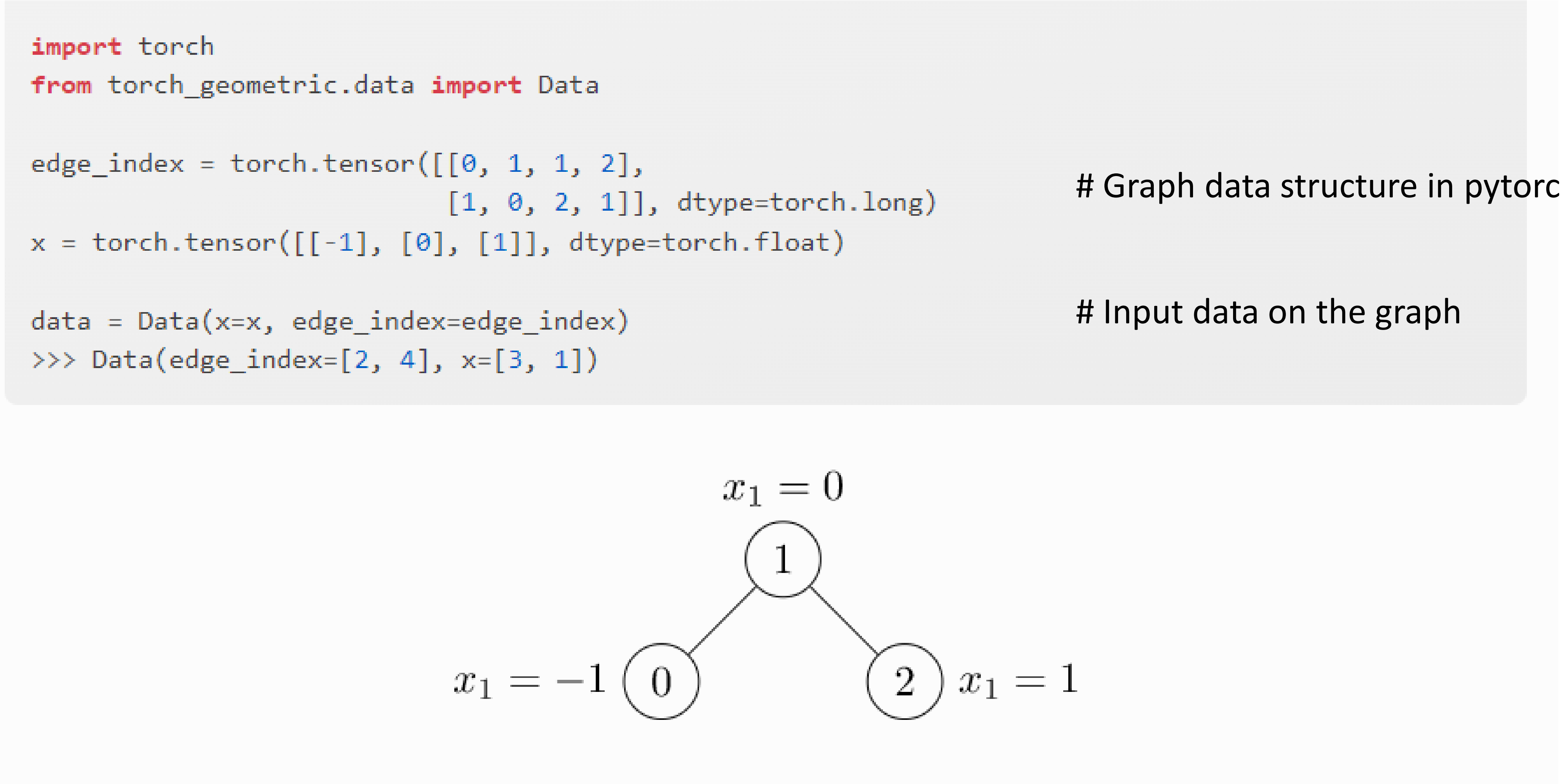
Image source

Pytorch Geometric

PyG (PyTorch Geometric) is a library built upon PyTorch to easily write and train Graph Neural Networks (GNNs) for a wide range of applications related to structured data.

https://pytorch-geometric.readthedocs.io/en/latest/get_started/colabs.html

Pytorch Geometric



```
import torch
from torch_geometric.data import Data

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

data = Data(x=x, edge_index=edge_index)
>>> Data(edge_index=[2, 4], x=[3, 1])
```

# Graph data structure in pytorch Geometric

# Input data on the graph

Although the graph has only two edges, we need to define four index tuples to account for both directions of an edge

https://pytorch-geometric.readthedocs.io/en/latest/get_started/introduction.html

Pytorch Geometric

```
import torch
from torch.nn import Linear
from torch_geometric.nn import GCNConv

conv1 = GCNConv(num_of_input_features, num_of_out_features) # init of the GNN
```

