Graph attention Networks (GAT)

Antonio Longa[1,2]

MobS[1] Lab, Fondazione Bruno Kessler,Trento, Italy
SML[2] Lab, University of Trento, Italy

TABLE OF CONTENTS

01 Recap

02 Introduction

03 Graph attention layer (GAT)

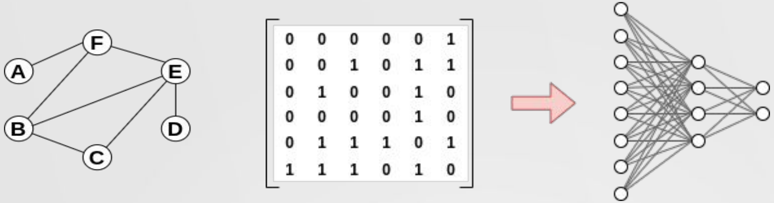
04 Pros of GAT

05 Message passing Implementation

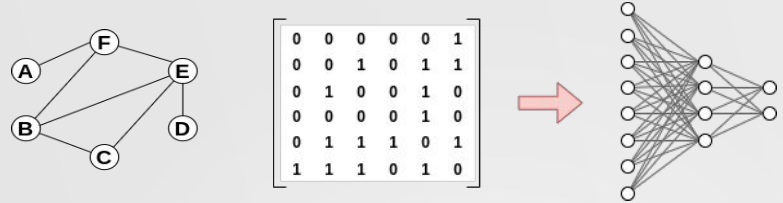
06 Implement our GCNConv

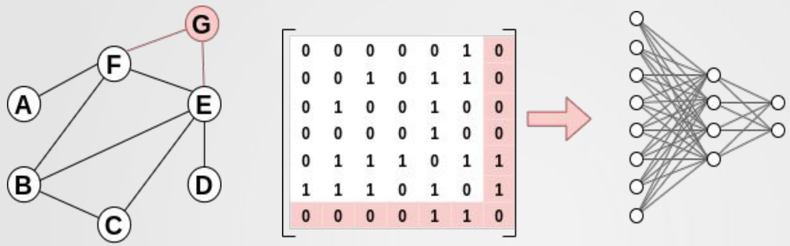
07 GAT implementation

01 Recap



01 Recap

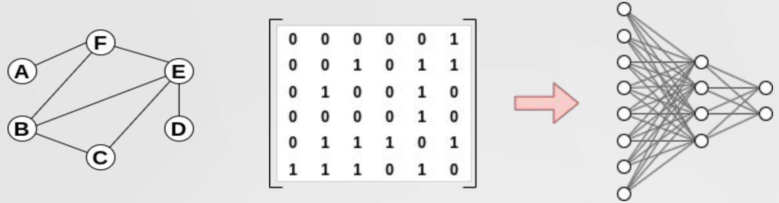


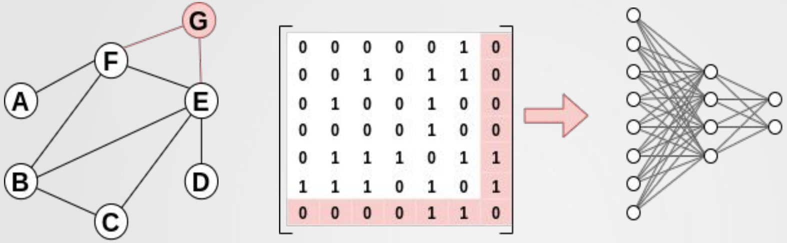


PROBLEMS:

- Different sizes

01 Recap





PROBLEMS:

