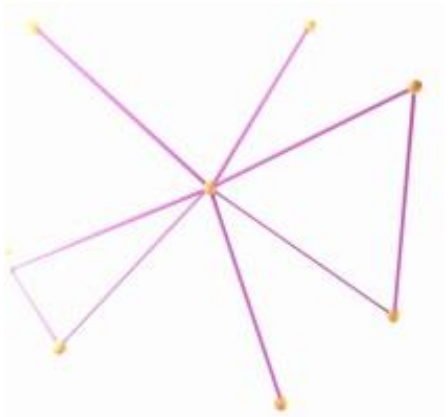


Graph Generative Models

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.

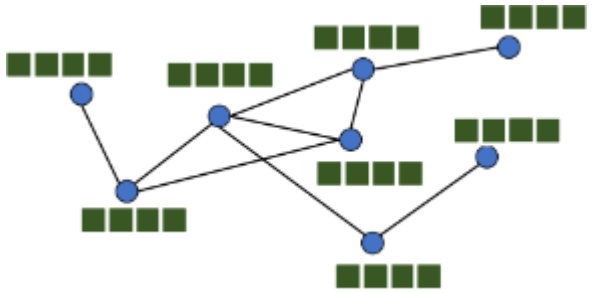
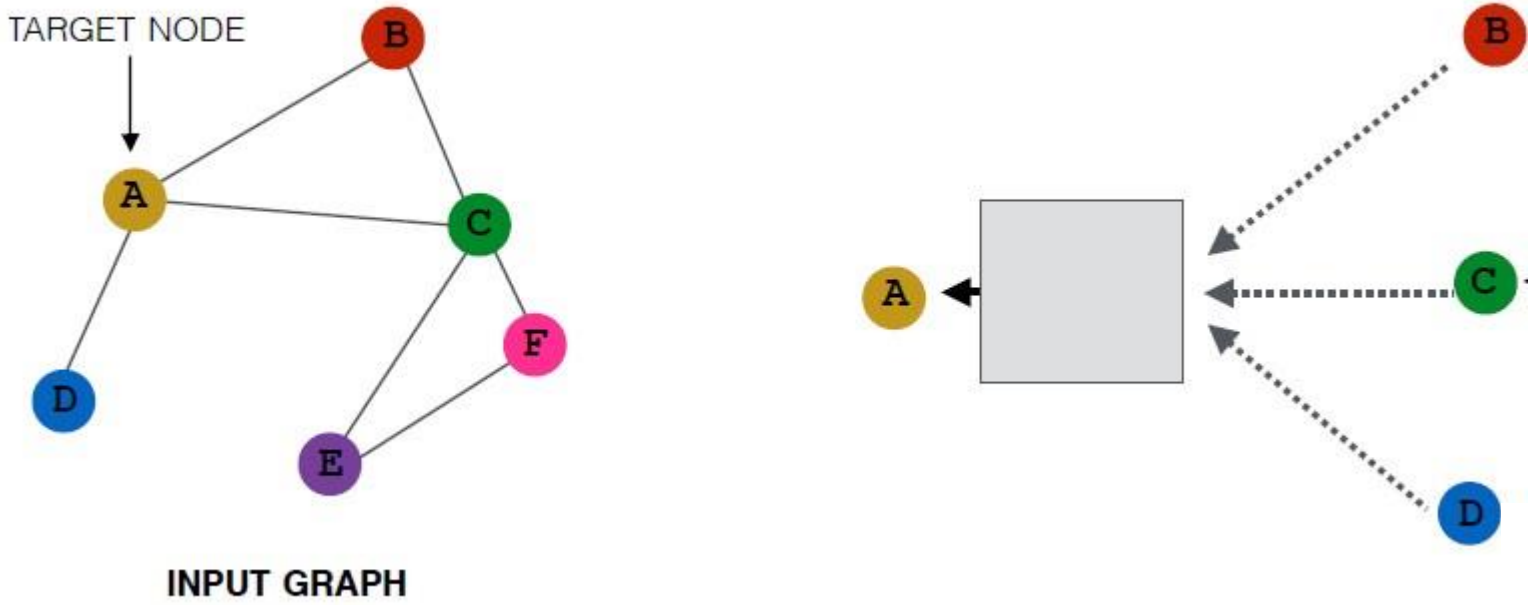


