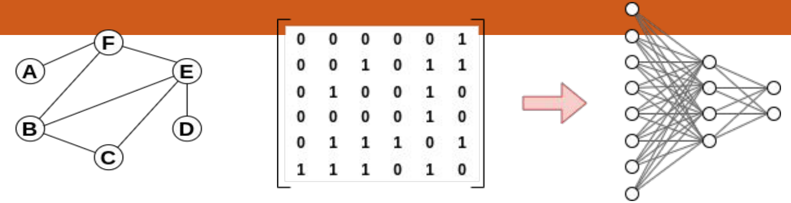
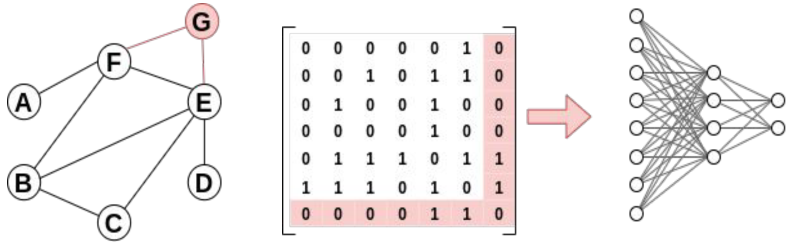
Graph attention Networks (GAT)

Ricapitoliamo



Problemi:

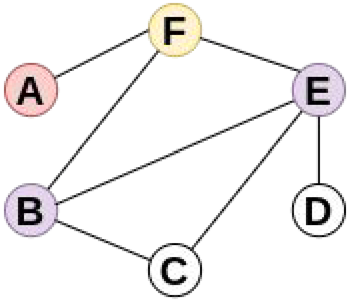
- Dimensioni differenti



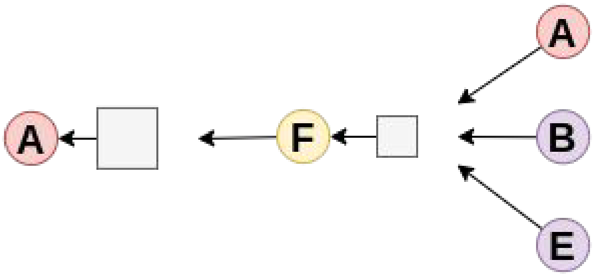
- Non invariante all'ordinamento dei nodi