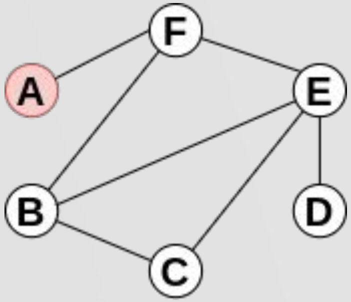
- Different sizes

- NOT invariant to nodes ordering

COMPUTATION GRAPH

The neighbour of a node defines its computation graph

INPUT GRAPH

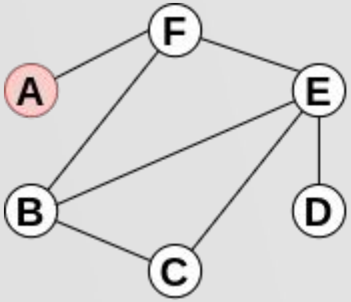


01 Recap

COMPUTATION GRAPH

The neighbour of a node defines its computation graph

INPUT GRAPH



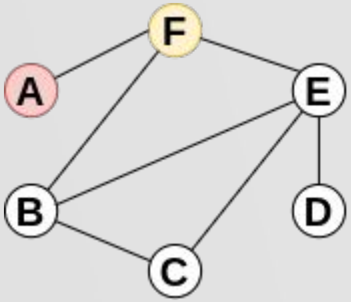
COMPUTATION GRAPH

01 Recap

COMPUTATION GRAPH

The neighbour of a node defines its computation graph

INPUT GRAPH



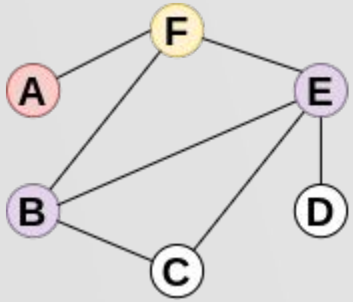
COMPUTATION GRAPH

01 Recap

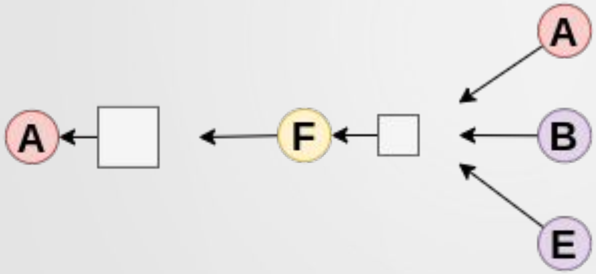
COMPUTATION GRAPH

The neighbour of a node defines its computation graph

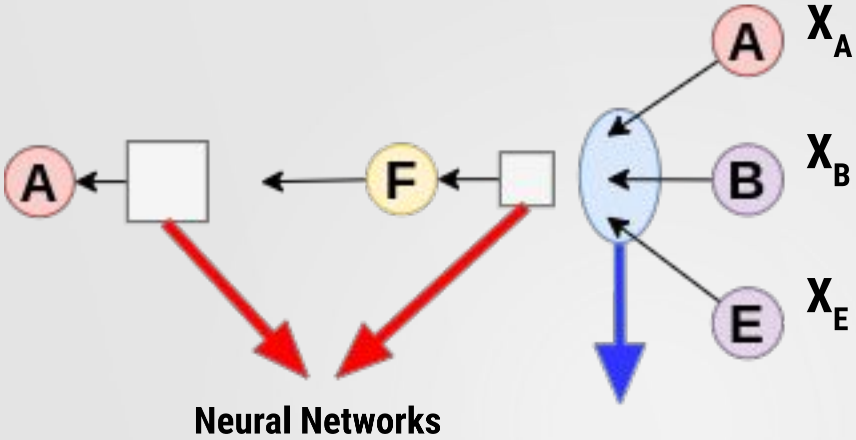
INPUT GRAPH



COMPUTATION GRAPH



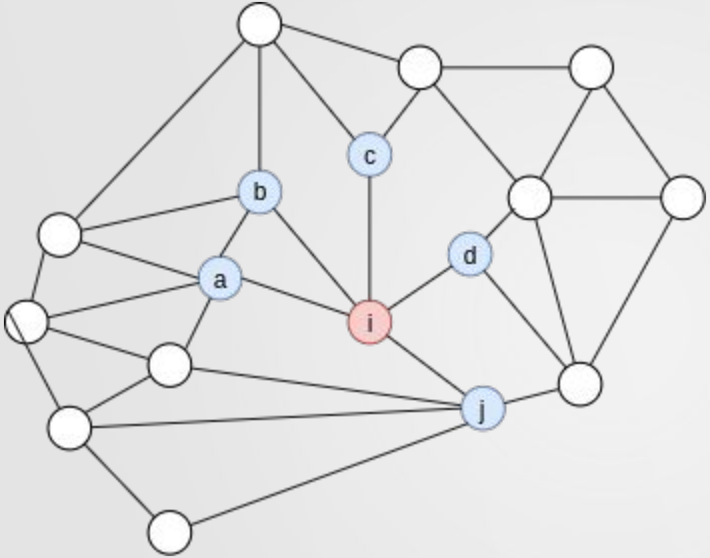
01 Recap



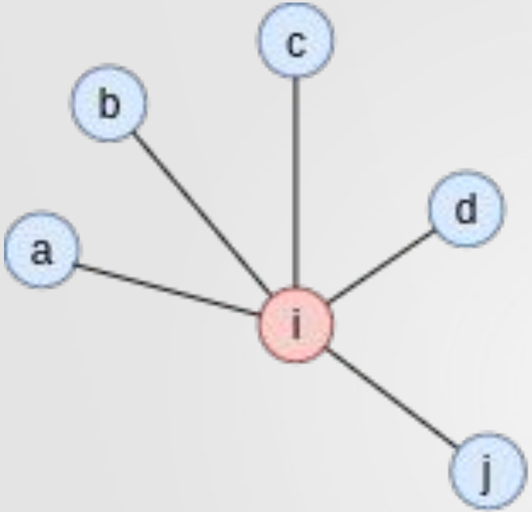
<u>Ordering invariant</u>
Aggregation

Sum
Average

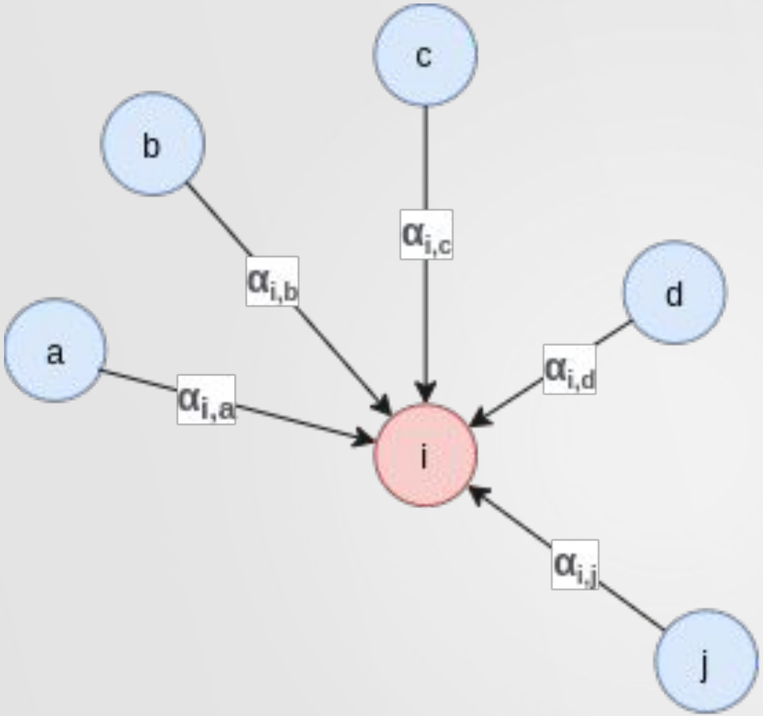
02 Introduction



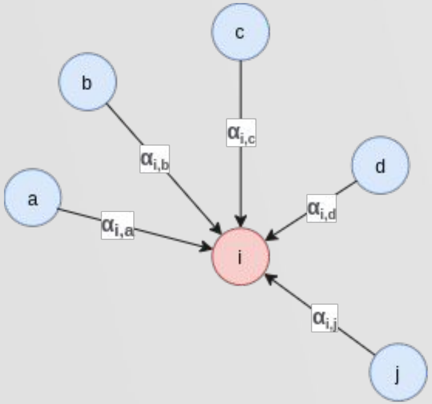
02 Introduction



02 Introduction

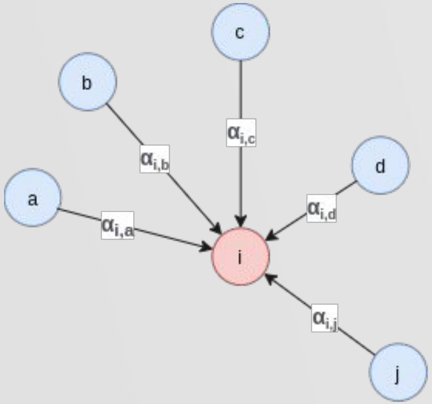


02 Introduction



How much features of node “c” are important to node “i”?

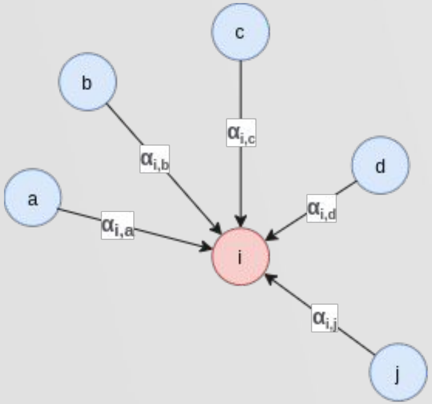
02 Introduction



How much features of node “c” are important to node “i”?

Can we learn such importance, in an automatic manner?

02 Introduction



How much features of node “c” are important to node “i”?

Can we learn such importance, in an automatic manner?

YES, with GAT

03 Graph Attention Networks GAT

Published as a conference paper at ICLR 2018

GRAPH ATTENTION NETWORKS

Petar Veličković*
Department of Computer Science and Technology
University of Cambridge
petar.velickovic@cst.cam.ac.uk

Guillem Cucurull*
Centre de Visió per Computador, UAB
gcucurull@gmail.com

Arantxa Casanova*
Centre de Visió per Computador, UAB
ar.casanova.8@gmail.com

Adriana Romero
Montréal Institute for Learning Algorithms
adriana.romero.soriano@umontreal.ca

Pietro Liò
Department of Computer Science and Technology
University of Cambridge
pietro.lio@cst.cam.ac.uk

Yoshua Bengio
Montréal Institute for Learning Algorithms
yoshua.umontreal@gmail.com

Petar Veličković

Senior Research Scientist at DeepMind

03 Graph Attention layer

INPUT: a set of node features $\mathbf{h} = \{\bar{h}_1, \bar{h}_2, \ldots, \bar{h}_n\}$ $\bar{h}_i \in \mathbf{R}^F$

OUTPUT: a **new** set of node features $\mathbf{h'} = \{\bar{h}'_1, \bar{h}'_2, \ldots, \bar{h}'_n\}$ $\bar{h}'_i \in \mathbf{R}^{F'}$