$$h_v^{k+1} = \sigma(W_k \sum_{u \in N(u)} \frac{h_u^k}{|N(v)|} \quad)$$

Pytorch Geometric

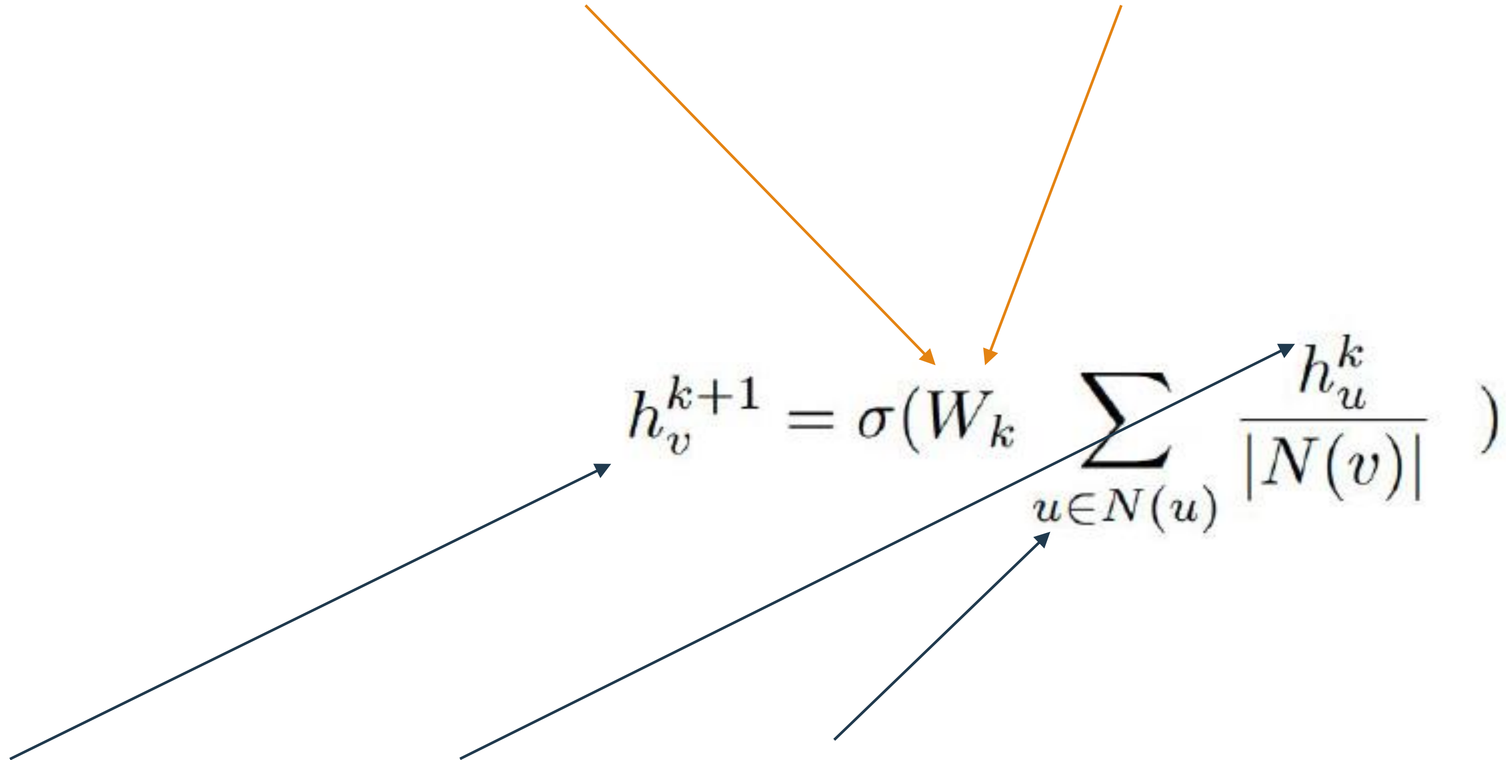
```
import torch
from torch.nn import Linear
from torch_geometric.nn import GCNConv

conv1 = GCNConv(num_of_input_features, num_of_out_features) # init of the GNN
```

$$h_v^{k+1} = \sigma(W_k \sum_{u \in N(u)} \frac{h_u^k}{|N(v)|} \quad)$$

```
h = conv1(input_features, graph_edges)      # running the GNN on an input
```

Pytorch Geometric

```python
import torch
from torch.nn import Linear
from torch_geometric.nn import GCNConv

conv1 = GCNConv(num_of_input_features, num_of_out_features) # init of the GNN
```



$$h_v^{k+1} = \sigma(W_k \sum_{u \in N(u)} \frac{h_u^k}{|N(v)|} \quad)$$

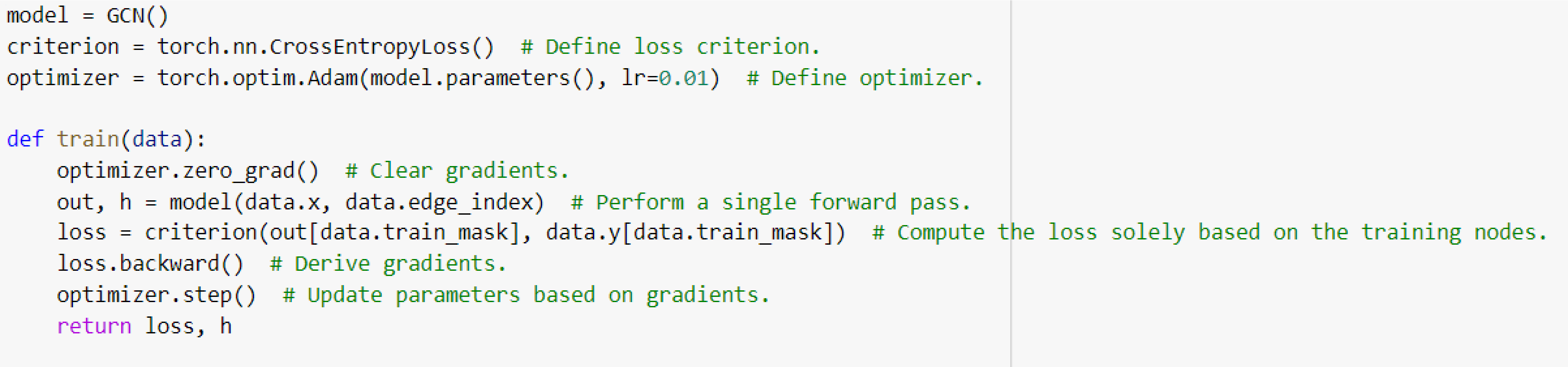
```python
h = conv1(input_features, graph_edges)      # running the GNN on an input
```

Pytorch Geometric

```
import torch
from torch.nn import Linear
from torch_geometric.nn import GCNConv


class GCN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        torch.manual_seed(1234)
        self.conv1 = GCNConv(dataset.num_features, 4)
        self.conv2 = GCNConv(4, 4)
        self.conv3 = GCNConv(4, 2)
        self.classifier = Linear(2, dataset.num_classes)

    def forward(self, x, edge_index):
        h = self.conv1(x, edge_index)
        h = h.tanh()
        h = self.conv2(h, edge_index)
        h = h.tanh()
        h = self.conv3(h, edge_index)
        h = h.tanh()  # Final GNN embedding space.

        # Apply a final (linear) classifier.
        out = self.classifier(h)

        return out, h
```

https://colab.research.google.com/drive/14OvFnAXggxB8vM4e8vSURUp1TaKnovzX?usp=sharing