Image source

Graph Neural Networks

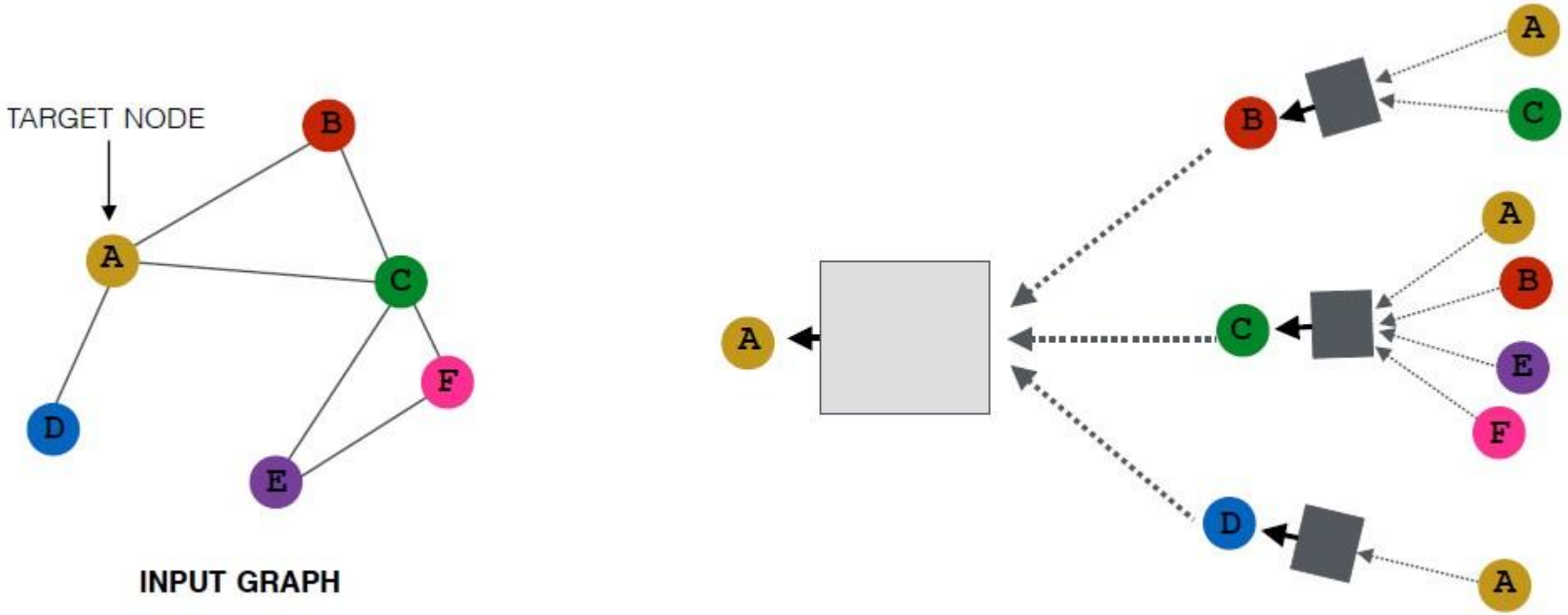


Key idea: Each node defines a computation graph

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

Image source

Graph Neural Networks



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

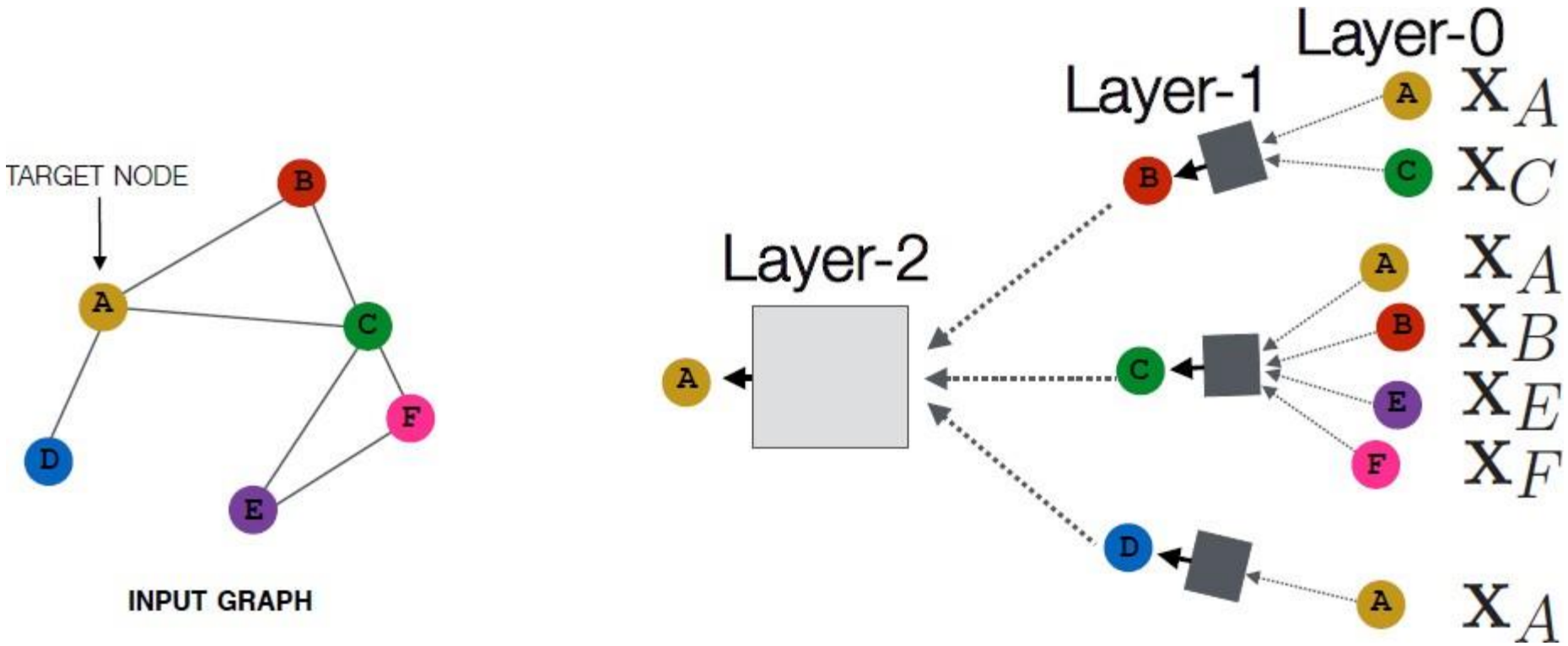
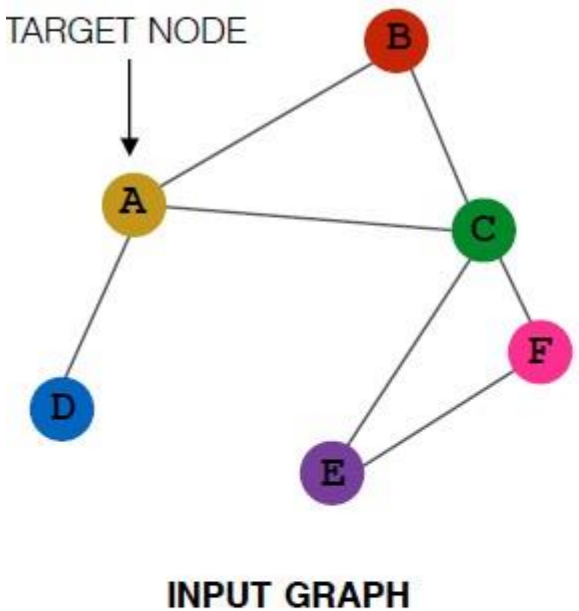
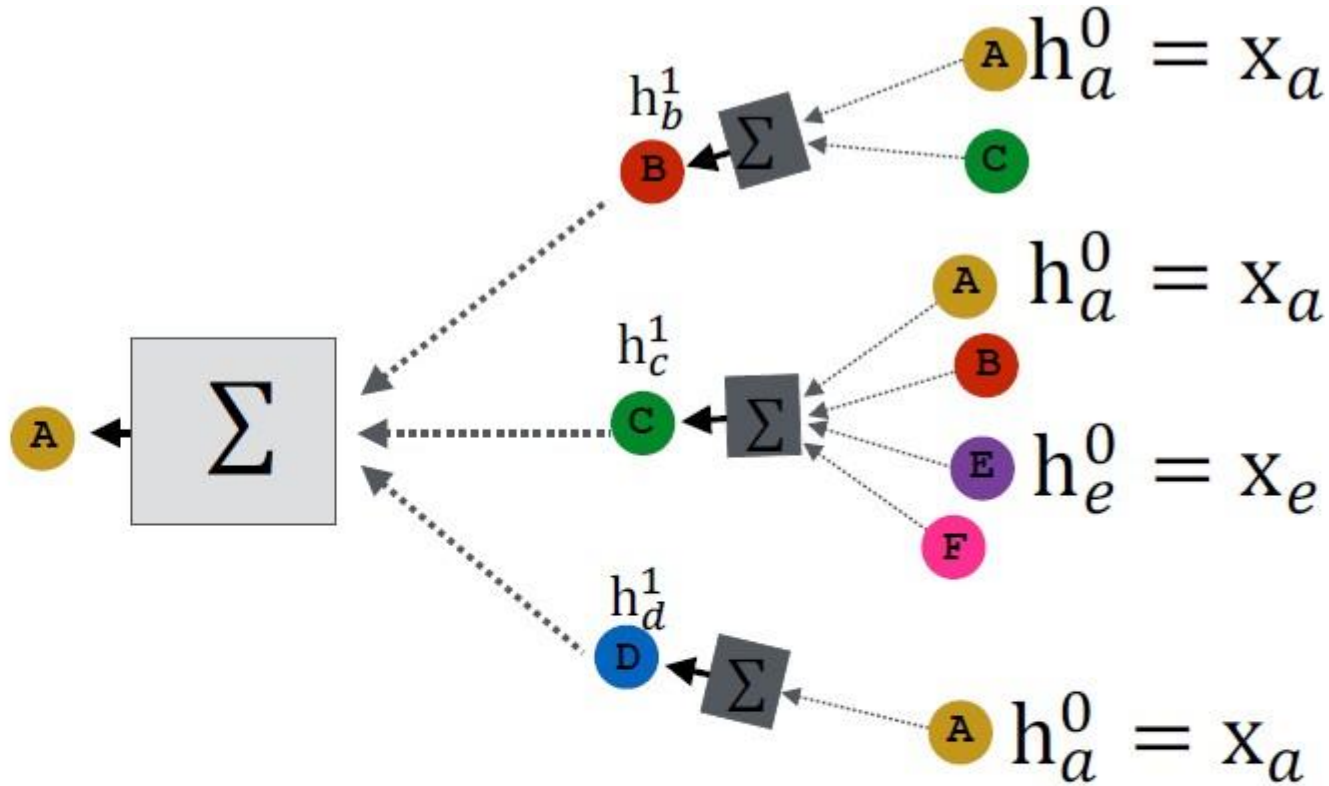