03 Graph Attention layer

1) apply a **parameterized linear transformation** to every node

$$\mathbf{W} \cdot \bar{h}_i \qquad \mathbf{W} \in \mathbf{R}^{F' \times F}$$

03 Graph Attention layer

1) apply a **parameterized linear transformation** to every node

$$\mathbf{W} \cdot \bar{h}_i \qquad \mathbf{W} \in \mathbf{R}^{F' \times F}$$

$$(F' \times F) \cdot F$$

03 Graph Attention layer

1) apply a **parameterized linear transformation** to every node

$$\mathbf{W} \cdot \bar{h}_i \qquad \mathbf{W} \in \mathbf{R}^{F' \times F}$$

$$(F' \cancel{\times F)} \cancel{\cdot F}$$

$$F'$$

03 Graph Attention layer

2) Self attention

$$a : \mathbf{R}^{F'} \times \mathbf{R}^{F'} \rightarrow \mathbf{R}$$

03 Graph Attention layer

2) Self attention

$$a : \mathbf{R}^{F'} \times \mathbf{R}^{F'} \rightarrow \mathbf{R}$$

$$e_{i,j} = a(\mathbf{W} \cdot \bar{h}_i , \mathbf{W} \cdot \bar{h}_j)$$

03 Graph Attention layer

2) Self attention

$$a : \mathbf{R}^{F'} \times \mathbf{R}^{F'} \rightarrow \mathbf{R}$$

$$e_{i,j} = a(\mathbf{W} \cdot \bar{h}_i , \mathbf{W} \cdot \bar{h}_j)$$

Specify the importance of node j's features to node i

03 Graph Attention layer

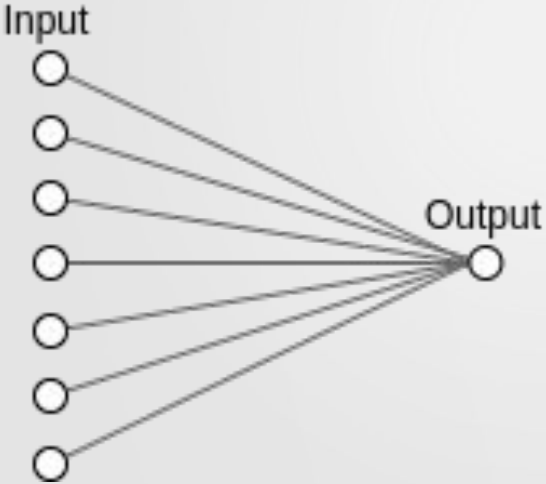
3) Normalization

$$\alpha_{i,j} = softmax_j(e_{i,j}) = \frac{exp(e_{i,j})}{\sum_{k\in N(i)} exp(e_{i,k})}$$