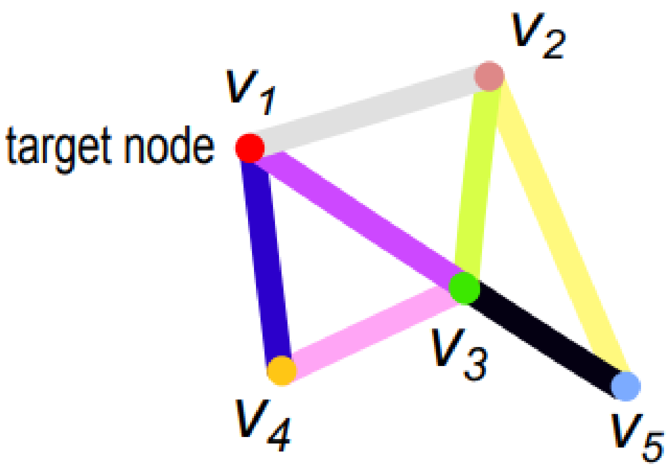
Pytorch Geometric



```python
model = GCN()
criterion = torch.nn.CrossEntropyLoss()  # Define loss criterion.
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)  # Define optimizer.

def train(data):
    optimizer.zero_grad()  # Clear gradients.
    out, h = model(data.x, data.edge_index)  # Perform a single forward pass.
    loss = criterion(out[data.train_mask], data.y[data.train_mask])  # Compute the loss solely based on the training nodes.
    loss.backward()  # Derive gradients.
    optimizer.step()  # Update parameters based on gradients.
    return loss, h
```

Message Passing Neural Networks

1. A graph $G = (V,E)$.
2. For each node $i \in V$ we have an initial vector $h_i^{(0)} \in \mathbb{R}^{l_0}$.

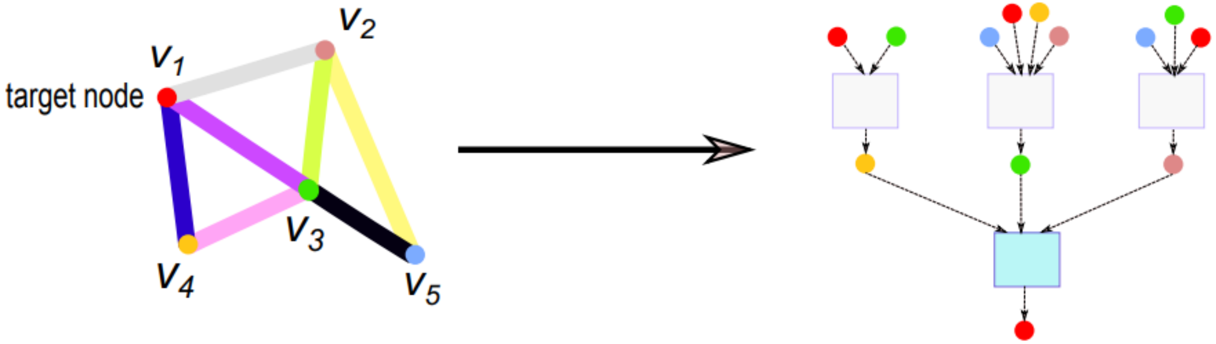


Message Passing Neural Networks

1. A graph $G = (V,E)$.
2. For each node $i \in V$ we have an initial vector $h_i^{(0)} \in \mathbb{R}^{l_0}$.

Given the above data, the feedforward neural algorithm on G executes L message passing schemes defined recursively for 0 ≤ k ≤ L by:

$$h_i^{(k)} := \alpha^k\left(h_i^{(k\text{-}1)}, E_{j\in\mathcal{N}(i)}\left(\phi^k(h_i^{(k-1)},h_j^{(k-1)}, e_{i,j})\right)\right) \in \mathbb{R}^{l_k},$$

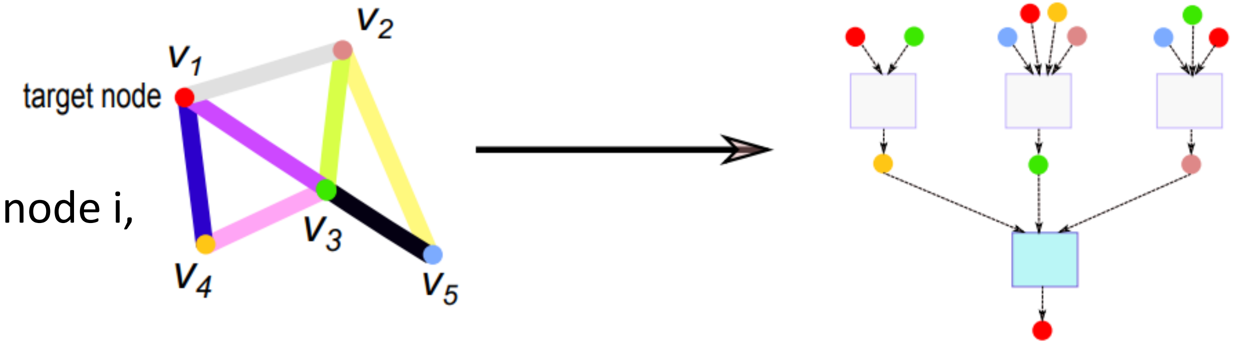


Message Passing Neural Networks

1. A graph $G = (V,E)$.
2. For each node $i \in V$ we have an initial vector $h_i^{(0)} \in \mathbb{R}^{l_0}$.

Given the above data, the feedforward neural algorithm on G executes L message passing schemes defined recursively for 0 ≤ k ≤ L by:

$$h_i^{(k)} := \alpha^k\left(h_i^{(k\text{-}1)}, E_{j\in\mathcal{N}(i)}\left(\phi^k\left(h_i^{(k-1)}, h_j^{(k-1)}, e_{i,j}\right)\right)\right) \in \mathbb{R}^{l_k},$$



where $e_{ij} \in R^D$ is an edge feature from the node j to the node i,

E is a permutation invariant differentiable function,

α^k , φ^k are trainable differentiable functions (regular neural networks)