Image source

Graph Neural Networks



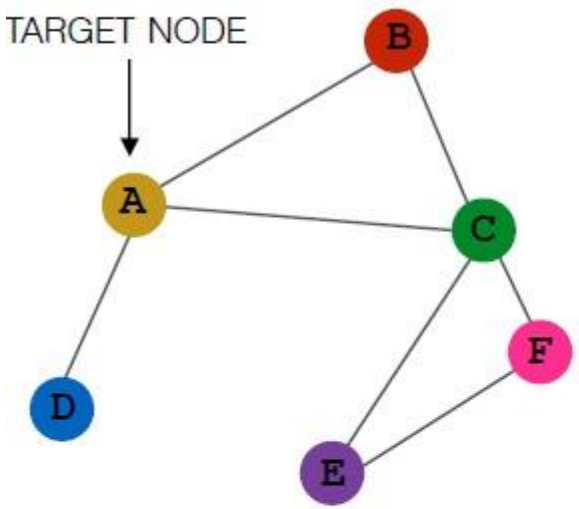
INPUT GRAPH



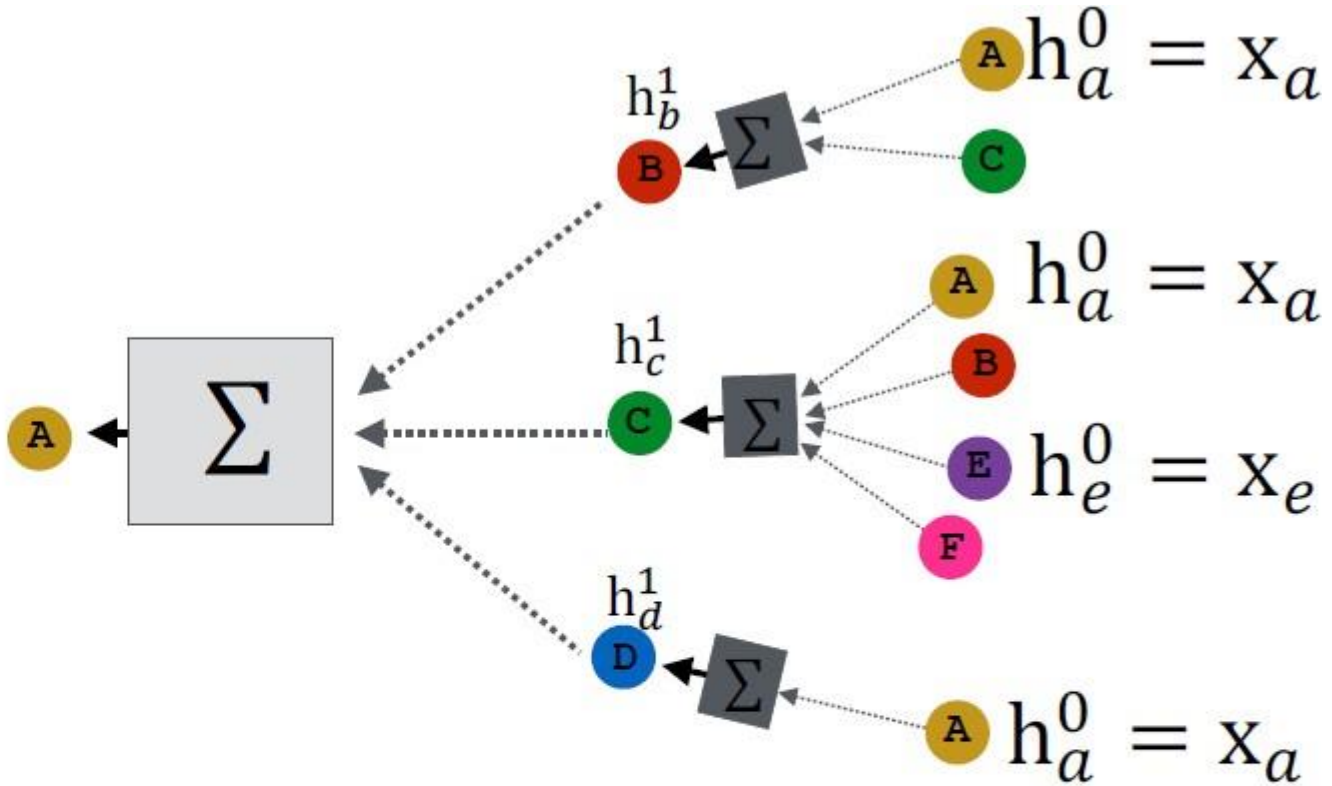
$$h_v^{(l+1)} = \sigma\left(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}\right)$$

Image source

Graph Neural Networks



INPUT GRAPH



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

Learnable parameters

[Image source]