Ricapitoliamo

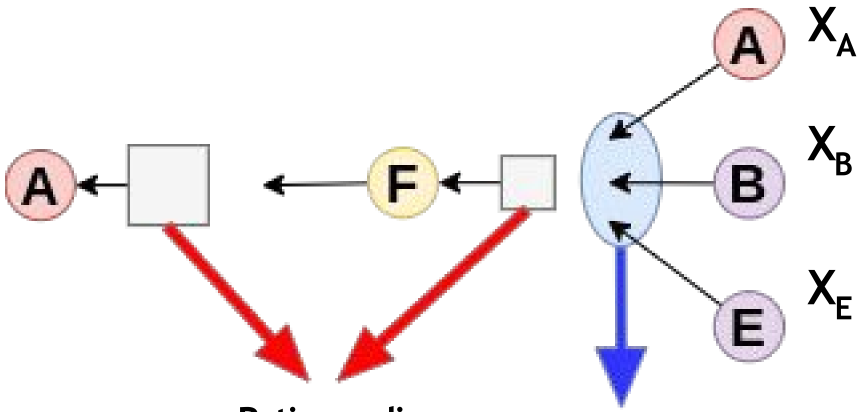
Grafo



Grafo computazionale



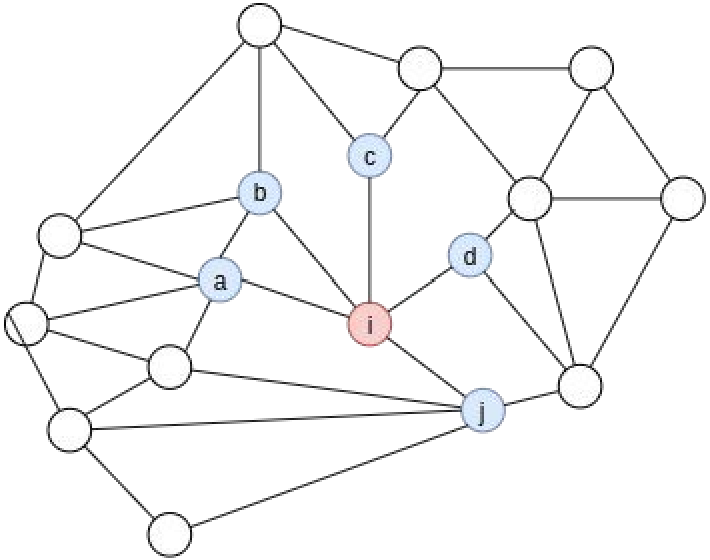
Ricapitoliamo

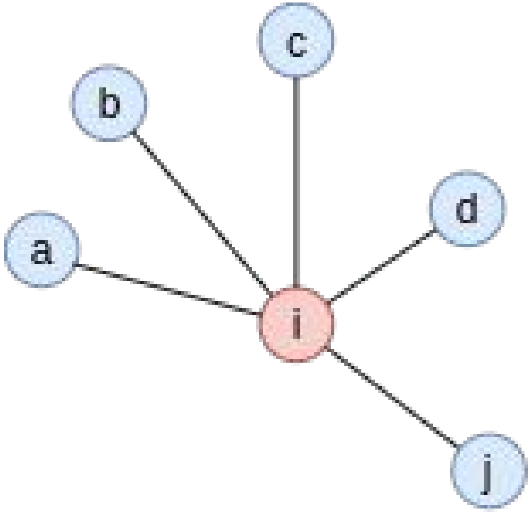


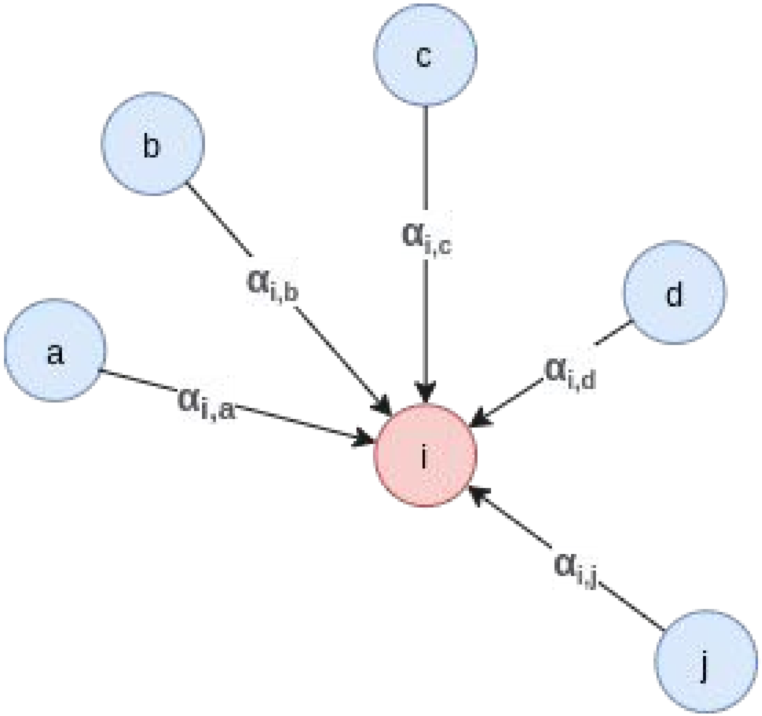
Reti neurali

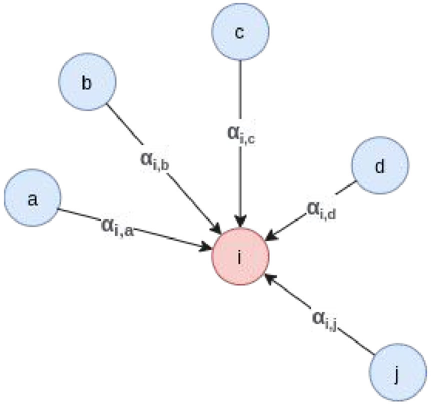
<u>Ordering invariant</u>
Aggregation

Sum
Average

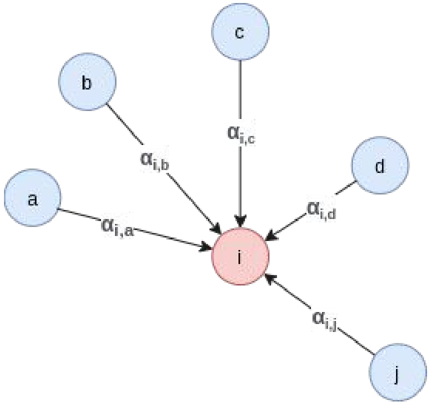

b
c
a
d
i
j


c
b
d
a
i
j


c
b
$\alpha_{i,c}$
$\alpha_{i,b}$
d
a
$\alpha_{i,d}$
$\alpha_{i,a}$
i
$\alpha_{i,j}$
j

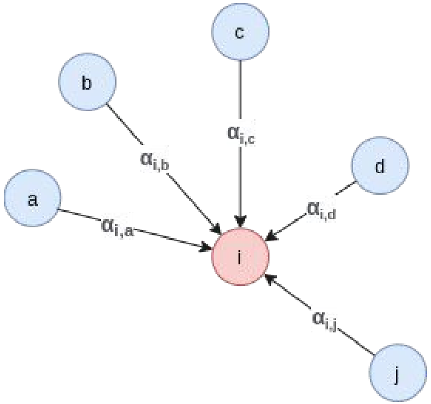


Quanto le caratteristiche del nodo “c” sono importanti per il nodo “i”?



Quanto le caratteristiche del nodo “c” sono importanti per il nodo “i”?

Possiamo imparare questa importanza in modo automatico?



Quanto le caratteristiche del nodo “c” sono importanti per il nodo “i”?

Possiamo imparare questa importanza in modo automatico?

Si con le GAT

Petar Veličković

Senior Research Scientist at DeepMind

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

Graph Attention layer

INPUT: insieme di feature per ogni nodo

$$\mathbf{h} = \{\bar{h}_1, \bar{h}_2, \ldots, \bar{h}_n\} \quad \bar{h}_i \in \mathbf{R}^F$$

$$\mathbf{h'} = \{\bar{h'}_1, \bar{h'}_2, \ldots, \bar{h'}_n\} \quad \bar{h'}_i \in \mathbf{R}^{F'}$$