GraphSage

$$h_v^{k+1} = \sigma(W_k \sum_{u \in N(u)} \frac{h_u^k}{|N(v)|} + B_k h_v^k)$$

$$h_v^{k+1} = \sigma([W_k \cdot AGG(\{h_u^{k-1}, \forall u \in N(v)\}), B_k h_v^k])$$

Graph Attention Networks

$$\mathbf{h}_v^{(l)} = \sigma(\sum_{u\in N(v)} \boxed{\alpha_{vu}}\,\mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)})$$

Attention weights

Not all node's neighbors are equally important

Attention is inspired by cognitive attention.

The **attention** α_{vu} focuses on the important parts of the input data and fades out the rest.

Idea: the NN should devote more computing power on that small but important part of the data. Which part of the data is more important depends on the context and is learned through training.

Image source

Graph Attention Networks

$$\mathbf{h}_v^{(l)} = \sigma(\textstyle\sum_{u \in N(v)} \underbrace{\boxed{\alpha_{vu}}}_{\text{Attention weights}} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$

Goal: Specify **arbitrary importance** to different neighbors of each node in the graph

Idea: Compute embedding $\boldsymbol{h}_v^{(l)}$ of each node in the graph following an attention strategy: ✦ Nodes attend over their neighborhoods' message

✦ Implicitly specifying different weights to different nodes in a neighborhood

Image source

Graph Attention Networks

$$\mathbf{h}_v^{(l)} = \sigma(\textstyle\sum_{u \in N(v)} \boxed{\alpha_{vu}} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$

Attention weights

- Let α_{vu} be computed as a byproduct of an **attention mechanism $\boldsymbol{a}$:**
  - (1) Let a compute **attention coefficients $\boldsymbol{e_{vu}}$** across pairs of nodes u, v based on their messages:

$$e_{vu} = a(\mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}, \mathbf{W}^{(l)}\boldsymbol{h}_v^{(l-1)})$$

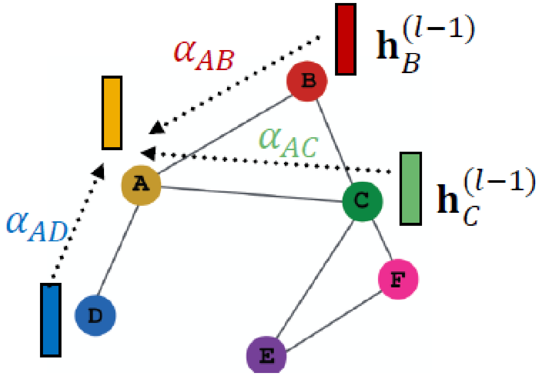


Image source

Graph Attention Networks

$$\mathbf{h}_v^{(l)} = \sigma(\textstyle\sum_{u \in N(v)} \boxed{\alpha_{vu}} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$

Attention weights

- Let α_{vu} be computed as a byproduct of an **attention mechanism $\boldsymbol{a}$:**
  - (1) Let a compute **attention coefficients $\boldsymbol{e_{vu}}$** across pairs of nodes u, v based on their messages:

$$e_{vu} = a(\mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}, \mathbf{W}^{(l)}\boldsymbol{h}_v^{(l-1)})$$

    - **e_{vu} indicates the importance of $u's$ message to node v**

$$e_{AB} = a(\mathbf{W}^{(l)}\mathbf{h}_A^{(l-1)}, \mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)})$$

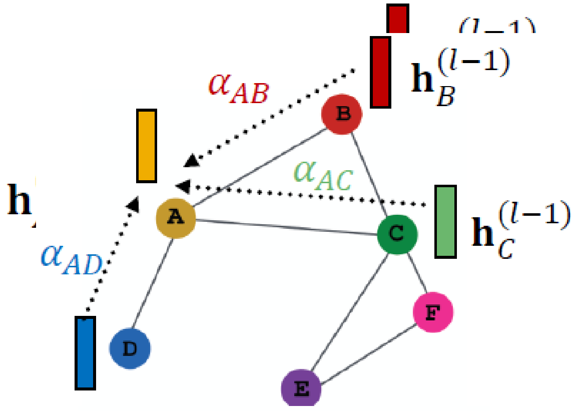


Image source

Graph Attention Networks

$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \boxed{\alpha_{vu}} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}\right)$$

Attention weights

- **Normalize** e_{vu} into the **final attention weight** α_{vu}
  - Use the **softmax** function, so that $\sum_{u \in N(v)} \alpha_{vu} = 1$:

$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

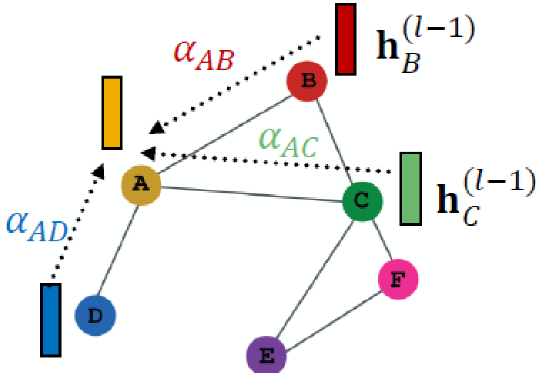


Image source

Graph Attention Networks

$$\mathbf{h}_v^{(l)} = \sigma(\textstyle\sum_{u \in N(v)} \boxed{\alpha_{vu}} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$

Attention weights

- **Normalize** e_{vu} into the **final attention weight** α_{vu}
  - Use the **softmax** function, so that $\sum_{u \in N(v)} \alpha_{vu} = 1$:

$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

- **Weighted sum** based on the **final attention weight** α_{vu}

$$\mathbf{h}_v^{(l)} = \sigma(\textstyle\sum_{u \in N(v)} \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)})$$

Weighted sum using α_{AB}, α_{AC}, α_{AD}:

$$\mathbf{h}_A^{(l)} = \sigma(\alpha_{AB}\mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)} + \alpha_{AC}\mathbf{W}^{(l)}\mathbf{h}_C^{(l-1)} + \alpha_{AD}\mathbf{W}^{(l)}\mathbf{h}_D^{(l-1)})$$

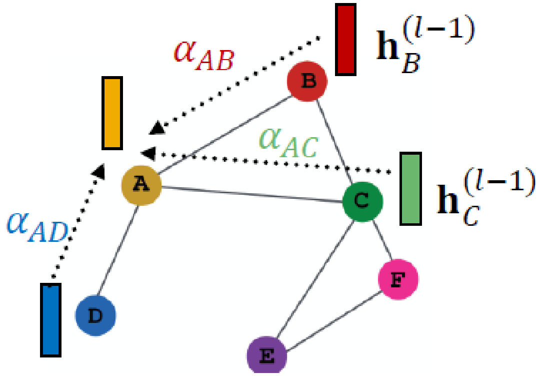


Image source

The over smoothing problem

The Issue of stacking many GNN layers

- GNN suffers from **the over-smoothing problem**
- **The over-smoothing problem: all the node embeddings converge to the same value**
- This is bad because we **want to use node embeddings to differentiate nodes**
- **Why does the over-smoothing problem happen?**

Image source

The over smoothing problem

Receptive field: the set of nodes that determine the embedding of a node of interest

- **In a K-layer GNN, each node has a receptive field of K-hop neighborhood**

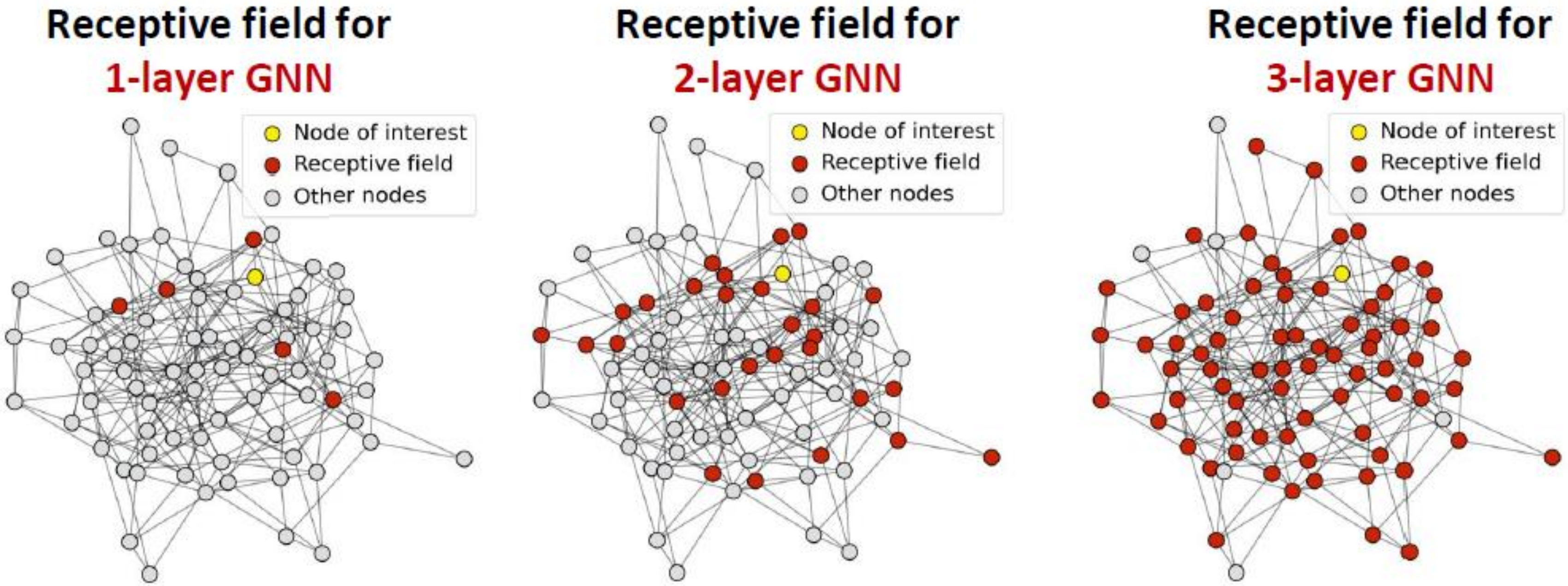
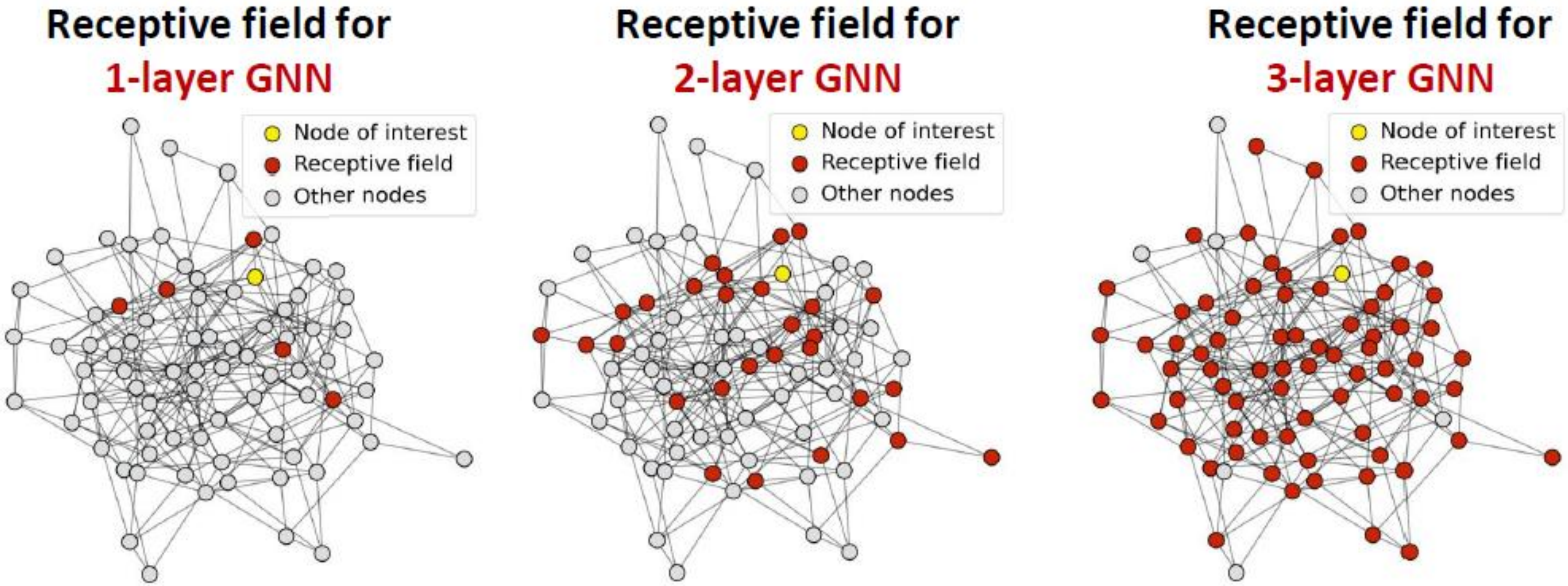