The *same* aggregation parameters are shared for all nodes

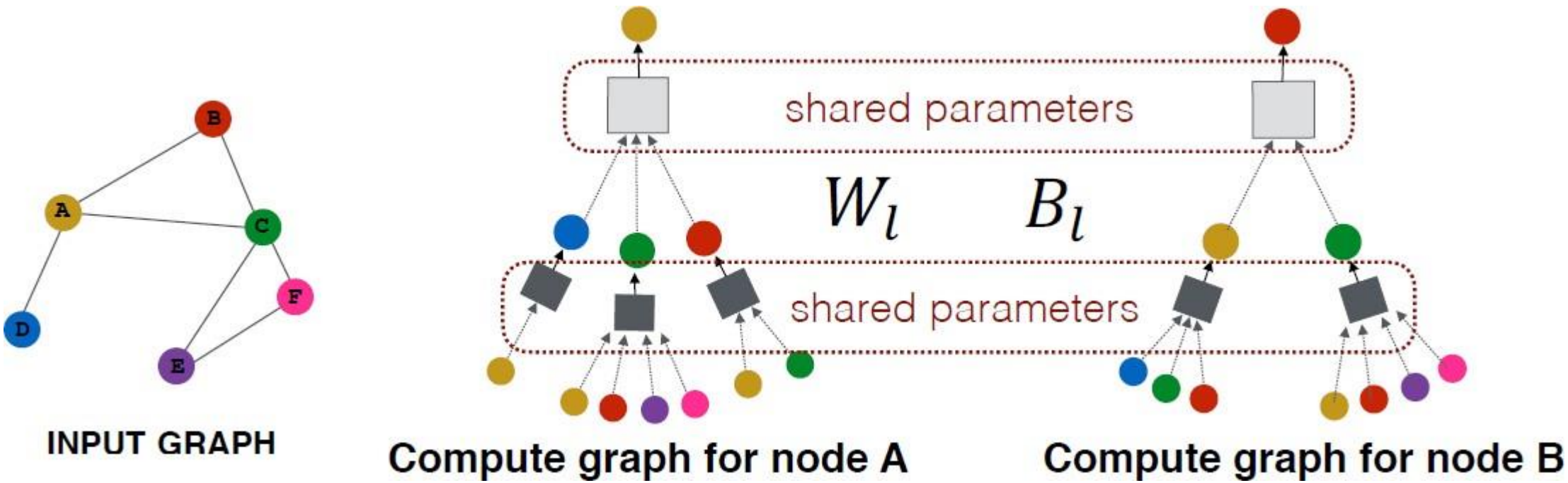


Image source

Graph Neural Networks

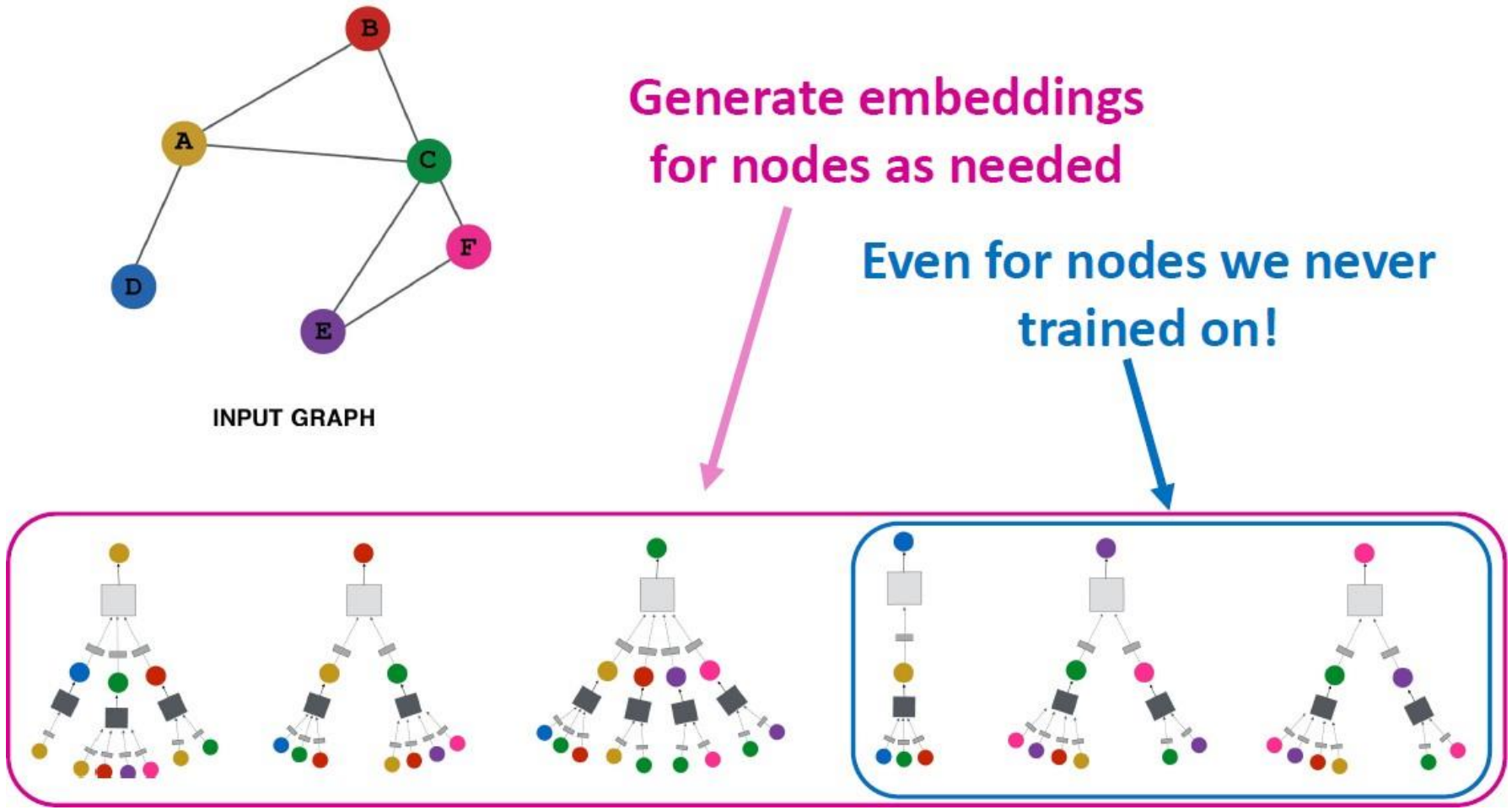


Image source

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

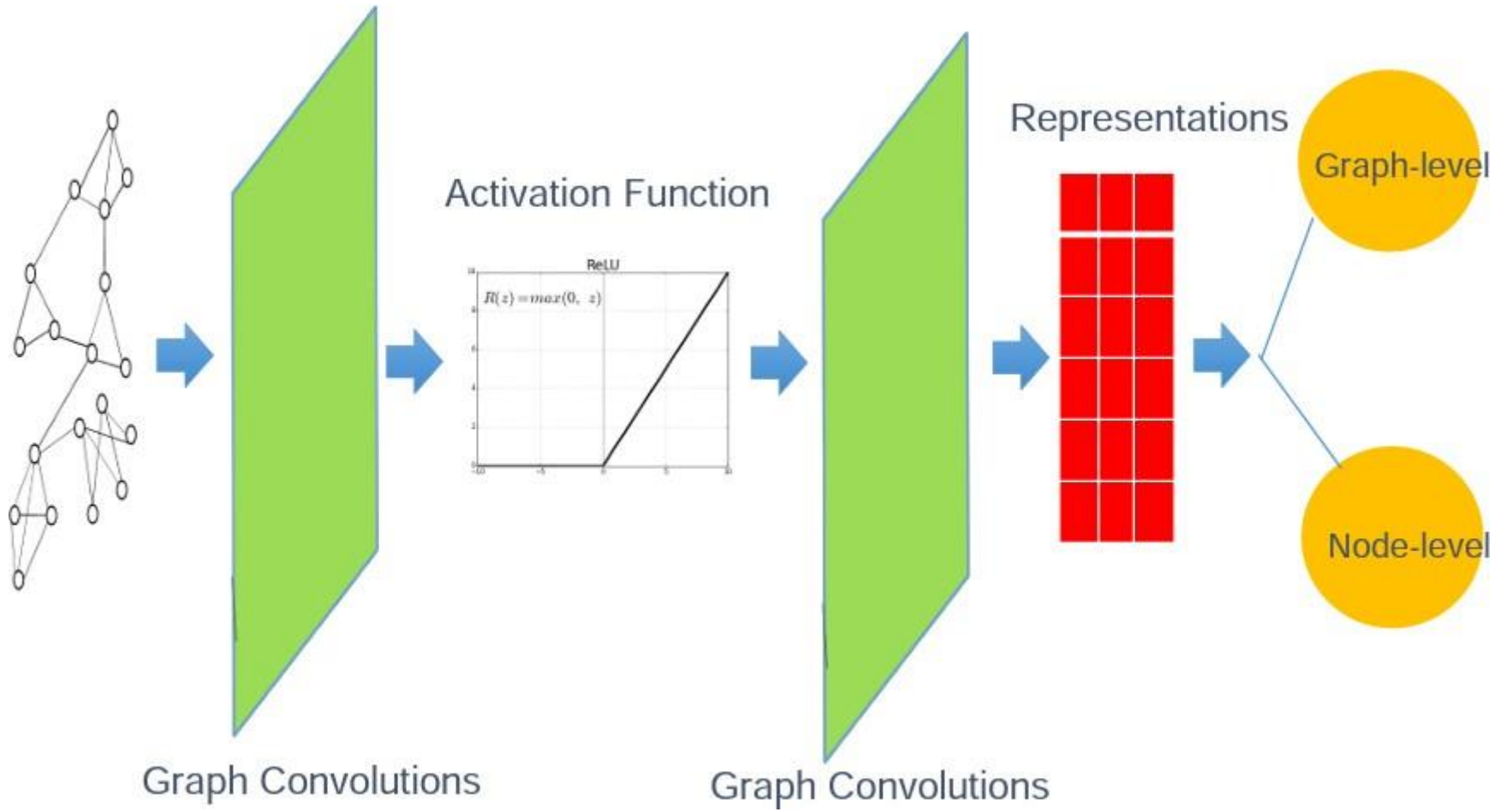


Image source

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

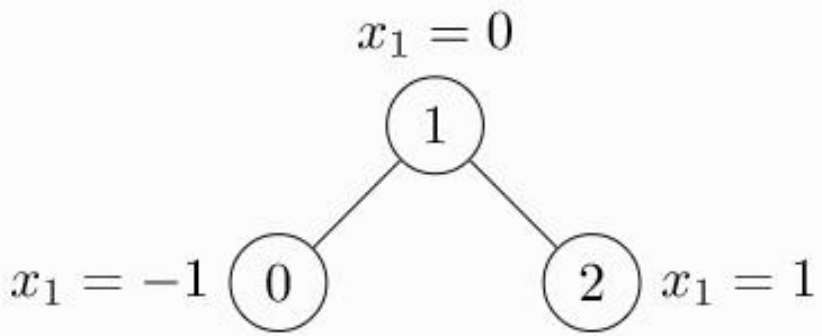
```
import torch
from torch_geometric.data import Data