OUTPUT: un nuovo set di feature

Graph Attention layer

1) applicare una trasformazione lineare parametrizzata a ogni nodo

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

Graph Attention layer

1) applicare una trasformazione lineare parametrizzata a ogni nodo

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

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

Graph Attention layer

1) applicare una trasformazione lineare parametrizzata a ogni nodo

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

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

$$F'$$

Graph Attention layer

2) Self attention

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

Graph Attention layer

2) Self attention

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

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

Graph Attention layer

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

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

Specificare l'importanza delle caratteristiche del nodo j per il nodo i

Graph Attention layer

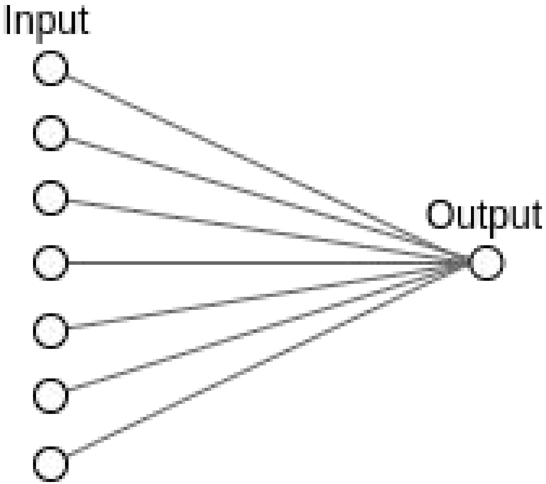