Image source

The over smoothing problem

Receptive field overlap for two nodes

The shared neighbors quickly grows when we increase the number of hops (num of GNN layers)



Receptive field for 1-layer GNN

Receptive field for 2-layer GNN

Receptive field for 3-layer GNN

Image source

The over smoothing problem

We can explain over-smoothing via the notion of receptive field

- We knew **the embedding of a node is determined by its receptive field**

- If two nodes **have highly-overlapped receptive fields, then their embeddings are highly similar**
- **Stack many GNN layers ->nodes will have highly-overlapped receptive fields->node embeddings will be highly similar ->suffer from the over-smoothing problem**
- **Next:** how do we overcome over-smoothing problem?

Image source

The over smoothing problem

What do we learn from the over-smoothing problem?

Lesson 1: Be cautious when adding GNN layers

- Unlike neural networks in other domains (CNN for image classification), **adding more GNN layers do not always help**
- **Step 1: Analyze the necessary receptive field** to solve your problem. E.g., by computing the diameter of the graph
- **Step 2:** Set number of GNN layers L to be a bit more than the receptive field we like. **Do not set L to be unnecessarily large**!

Image source

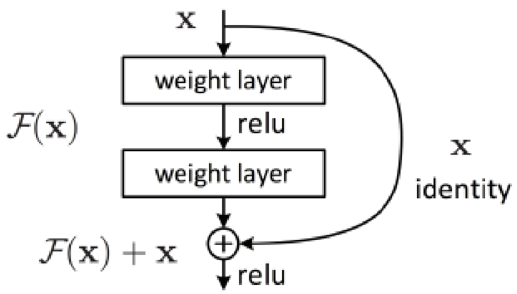
One possible solution

- A standard GCN layer
- $$\mathbf{h}_v^{(l)} = \sigma\left(\boxed{\sum_{u\in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}}\right)$$

This is our $F(\mathbf{x})$

- A GCN layer with skip connection
- $$\mathbf{h}_v^{(l)} = \sigma\left(\boxed{\sum_{u\in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}} + \boxed{\mathbf{h}_v^{(l-1)}}\right)$$

$F(\mathbf{x})$ $+$ $\mathbf{x}$



x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

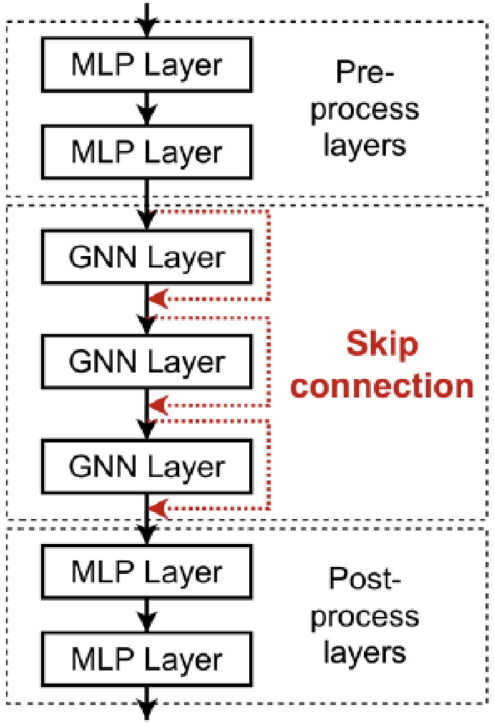


Image source

Graph Representation learning

Goal is to encode nodes so that similarity in the embedding space (e.g., dot product) approximates similarity in the original network.