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



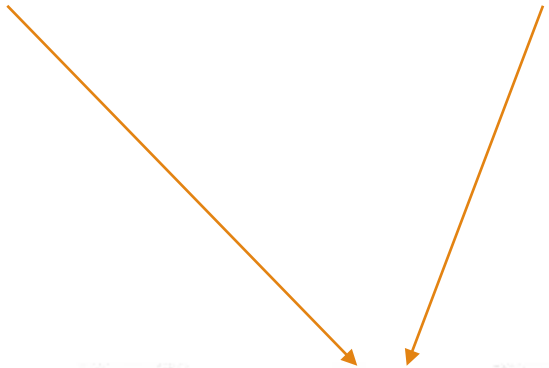
Although the graph has only two edges, we need to define four index tuples to account for both directions of an edge

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
```

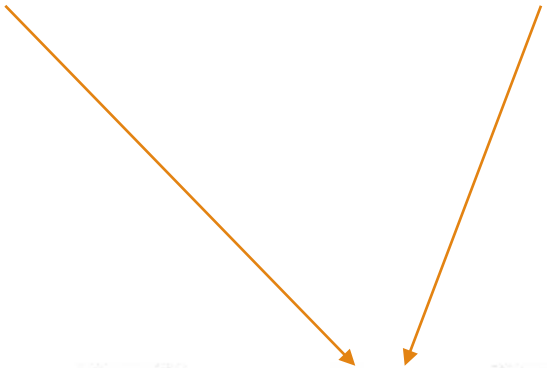


$$h_v^{k+1} = \sigma(W_k \sum_{u \in N(u)} \frac{h_u^k}{|N(v)|})$$

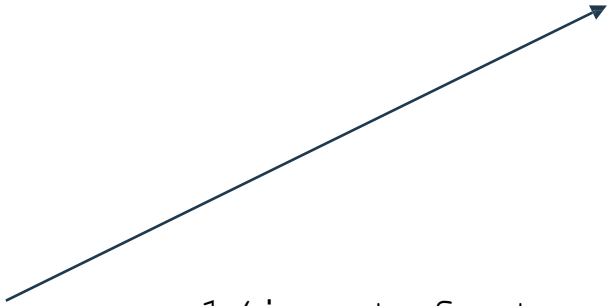
Pytorch Geometric

```
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
```