03 Graph Attention layer

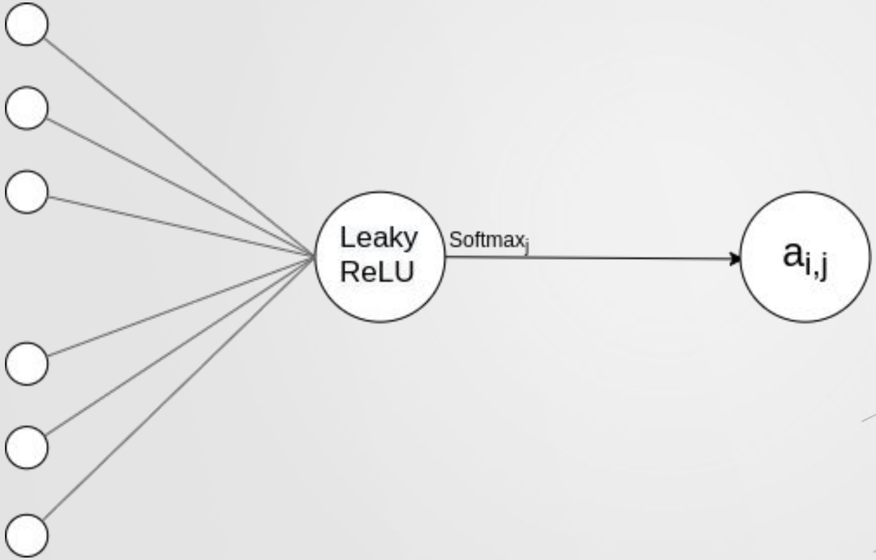
4) Attention mechanism a

Is a single-layer feed forward neural network



03 Graph Attention layer

4) Attention mechanism a

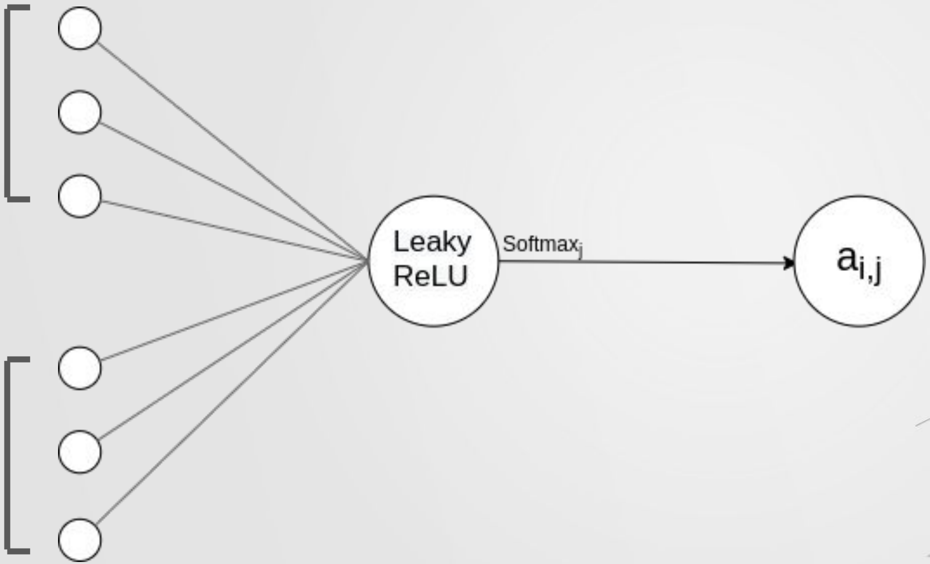


03 Graph Attention layer

4) Attention mechanism a

$$\mathbf{W}h_i \rightarrow h'_i$$

$$\mathbf{W}h_j \rightarrow h'_j$$



03 Graph Attention layer

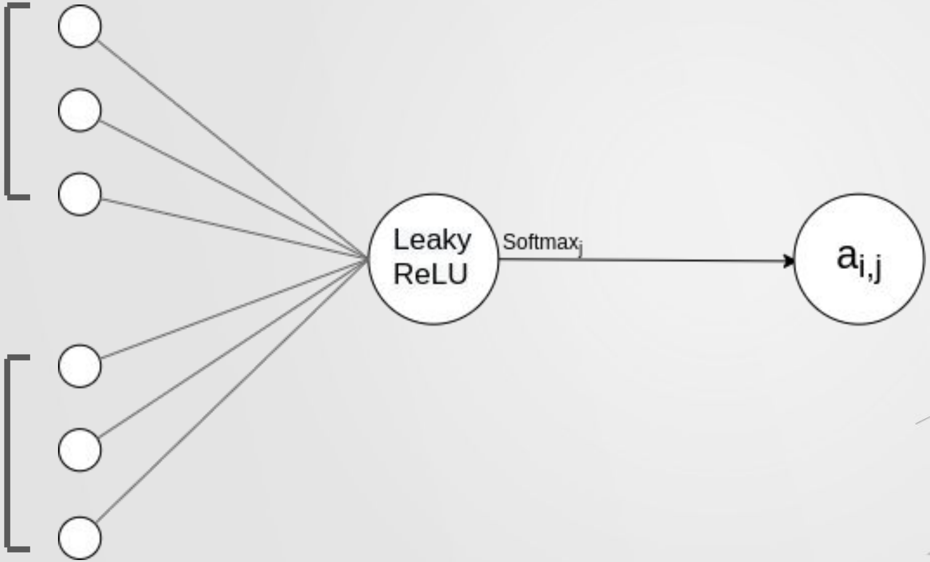
4) Attention mechanism a

$$\mathbf{W}h_i \rightarrow h'_i$$

$$(F' \times F)F \rightarrow F'$$

$$\mathbf{W}h_j \rightarrow h'_j$$

$$(F' \times F)F \rightarrow F'$$



03 Graph Attention layer

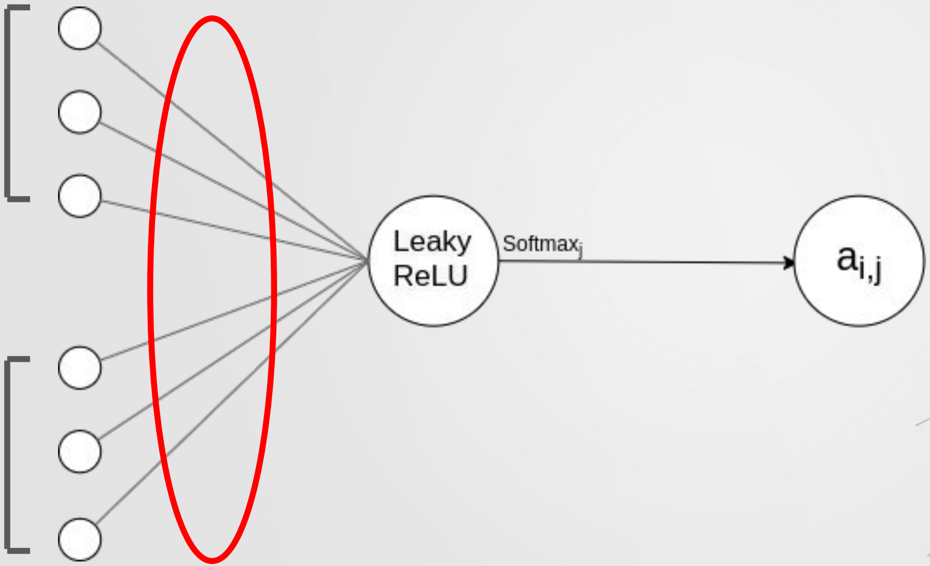
4) Attention mechanism a

$$\mathbf{W}h_i \rightarrow h'_i$$

$$(F' \times F)F \rightarrow F'$$

$$\mathbf{W}h_j \rightarrow h'_j$$

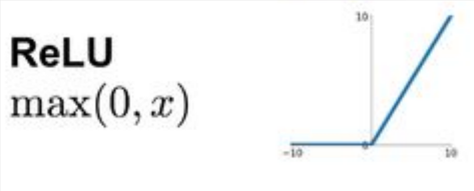
$$(F' \times F)F \rightarrow F'$$



$$\bar{a} \in \mathbf{R}^{2F'}$$

03 Graph Attention layer

4) Attention mechanism a

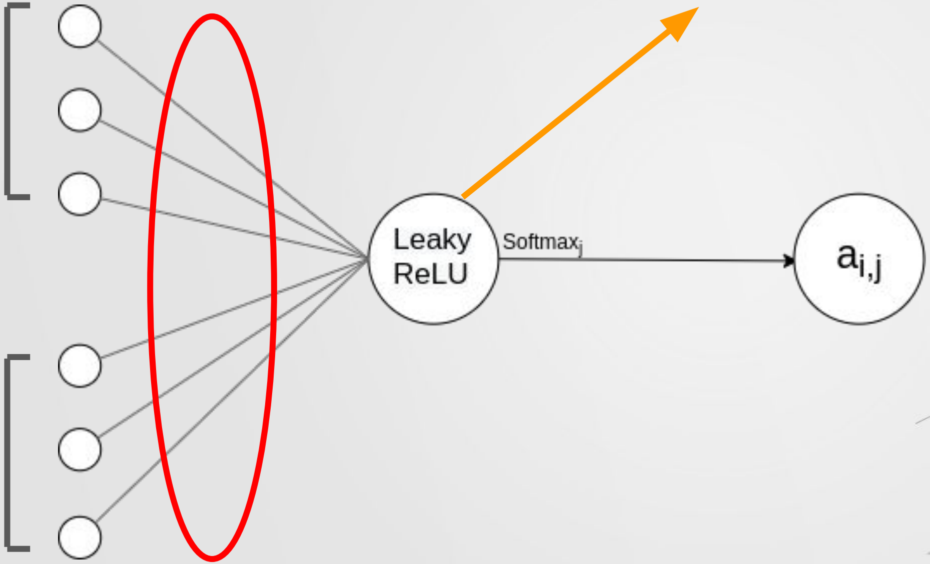


$$\mathbf{W}h_i \rightarrow h'_i$$

$$(F' \times F)F \rightarrow F'$$

$$\mathbf{W}h_j \rightarrow h'_j$$

$$(F' \times F)F \rightarrow F'$$



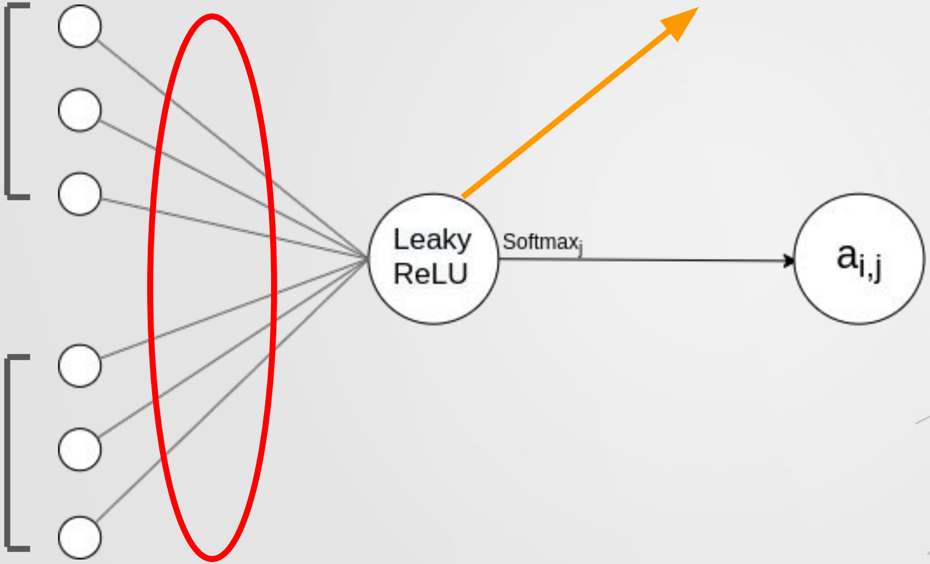
$$\bar{a} \in \mathbf{R}^{2F'}$$

03 Graph Attention layer

4) Attention mechanism a

$$\mathbf{W}h_i \rightarrow h'_i$$
$$(F' \times F)F \rightarrow F'$$

$$\mathbf{W}h_j \rightarrow h'_j$$
$$(F' \times F)F \rightarrow F'$$



$$\bar{a} \in \mathbf{R}^{2F'}$$

03 Graph Attention layer

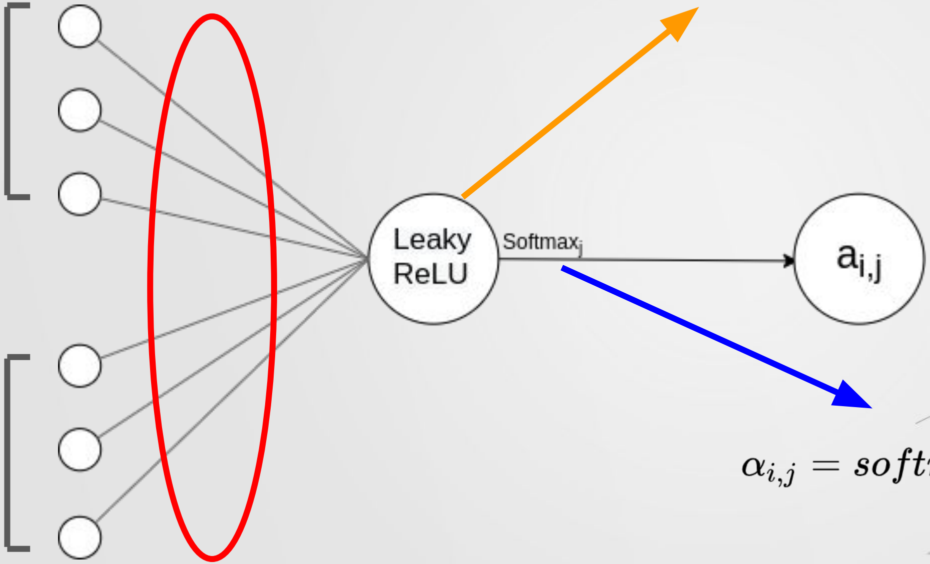
4) Attention mechanism a

$$\mathbf{W}h_i \rightarrow h'_i$$

$$(F' \times F)F \rightarrow F'$$

$$\mathbf{W}h_j \rightarrow h'_j$$

$$(F' \times F)F \rightarrow F'$$



$$\bar{a} \in \mathbf{R}^{2F'}$$

$$\alpha_{i,j} = softmax_j(e_{i,j}) = \frac{exp(e_{i,j})}{\sum_{k \in N(i)} exp(e_{i,k})}$$