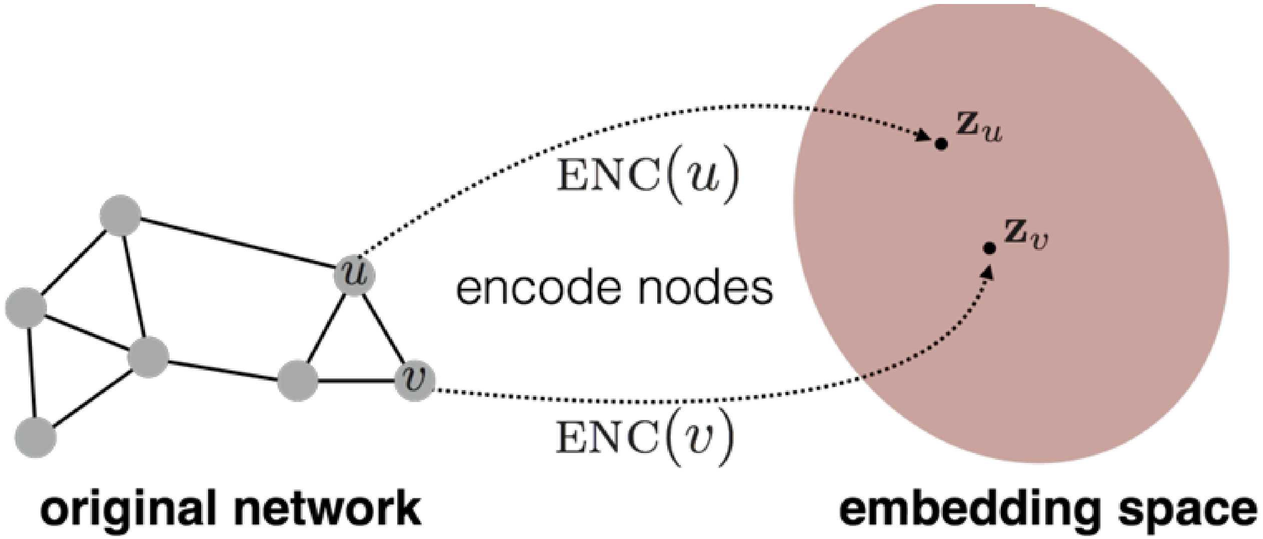


Image source

Graph Representation learning

Goal is to encode nodes so that similarity in the embedding space (e.g., dot product) approximates similarity in the original network.

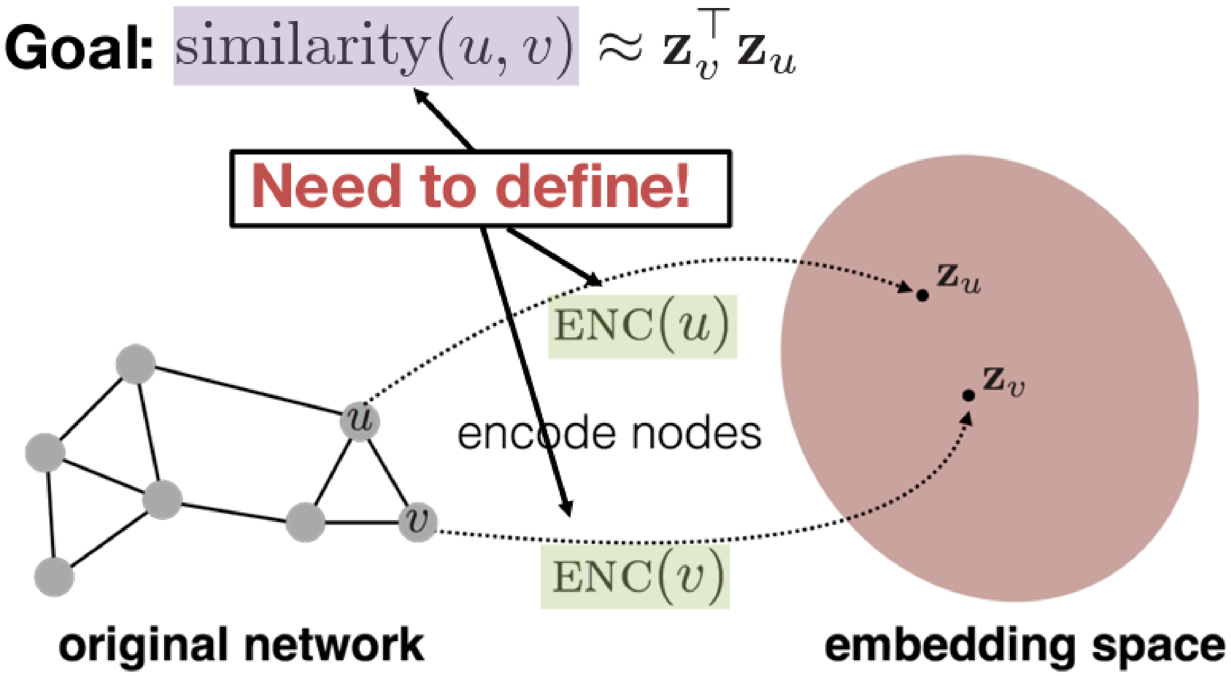
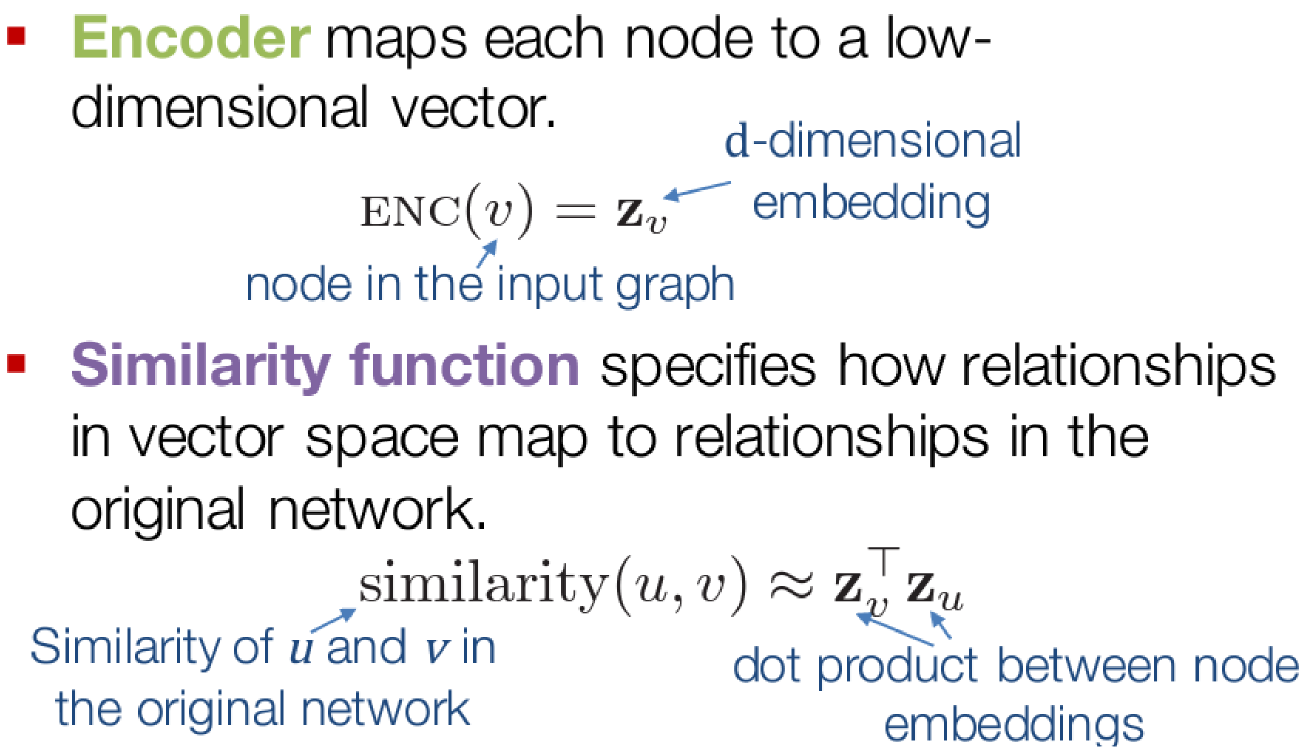