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

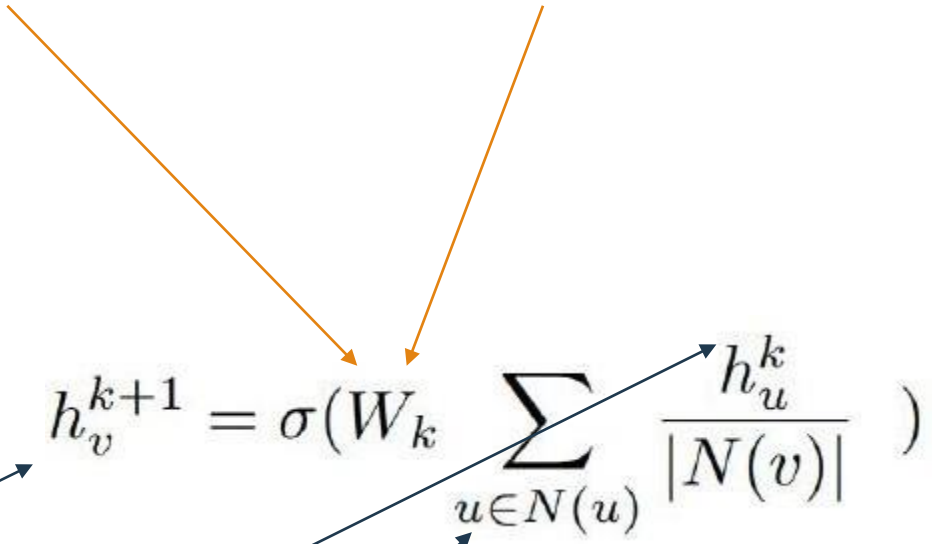


```
h = conv1(input_features, graph_edges)      # running the GNN on an input
```

Pytorch Geometric

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

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



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

```
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

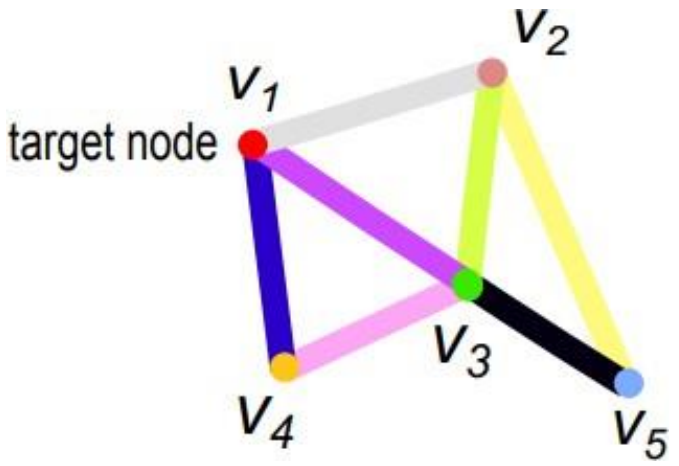
Pytorch Geometric

```
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 $h_i^{(0)} \in \mathbb{R}^{l_0}$.

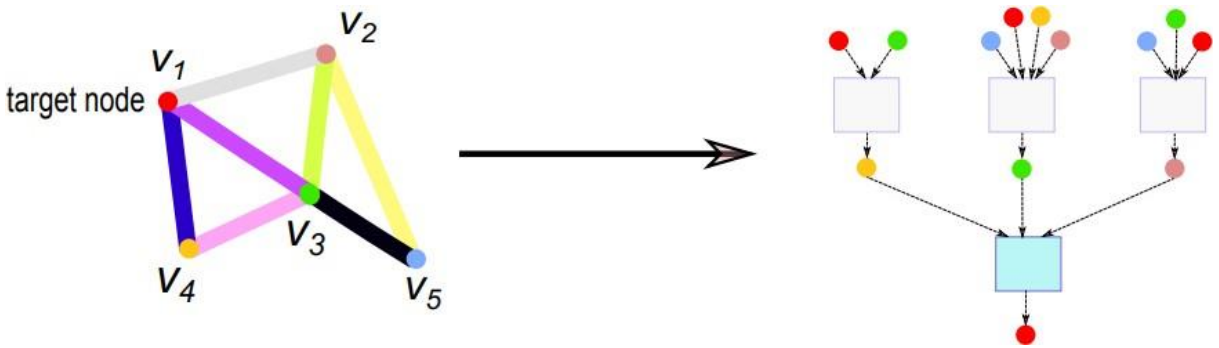


Message Passing Neural Networks

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 $0 \leq k \leq L$ by:

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

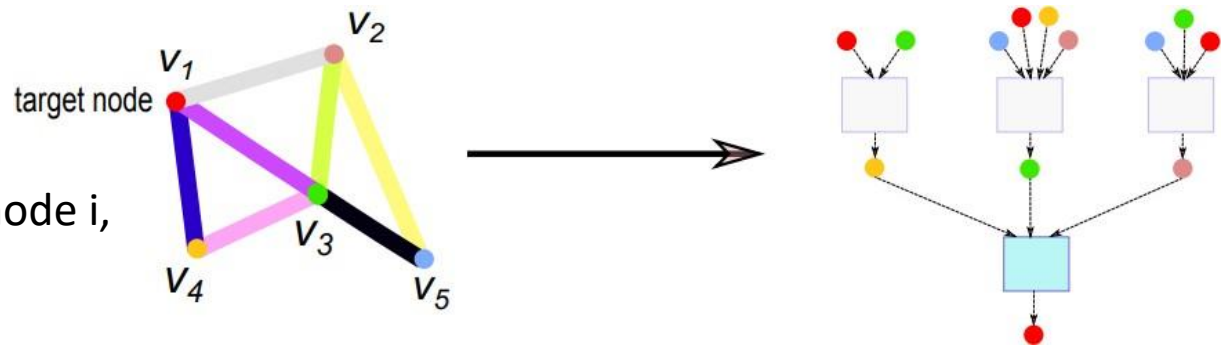


Message Passing Neural Networks

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:

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

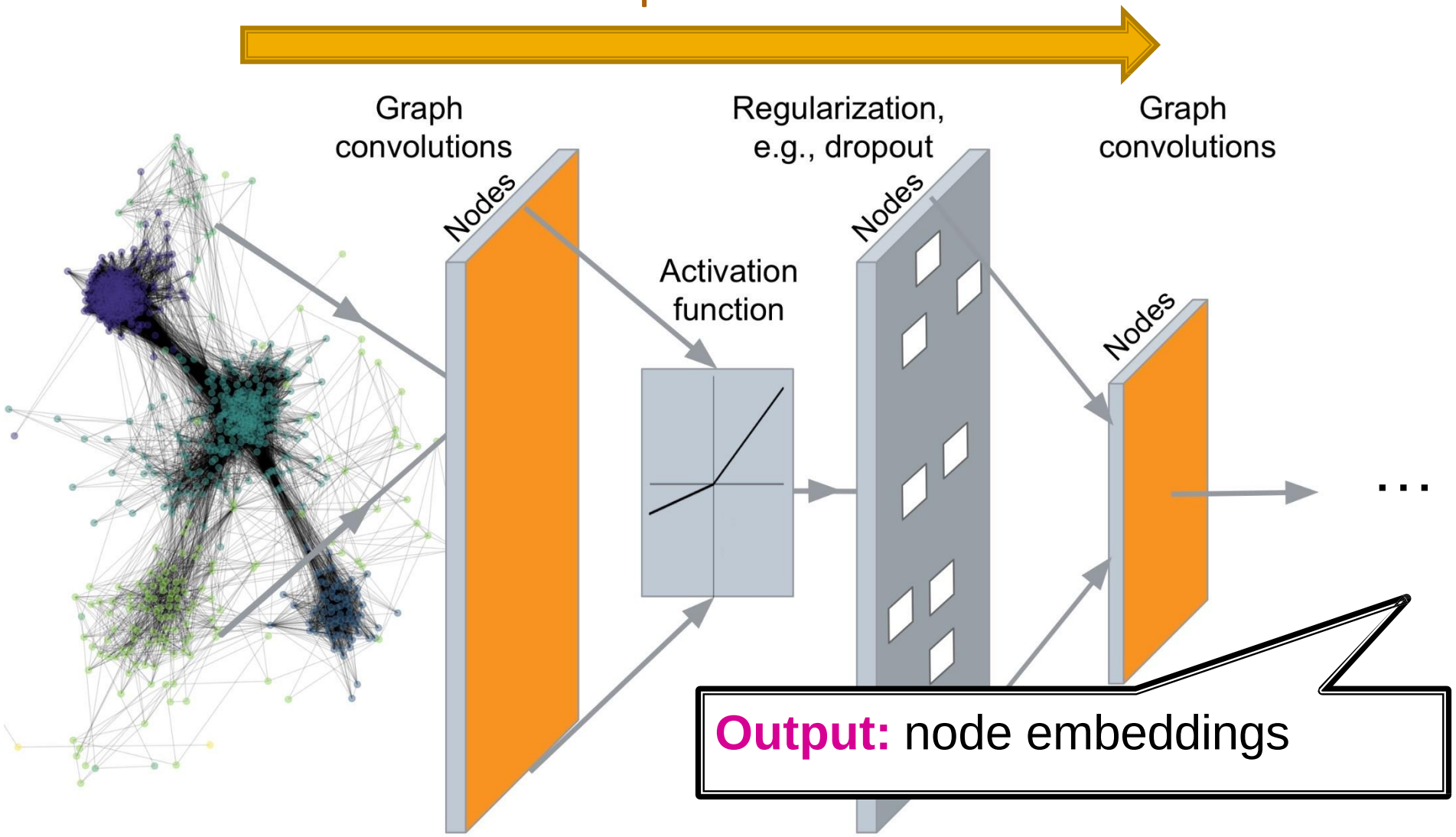


where $e_{ij} \in R^D$ is an edge feature from the node j to the node i,

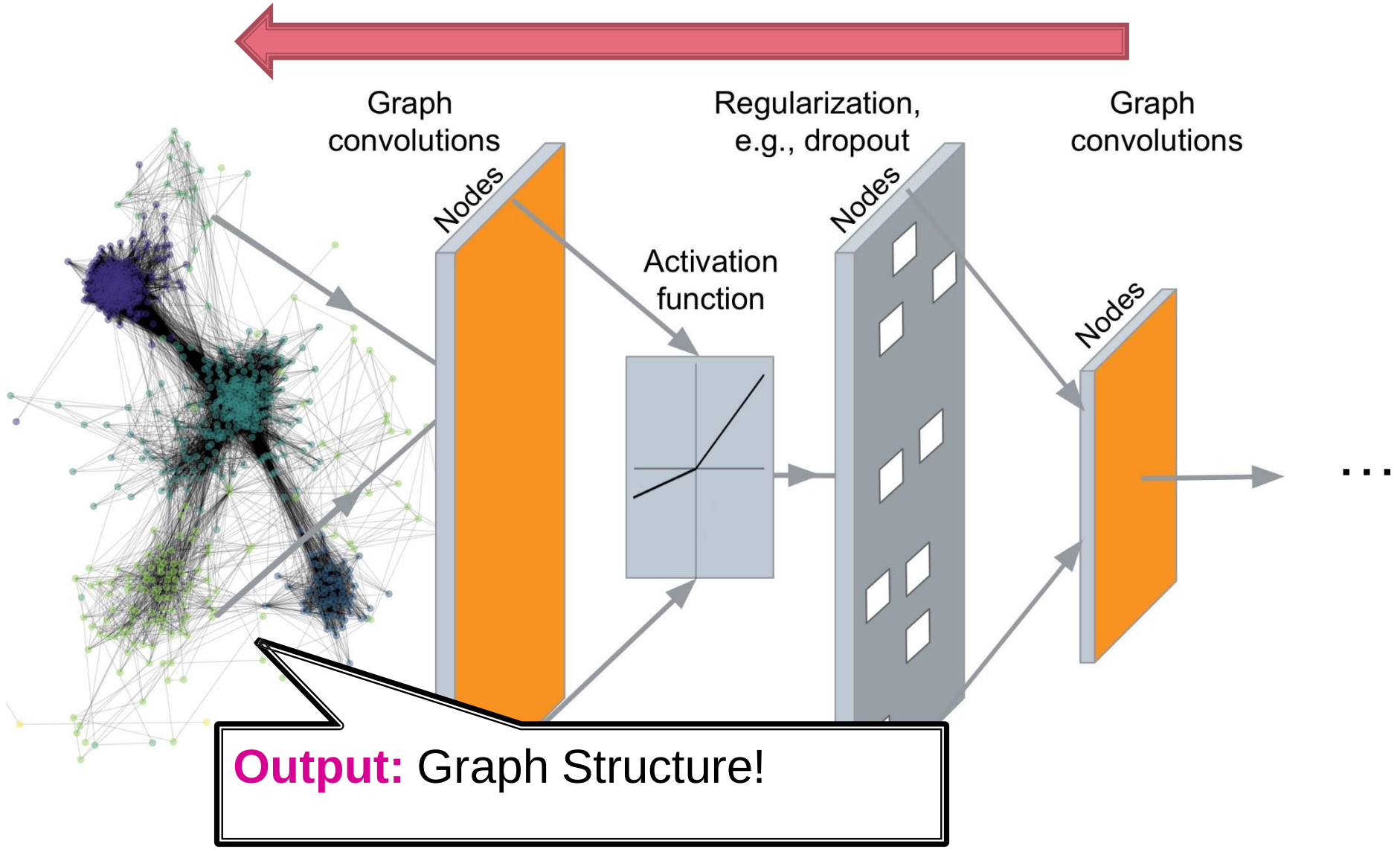
E is a permutation invariant differentiable function,