03 Graph Attention layer

4) Attention mechanism a

$$\alpha_{i,j} = \frac{exp(LeakyReLU(\bar{a}^T[\overbrace{\mathbf{W}h_i}^{(F' \times F)F}||\overbrace{\mathbf{W}h_j}^{(F' \times F)F}]))}{\sum_{k \in N(i)} exp(LeakyReLU(\bar{a}^T[\mathbf{W}h_i||\mathbf{W}h_k]))}$$

$\bar{a}^T \rightarrow transpose(a)$

$|| \rightarrow concatenation$

03 Graph Attention layer

4) Attention mechanism a

$$\alpha_{i,j} = \frac{exp(LeakyReLU(\bar{a}^T [\overbrace{\mathbf{W}h_i}^{\substack{F' \\ (F' \times F)F}} || \overbrace{\mathbf{W}h_j}^{\substack{F' \\ (F' \times F)F}}]))}{\sum_{k \in N(i)} exp(LeakyReLU(\bar{a}^T [\mathbf{W}h_i || \mathbf{W}h_k]))}$$

$\bar{a}^T \rightarrow transpose(a)$

$|| \rightarrow concatenation$

03 Graph Attention layer

4) Attention mechanism a

$$\alpha_{i,j} = \frac{exp(LeakyReLU(\bar{a}^T \overbrace{[\underbrace{\mathbf{W}h_i}_{\substack{F' \\ (F' \times F)F}} || \underbrace{\mathbf{W}h_j}_{\substack{F' \\ (F' \times F)F}}]}^{(2F' \times 1)}))}{\sum_{k \in N(i)} exp(LeakyReLU(\bar{a}^T [\mathbf{W}h_i || \mathbf{W}h_k]))}$$

$\bar{a}^T \rightarrow transpose(a)$

$|| \rightarrow concatenation$

03 Graph Attention layer

4) Attention mechanism a

$$\alpha_{i,j} = \frac{exp(LeakyReLU(\overbrace{\bar{a}^T}^{(1 \times 2F')} \overbrace{[\overbrace{\overbrace{\mathbf{W}h_i}^{(F' \times F)F}}^{F'} || \overbrace{\overbrace{\mathbf{W}h_j}^{(F' \times F)F}}^{F'}]}^{(2F' \times 1)}))}{\sum_{k \in N(i)} exp(LeakyReLU(\bar{a}^T [\mathbf{W}h_i || \mathbf{W}h_k]))}$$

$\bar{a}^T \rightarrow transpose(a)$

$|| \rightarrow concatenation$

03 Graph Attention layer

4) Attention mechanism a

$$\alpha_{i,j} = \frac{exp(LeakyReLU(\overbrace{\overbrace{\bar{a}^T}^{(1 \times 2F')} \overbrace{[\overbrace{\overbrace{\mathbf{W}h_i}^{(F' \times F)F}}^{F'} || \overbrace{\overbrace{\mathbf{W}h_j}^{(F' \times F)F}}^{F'}]}^{(2F' \times 1)}}^{1}))}{\sum_{k \in N(i)} exp(LeakyReLU(\bar{a}^T [\mathbf{W}h_i || \mathbf{W}h_k]))}$$

$\bar{a}^T \rightarrow transpose(a)$

$|| \rightarrow concatenation$

03 Graph Attention layer

4) Attention mechanism a

$$\alpha_{i,j} = \frac{\overbrace{exp(LeakyReLU(\overbrace{\underbrace{}_{}\overbrace{\bar{a}^T}^{(1\times 2F')}\overbrace{[\overbrace{\overbrace{\mathbf{W}h_i}^{(F'\times F)F}}^{F'}||\overbrace{\overbrace{\mathbf{W}h_j}^{(F'\times F)F}}^{F'}]}^{(2F'\times 1)}}^{1}))}^{1}}{\sum_{k\in N(i)} exp(LeakyReLU(\bar{a}^T[\mathbf{W}h_i||\mathbf{W}h_k]))}$$

$\bar{a}^T \rightarrow transpose(a)$

$|| \rightarrow concatenation$

03 Graph Attention layer

4) Attention mechanism a

$$\alpha_{i,j} = \frac{\overbrace{exp(LeakyReLU(\overbrace{\overbrace{\bar{a}^T}^{(1 \times 2F')}\overbrace{[\overbrace{\overbrace{\mathbf{W}h_i}^{(F' \times F)F}}^{F'}||\overbrace{\overbrace{\mathbf{W}h_j}^{(F' \times F)F}}^{F'}]}^{(2F' \times 1)}}^{1}))}^{1}}{\sum_{k \in N(i)} exp(LeakyReLU(\bar{a}^T[\mathbf{W}h_i||\mathbf{W}h_k]))} = \frac{\mathbf{R}}{\mathbf{R}} = \mathbf{R}$$

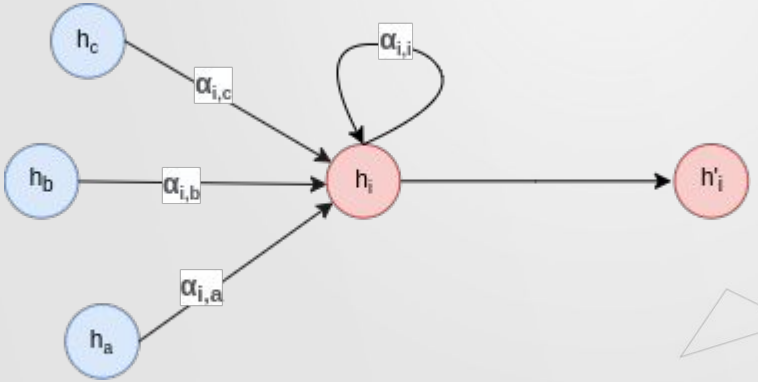
$\bar{a}^T \rightarrow transpose(a)$

$|| \rightarrow concatenation$

03 Graph Attention layer

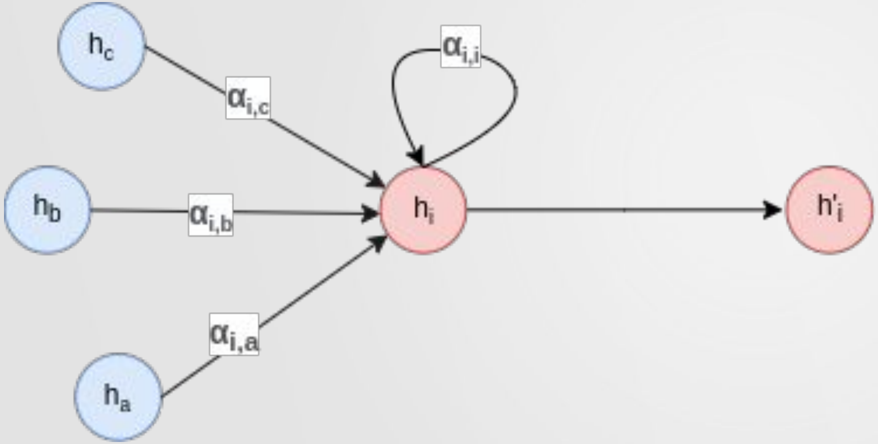
5) Use it :)