Image source

Graph Representation learning



- **Encoder** maps each node to a low-dimensional vector.

$$\mathrm{ENC}(v) = \mathbf{z}_v$$

d-dimensional embedding

node in the input graph

- **Similarity function** specifies how relationships in vector space map to relationships in the original network.

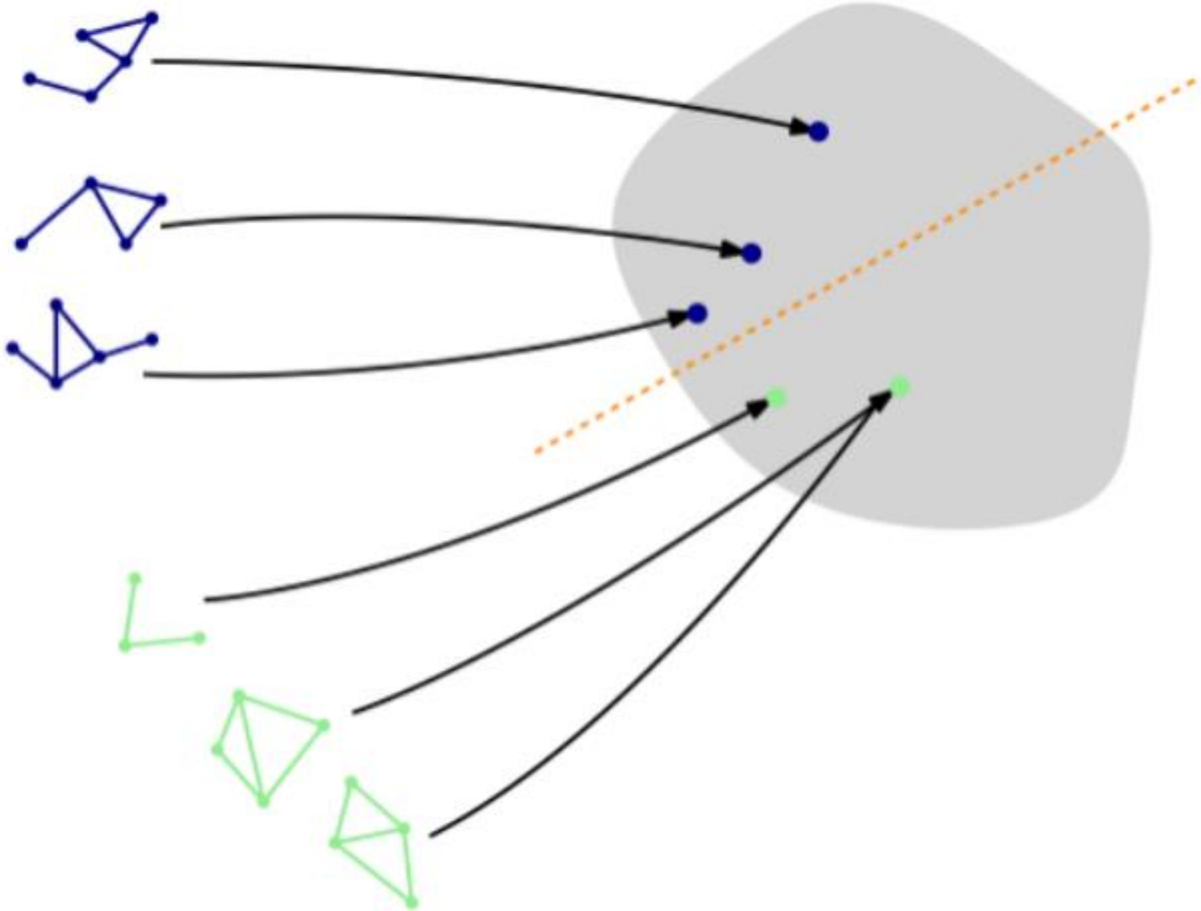
$$\mathrm{similarity}(u, v) \approx \mathbf{z}_v^\top \mathbf{z}_u$$

Similarity of u and v in the original network

dot product between node embeddings

[Image source](#)

Graph Classification with Pytorch Geometric



https://colab.research.google.com/drive/1I8a0DfQ3fI7Njc62__mVXUlcAleUclnb?usp=sharing

Graph Classification with Pytorch Geometric

Typically we have the following steps :

(1) Embed each node of the graph using one of the GNNs we introduced here.

(2) Aggregate all node embeddings and define a global graph embedding using aggregation invariant function (sum,min,max). For instance :

$$\mathbf{x}_{\mathcal{G}} = \frac{1}{|\mathcal{V}|} \sum_{v \in \mathcal{V}} x_v^{(L)}$$

https://colab.research.google.com/drive/1I8a0DfQ3fI7Njc62__mVXUlcAleUclnb?usp=sharing