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

Graph Encoder



Graph Decoder



Graph Generation task

Task 1: Realistic graph generation

- Generate graphs that are **similar to a given set of graphs**
- **Task 2: Goal-directed graph generation**
- Generate graphs that **optimize given objectives/constraints**
  - E.g., Drug molecule generation/optimization

Graph Generative Models

- **Given:** Graphs sampled from $p_{data}(G)$
- **Goal:**
  - Learn the distribution $p_{model}(G)$
  - Sample from $p_{model}(G)$

Basics

- Assume we want to learn a generative model from a set of data points (i.e., graphs) $\{\boldsymbol{x}_i\}$
  - $p_{data}(\boldsymbol{x})$ is the data distribution, which is never known to us, but we have sampled $\boldsymbol{x}_i \sim p_{data}(\boldsymbol{x})$
  - $p_{model}(\boldsymbol{x}; \theta)$ is the model, parametrized by θ, that we use to approximate $p_{data}(\boldsymbol{x})$
- **Goal:**
  - **(1)** Make $p_{model}(\boldsymbol{x}; \theta)$ close to $p_{data}(\boldsymbol{x})$ (**Density estimation**)
  - **(2)** Make sure we can sample from $p_{model}(\boldsymbol{x}; \theta)$ (**Sampling**)
    - We need to generate examples (graphs) from $p_{model}(\boldsymbol{x}; \theta)$

(1) Make $p_{model}(x; \theta)$ close to $p_{data}(x)$

- **Key Principle**: **Maximum Likelihood**
- Fundamental approach to modeling distributions

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{x \sim p_{\text{data}}} \log p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{\theta})$$

  - Find parameters θ^*, such that for observed data points $\boldsymbol{x}_i \sim p_{data}$ the $\sum_i \log p_{model}(\boldsymbol{x}_i; \theta^*)$ has the highest value, among all possible choices of θ
    - That is, find the model that is most likely to have generated the observed data x

Non-probabilistic graph auto-encoder (GAE) model

we calculate embeddings Z and the reconstructed adjacency matrix $\hat{\mathbf{A}}$ as follows:

$$\hat{\mathbf{A}} = \sigma\left(\mathbf{Z}\mathbf{Z}^{\top}\right), \quad \text{with} \quad \mathbf{Z} = \mathrm{GCN}(\mathbf{X}, \mathbf{A})$$

Probabilistic graph auto-encoder (GAE) model

The inference model

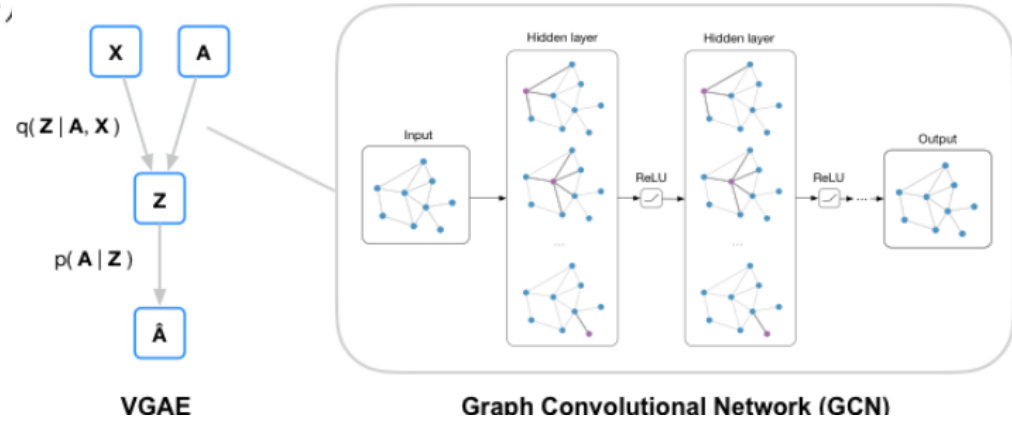
$$q(\mathbf{Z} \mid \mathbf{X}, \mathbf{A}) = \prod_{i=1}^{N} q(\mathbf{z}_i \mid \mathbf{X}, \mathbf{A}), \quad \text{with} \quad q(\mathbf{z}_i \mid \mathbf{X}, \mathbf{A}) = \mathcal{N}(\mathbf{z}_i \mid \boldsymbol{\mu}_i, \mathrm{diag}(\boldsymbol{\sigma}_i^2)).$$

Here

$$\boldsymbol{\mu} = \mathrm{GCN}_{\boldsymbol{\mu}}(\mathbf{X}, \mathbf{A}) \qquad \log \boldsymbol{\sigma} = \mathrm{GCN}_{\boldsymbol{\sigma}}(\mathbf{X}, \mathbf{A})$$

Where

$$\mathrm{GCN}(\mathbf{X}, \mathbf{A}) = \tilde{\mathbf{A}}\,\mathrm{ReLU}\big(\tilde{\mathbf{A}}\mathbf{X}\mathbf{W}_0\big)\mathbf{W}_1,$$



Here also recall :

$\tilde{\mathbf{A}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$ is the symmetrically normalized adjacency matrix.

Probabilistic graph auto-encoder (GAE) model

Generative model

$$p\left(\mathbf{A} \mid \mathbf{Z}\right) = \prod_{i=1}^{N} \prod_{j=1}^{N} p\left(A_{ij} \mid \mathbf{z}_i, \mathbf{z}_j\right) \ , \quad \text{with} \quad p\left(A_{ij} = 1 \mid \mathbf{z}_i, \mathbf{z}_j\right) = \sigma(\mathbf{z}_i^\top \mathbf{z}_j)$$

Probabilistic graph auto-encoder (GAE) model

Generative model

$$p\left(\mathbf{A} \mid \mathbf{Z}\right) = \prod_{i=1}^{N} \prod_{j=1}^{N} p\left(A_{ij} \mid \mathbf{z}_i, \mathbf{z}_j\right) \ , \quad \text{with} \quad p\left(A_{ij} = 1 \mid \mathbf{z}_i, \mathbf{z}_j\right) = \sigma(\mathbf{z}_i^\top \mathbf{z}_j)$$

Refs

12-deep-generation.pdf (stanford.edu)