$$h'_i = \sigma(\sum_{j \in N(i)} \alpha_{i,j} \mathbf{W} h_j)$$



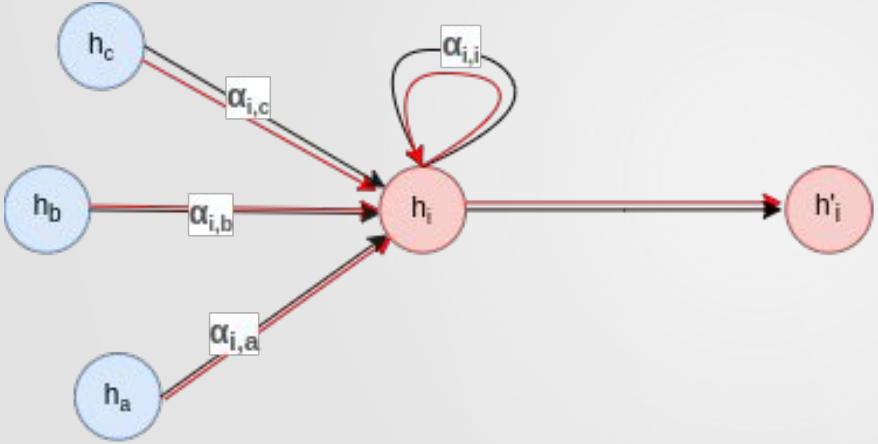
03 Graph Attention layer

6) Multi-head attention



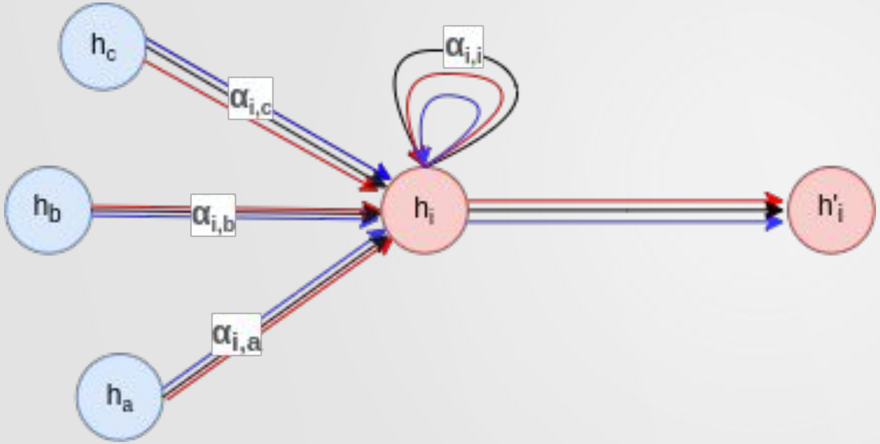
03 Graph Attention layer

6) Multi-head attention



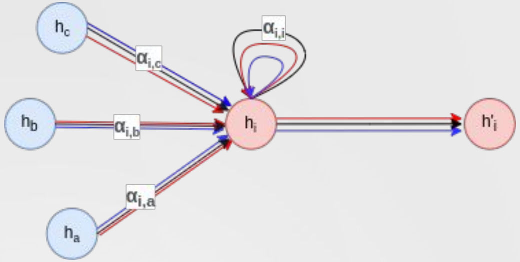
03 Graph Attention layer

6) Multi-head attention



03 Graph Attention layer

6) Multi-head attention



Concatenation

$$h_i' = ||_{k=1}^{K} \sigma(\sum_{j \in N(i)} \alpha_{i,j}^{k} \mathbf{W}^k h_j)$$

Average

$$h_i' = \sigma(\frac{1}{K} \sum_{k=1}^{K} \sum_{j \in N(i)} \alpha_{i,j}^{k} \mathbf{W}^k h_j)$$

- On the final (prediction) layer of the network

04 Pros of GAT

- Computationally **efficient**
  - Self-attention layers can be **parallelized across edges**
  - Output features can be **parallelized across nodes**

04 Pros of GAT

- Computationally **efficient**
  - Self-attention layers can be **parallelized across edges**
  - Output features can be **parallelized across nodes**

- Allows to assign **different importances to nodes** of a same neighborhood

04 Pros of GAT

- Computationally **efficient**
  - Self-attention layers can be **parallelized across edges**
  - Output features can be **parallelized across nodes**

- Allows to assign **different importances to nodes** of a same neighborhood

- It is applied in a **shared manner** to all edges in the graph
  - **Not** required to have the **entire graph**

04 Pros of GAT

- Computationally **efficient**
  - Self-attention layers can be **parallelized across edges**
  - Output features can be **parallelized across nodes**
- Allows to assign **different importances to nodes** of a same neighborhood
- It is applied in a **shared manner** to all edges in the graph
  - **Not** required to have the **entire graph**
- Works in **both**:
  - **Transductive** learning (Cora, Citeseer, Pubmed)
  - **Inductive** learning (PPI)

05 Message passing implementation

$$\mathbf{x}_i^{(k)} = \gamma^{(k)} \left(\mathbf{x}_i^{(k-1)}, \square_{j \in \mathcal{N}(i)} \, \phi^{(k)} \left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i} \right) \right),$$

Features representations of node i at the k-th layer

05 Message passing implementation

$$\mathbf{x}_i^{(k)} = \gamma^{(k)} \left(\mathbf{x}_i^{(k-1)}, \square_{j \in \mathcal{N}(i)} \boxed{\phi^{(k)}} \left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i} \right) \right),$$

Features representations of node i at the k-th layer

Differentiable function
Eg: MLP

05 Message passing implementation

$$\mathbf{x}_i^{(k)} = \gamma^{(k)}\left(\mathbf{x}_i^{(k-1)}, \square_{j\in\mathcal{N}(i)} \boxed{\phi^{(k)}} \boxed{\left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i}\right)}\right),$$

Features representations of node i at the k-th layer

Differentiable function
Eg: MLP

- Feature rep of node i at the (k-1)-th layer
- Feature rep of node j at the (k-1)-th layer
- [**optionally**] features of edge (i,j)

05 Message passing implementation

$$\mathbf{x}_i^{(k)} = \gamma^{(k)}\left(\mathbf{x}_i^{(k-1)}, \boxed{\square_{j\in\mathcal{N}(i)}}\, \boxed{\phi^{(k)}}\boxed{\left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i}\right)}\right),$$

Features representations of node i at the k-th layer

Differentiable, ordering invariant function.
For every j in the neighbourhood of i.
Eg: sum, average, etc...

Differentiable function
Eg: MLP

- Feature rep of node i at the (k-1)-th layer
- Feature rep of node j at the (k-1)-th layer
- [**optionally**] features of edge (i,j)

05 Message passing implementation

$$\mathbf{x}_i^{(k)} = \boxed{\gamma^{(k)}}\left(\mathbf{x}_i^{(k-1)}, \boxed{\square_{j\in\mathcal{N}(i)}}\,\boxed{\phi^{(k)}}\boxed{\left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i}\right)}\right),$$

Features representations of node i at the k-th layer

Differentiable function
Eg: MLP

Differentiable, ordering invariant function.
For every j in the neighbourhood of i.
Eg: sum, average, etc...

Differentiable function
Eg: MLP

- Feature rep of node i at the (k-1)-th layer
- Feature rep of node j at the (k-1)-th layer
- [**optionally**] features of edge (i,j)

05 Message passing implementation

PyTorch Geometric provides the MessagePassing base class.

05 Message passing implementation

PyTorch Geometric provides the MessagePassing base class.

$$\mathbf{x}_i^{(k)} = \gamma^{(k)}\left(\mathbf{x}_i^{(k-1)}, \square_{j\in\mathcal{N}(i)}\, \boxed{\phi^{(k)}}\left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i}\right)\right),$$

Differentiable function
Eg: MLP

message()

05 Message passing implementation

PyTorch Geometric provides the MessagePassing base class.

$$\mathbf{x}_i^{(k)} = \boxed{\gamma^{(k)}}\left(\mathbf{x}_i^{(k-1)}, \square_{j\in\mathcal{N}(i)} \boxed{\phi^{(k)}}\left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i}\right)\right),$$

Differentiable function
Eg: MLP

update()

Differentiable functions
Eg: MLP

message()

05 Message passing implementation

PyTorch Geometric provides the MessagePassing base class.

$$\mathbf{x}_i^{(k)} = \boxed{\gamma^{(k)}}\left(\mathbf{x}_i^{(k-1)}, \boxed{\square_{j\in\mathcal{N}(i)}}\,\boxed{\phi^{(k)}}\left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i}\right)\right),$$

Differentiable function
Eg: MLP

update()

Aggregation

Sum, avg, concat

Differentiable function
Eg: MLP

message()

05 Message passing implementation

PyTorch Geometric provides the MessagePassing base class.

PARAMETERS

CLASS `MessagePassing ( aggr: Optional[str] = 'add', flow: str = 'source_to_target', node_dim: int = - 2 )` [source]

Base class for creating message passing layers of the form

$$\mathbf{x}_i' = \gamma_{\mathbf{\Theta}} \left(\mathbf{x}_i, \square_{j \in \mathcal{N}(i)} \, \phi_{\mathbf{\Theta}} \left(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{j,i} \right) \right),$$

where $\square$ denotes a differentiable, permutation invariant function, *e.g.*, sum, mean or max, and $\gamma_{\mathbf{\Theta}}$ and $\phi_{\mathbf{\Theta}}$ denote differentiable functions such as MLPs. See here for the accompanying tutorial.

05 Message passing implementation

PyTorch Geometric provides the MessagePassing base class.

PARAMETERS

aggr (*string, optional*) – The aggregation scheme to use (`"add"` , `"mean"` , `"max"` or `None`). (default: `"add"`)

CLASS **`MessagePassing`** (**aggr**: Optional[str] = `'add'`, **flow**: str = `'source_to_target'`, **node_dim**: int = `- 2`) [source]

Base class for creating message passing layers of the form

$$\mathbf{x}_i' = \gamma_{\Theta} \left(\mathbf{x}_i, \square_{j \in \mathcal{N}(i)} \, \phi_{\Theta} \left(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{j,i} \right) \right),$$

where $\square$ denotes a differentiable, permutation invariant function, *e.g.*, sum, mean or max, and γ_{Θ} and ϕ_{Θ} denote differentiable functions such as MLPs. See here for the accompanying tutorial.

05 Message passing implementation

PyTorch Geometric provides the MessagePassing base class.

PARAMETERS

aggr (*string, optional*) – The aggregation scheme to use (`"add"`, `"mean"`, `"max"` or `None`). (default: `"add"`)

flow (*string, optional*) – The flow direction of message passing (`"source_to_target"` or `"target_to_source"`). (default: `"source_to_target"`)

CLASS `MessagePassing` (**aggr**: Optional[str] = `'add'`, **flow**: str = `'source_to_target'`, **node_dim**: int = `- 2`) [source]

Base class for creating message passing layers of the form

$$\mathbf{x}_i' = \gamma_{\Theta} \left(\mathbf{x}_i, \square_{j \in \mathcal{N}(i)} \, \phi_{\Theta} \left(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{j,i} \right) \right),$$

where $\square$ denotes a differentiable, permutation invariant function, *e.g.*, sum, mean or max, and γ_{Θ} and ϕ_{Θ} denote differentiable functions such as MLPs. See here for the accompanying tutorial.

05 Message passing implementation

PyTorch Geometric provides the MessagePassing base class.

METHODS

CLASS **MessagePassing** (**aggr**: Optional[str] = 'add', **flow**: str = 'source_to_target', **node_dim**: int = - 2) [source]

Aggregates messages from neighbors (sum, mean, max)

aggregate (**inputs**: torch.Tensor, **index**: torch.Tensor, **ptr**: Optional[torch.Tensor] = None, **dim_size**: Optional[int] = None) → **torch.Tensor** [source]

05 Message passing implementation

PyTorch Geometric provides the MessagePassing base class.

METHODS

CLASS **MessagePassing** (**aggr**: Optional[str] = 'add', **flow**: str = 'source_to_target', **node_dim**: int = - 2) [source]

Aggregates messages from neighbors (sum, mean, max)

aggregate (**inputs**: torch.Tensor, **index**: torch.Tensor, **ptr**: Optional[torch.Tensor] = None, **dim_size**: Optional[int] = None) → **torch.Tensor** [source]

Constructs messages from node j to node i in analogy to ϕΘ

message (**x_j**: torch.Tensor) → **torch.Tensor** [source]

05 Message passing implementation

PyTorch Geometric provides the MessagePassing base class.

METHODS

`CLASS MessagePassing ( aggr: Optional[str] = 'add', flow: str = 'source_to_target', node_dim: int = - 2 )` [source]

Aggregates messages from neighbors (sum, mean, max)

`aggregate ( inputs: torch.Tensor, index: torch.Tensor, ptr: Optional[torch.Tensor] = None, dim_size: Optional[int] = None ) → torch.Tensor` [source]

Constructs messages from node j to node i in analogy to ϕΘ

`message ( x_j: torch.Tensor ) → torch.Tensor` [source]

Propagate messages

`propagate ( edge_index: Union[torch.Tensor, torch_sparse.tensor.SparseTensor], size: Optional[Tuple[int, int]] = None, **kwargs )` [source]

05 Message passing implementation

PyTorch Geometric provides the MessagePassing base class.

METHODS

CLASS **MessagePassing** (**aggr**: Optional[str] = 'add', **flow**: str = 'source_to_target', **node_dim**: int = - 2) [source]

Aggregates messages from neighbors (sum, mean, max)

aggregate (**inputs**: torch.Tensor, **index**: torch.Tensor, **ptr**: Optional[torch.Tensor] = None, **dim_size**: Optional[int] = None) → torch.Tensor [source]

Constructs messages from node j to node i in analogy to ϕΘ

message (**x_j**: torch.Tensor) → torch.Tensor [source]

Propagate messages

propagate (**edge_index**: Union[torch.Tensor, torch_sparse.tensor.SparseTensor], **size**: Optional[Tuple[int, int]] = None, ****kwargs**) [source]

Updates node embeddings in analogy to γΘ

update (**inputs**: torch.Tensor) → torch.Tensor [source]

05 Message passing implementation

HOW TO USE IT?

Layer Name

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


    def forward(self, x, edge_index):

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


    def message(self,...):


        return ...
```

05 Message passing implementation

HOW TO USE IT?

GCNConv inherits from MessagePassing

Layer Name

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


    def forward(self, x, edge_index):

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


    def message(self,...):


        return ...
```

05 Message passing implementation

HOW TO USE IT?

GCNConv inherits from MessagePassing

Layer Name

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


    def forward(self, x, edge_index):

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


    def message(self,...):


        return ...
```

Initialize the class, call “super” specifying your aggregations (add,max,mean)

05 Message passing implementation

HOW TO USE IT?

GCNConv inherits from MessagePassing

Layer Name

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

    def forward(self, x, edge_index):

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

    def message(self,...):

        return ...
```

Initialize the class, call “super” specifying your aggregations (add,max,mean)

Forward and propagate

05 Message passing implementation

HOW TO USE IT?

GCNConv inherits from MessagePassing

Layer Name

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

    def forward(self, x, edge_index):

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

    def message(self,...):

        return ...
```

Initialize the class, call “super” specifying your aggregations (add,max,mean)

Forward and propagate

Compute the message

06 Implement our GCNConv

Simple example

$$\mathbf{x}_i^{(k)} = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{\deg(i)} \cdot \sqrt{\deg(j)}} \cdot \left(\mathbf{\Theta} \cdot \mathbf{x}_j^{(k-1)} \right)$$

06 Implement our GCNConv

Simple example

$$\mathbf{x}_i^{(k)} = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{\deg(i)} \cdot \sqrt{\deg(j)}} \cdot \left(\mathbf{\Theta} \cdot \mathbf{x}_j^{(k-1)} \right)$$

$$\mathbf{x}_i^{(k)} = \gamma^{(k)} \left(\mathbf{x}_i^{(k-1)}, \square_{j \in \mathcal{N}(i)} \, \phi^{(k)} \left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i} \right) \right),$$

06 Implement our GCNConv

Simple example

$$\mathbf{x}_i^{(k)} = \boxed{\sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{\deg(i)} \cdot \sqrt{\deg(j)}}} \cdot \left(\boxed{\mathbf{\Theta}} \cdot \mathbf{x}_j^{(k-1)} \right)$$

$$\mathbf{x}_i^{(k)} = \gamma^{(k)} \left(\mathbf{x}_i^{(k-1)}, \boxed{\square_{j \in \mathcal{N}(i)}} \boxed{\phi^{(k)}} \left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i} \right) \right),$$

06 Implement our GCNConv

Simple example

$$\mathbf{x}_i^{(k)} = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{\deg(i)} \cdot \sqrt{\deg(j)}} \cdot \left(\mathbf{\Theta} \cdot \mathbf{x}_j^{(k-1)} \right)$$

In steps:

1. Add self loops
2. A linear transformation to node feature matrix
3. Compute normalization coefficients
4. Normalize node features
5. Sum up neighboring node features

06 Implement our GCNConv

Simple example

$$\mathbf{x}_i^{(k)} = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{\deg(i)} \cdot \sqrt{\deg(j)}} \cdot \left(\mathbf{\Theta} \cdot \mathbf{x}_j^{(k-1)} \right)$$

In steps:

1. Add self loops
2. A linear transformation to node feature matrix
3. Compute normalization coefficients
4. Normalize node features
5. Sum up neighboring node features

Forward method

Message method

int

06 Implement our GCNConv

GCNConv inherits from MessagePassing

```
class GCNConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(GCNConv, self).__init__(aggr='add')  # "Add" aggregation (Step 5).
        self.lin = torch.nn.Linear(in_channels, out_channels)


    def forward(self, x, edge_index):
        # x has shape [N, in_channels]
        # edge_index has shape [2, E]

        # Step 1: Add self-loops to the adjacency matrix.
        edge_index, _ = add_self_loops(edge_index, num_nodes=x.size(0))

        # Step 2: Linearly transform node feature matrix.
        x = self.lin(x)

        # Step 3: Compute normalization.
        row, col = edge_index
        deg = degree(col, x.size(0), dtype=x.dtype)
        deg_inv_sqrt = deg.pow(-0.5)
        norm = deg_inv_sqrt[row] * deg_inv_sqrt[col]

        # Step 4-5: Start propagating messages.
        return self.propagate(edge_index, x=x, norm=norm)



    def message(self, x_j, norm):
        # x_j has shape [E, out_channels]

        # Step 4: Normalize node features.
        return norm.view(-1, 1) * x_j
```

06 Implement our GCNConv

GCNConv inherits from MessagePassing

```
class GCNConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(GCNConv, self).__init__(aggr='add')  # "Add" aggregation (Step 5).
        self.lin = torch.nn.Linear(in_channels, out_channels)


    def forward(self, x, edge_index):
        # x has shape [N, in_channels]
        # edge_index has shape [2, E]

        # Step 1: Add self-loops to the adjacency matrix.
        edge_index, _ = add_self_loops(edge_index, num_nodes=x.size(0))

        # Step 2: Linearly transform node feature matrix.
        x = self.lin(x)

        # Step 3: Compute normalization.
        row, col = edge_index
        deg = degree(col, x.size(0), dtype=x.dtype)
        deg_inv_sqrt = deg.pow(-0.5)
        norm = deg_inv_sqrt[row] * deg_inv_sqrt[col]

        # Step 4-5: Start propagating messages.
        return self.propagate(edge_index, x=x, norm=norm)



    def message(self, x_j, norm):
        # x_j has shape [E, out_channels]

        # Step 4: Normalize node features.
        return norm.view(-1, 1) * x_j
```

1) Add self loops

06 Implement our GCNConv

GCNConv inherits from MessagePassing

```
class GCNConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(GCNConv, self).__init__(aggr='add')  # "Add" aggregation (Step 5).
        self.lin = torch.nn.Linear(in_channels, out_channels)


    def forward(self, x, edge_index):
        # x has shape [N, in_channels]
        # edge_index has shape [2, E]

        # Step 1: Add self-loops to the adjacency matrix.
        edge_index, _ = add_self_loops(edge_index, num_nodes=x.size(0))

        # Step 2: Linearly transform node feature matrix.
        x = self.lin(x)

        # Step 3: Compute normalization.
        row, col = edge_index
        deg = degree(col, x.size(0), dtype=x.dtype)
        deg_inv_sqrt = deg.pow(-0.5)
        norm = deg_inv_sqrt[row] * deg_inv_sqrt[col]

        # Step 4-5: Start propagating messages.
        return self.propagate(edge_index, x=x, norm=norm)



    def message(self, x_j, norm):
        # x_j has shape [E, out_channels]

        # Step 4: Normalize node features.
        return norm.view(-1, 1) * x_j
```

1) **Add self loops**

2) **A linear transformation to node feature matrix**

06 Implement our GCNConv

GCNConv inherits from MessagePassing

```
class GCNConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(GCNConv, self).__init__(aggr='add')  # "Add" aggregation (Step 5).
        self.lin = torch.nn.Linear(in_channels, out_channels)


    def forward(self, x, edge_index):
        # x has shape [N, in_channels]
        # edge_index has shape [2, E]

        # Step 1: Add self-loops to the adjacency matrix.
        edge_index, _ = add_self_loops(edge_index, num_nodes=x.size(0))

        # Step 2: Linearly transform node feature matrix.
        x = self.lin(x)

        # Step 3: Compute normalization.
        row, col = edge_index
        deg = degree(col, x.size(0), dtype=x.dtype)
        deg_inv_sqrt = deg.pow(-0.5)
        norm = deg_inv_sqrt[row] * deg_inv_sqrt[col]

        # Step 4-5: Start propagating messages.
        return self.propagate(edge_index, x=x, norm=norm)



    def message(self, x_j, norm):
        # x_j has shape [E, out_channels]

        # Step 4: Normalize node features.
        return norm.view(-1, 1) * x_j
```

1) **Add self loops**

2) **A linear transformation to node feature matrix**

3) **Compute normalization coefficients**

06 Implement our GCNConv

GCNConv inherits from MessagePassing

```
class GCNConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(GCNConv, self).__init__(aggr='add')  # "Add" aggregation (Step 5).
        self.lin = torch.nn.Linear(in_channels, out_channels)


    def forward(self, x, edge_index):
        # x has shape [N, in_channels]
        # edge_index has shape [2, E]

        # Step 1: Add self-loops to the adjacency matrix.
        edge_index, _ = add_self_loops(edge_index, num_nodes=x.size(0))

        # Step 2: Linearly transform node feature matrix.
        x = self.lin(x)

        # Step 3: Compute normalization.
        row, col = edge_index
        deg = degree(col, x.size(0), dtype=x.dtype)
        deg_inv_sqrt = deg.pow(-0.5)
        norm = deg_inv_sqrt[row] * deg_inv_sqrt[col]

        # Step 4-5: Start propagating messages.
        return self.propagate(edge_index, x=x, norm=norm)



    def message(self, x_j, norm):
        # x_j has shape [E, out_channels]

        # Step 4: Normalize node features.
        return norm.view(-1, 1) * x_j
```

1) Add self loops
2) A linear transformation to node feature matrix
3) Compute normalization coefficients
4) Normalize node features

06 Implement our GCNConv

GCNConv inherits from MessagePassing

```
class GCNConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(GCNConv, self).__init__(aggr='add')  # "Add" aggregation (Step 5).
        self.lin = torch.nn.Linear(in_channels, out_channels)


    def forward(self, x, edge_index):
        # x has shape [N, in_channels]
        # edge_index has shape [2, E]

        # Step 1: Add self-loops to the adjacency matrix.
        edge_index, _ = add_self_loops(edge_index, num_nodes=x.size(0))

        # Step 2: Linearly transform node feature matrix.
        x = self.lin(x)

        # Step 3: Compute normalization.
        row, col = edge_index
        deg = degree(col, x.size(0), dtype=x.dtype)
        deg_inv_sqrt = deg.pow(-0.5)
        norm = deg_inv_sqrt[row] * deg_inv_sqrt[col]

        # Step 4-5: Start propagating messages.
        return self.propagate(edge_index, x=x, norm=norm)



    def message(self, x_j, norm):
        # x_j has shape [E, out_channels]

        # Step 4: Normalize node features.
        return norm.view(-1, 1) * x_j
```

5) **Sum up neighboring node features**

1) **Add self loops**

2) **A linear transformation to node feature matrix**

3) **Compute normalization coefficients**

4) **Normalize node features**

06 GAT implementation

Jupyter-Notebook