3) Normalizzazione

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

Graph Attention layer

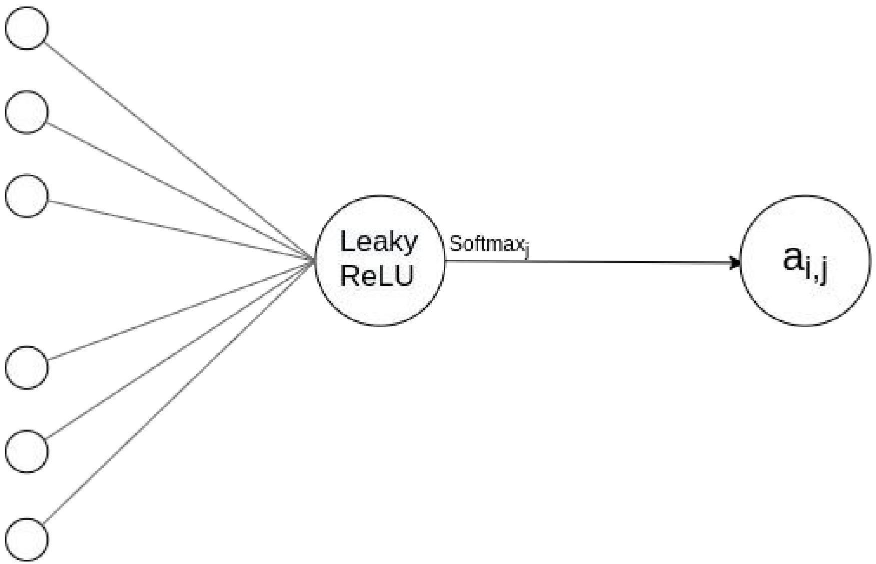
4) Meccanismo di attenzione a

È una rete neurale feed forward a singolo layer



Graph Attention layer

4) Meccanismo di attenzione a



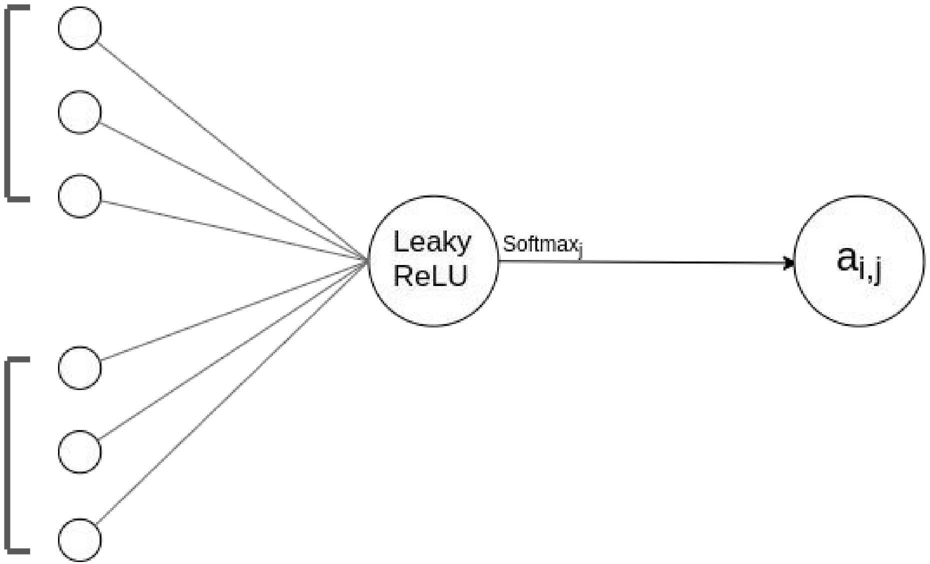
Graph Attention layer

4) Meccanismo di attenzione

$$\boldsymbol{a}$$

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

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



Graph Attention layer

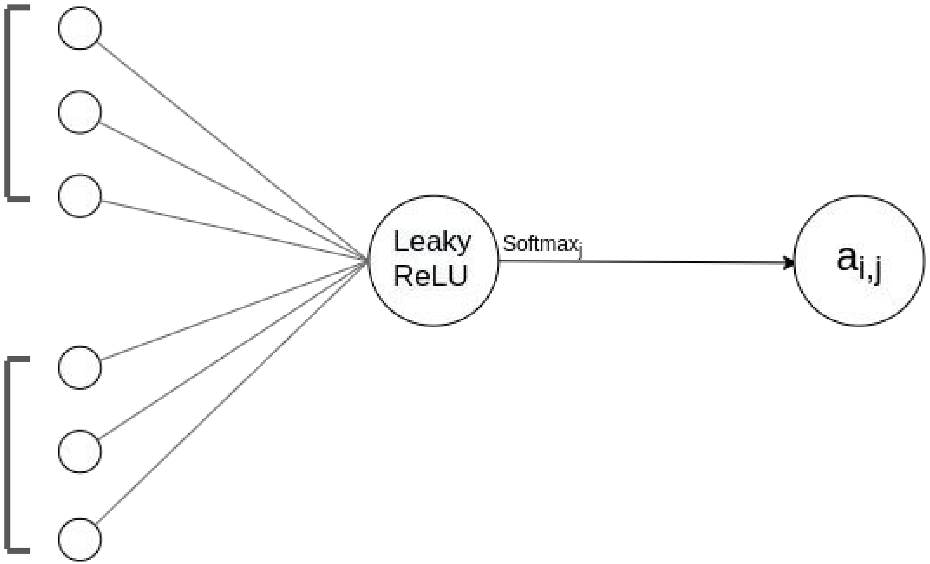
4) Meccanismo di attenzione a

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

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

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

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



Graph Attention layer

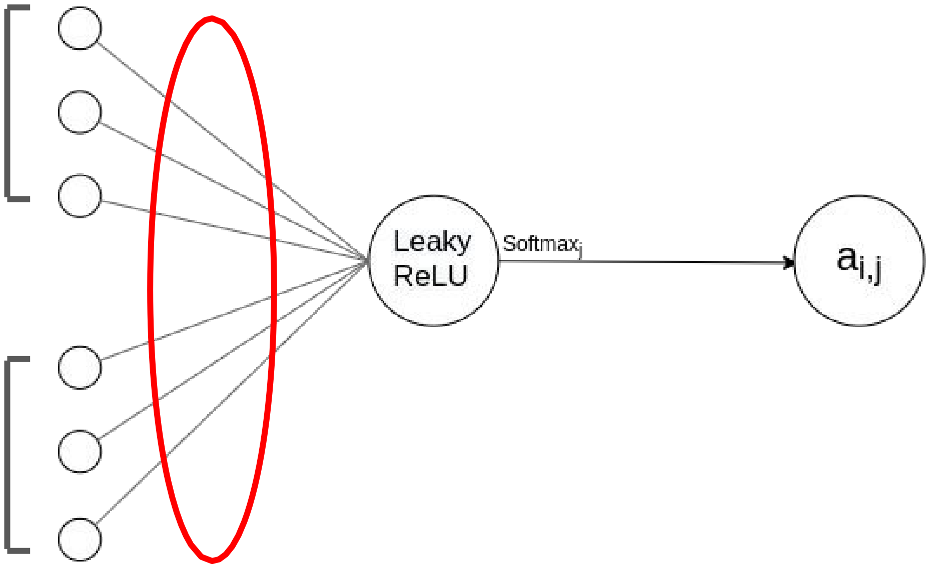
4) Meccanismo di attenzione $\boldsymbol{a}$

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

$$(F' \times F)F \to F'$$

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

$$(F' \times F)F \to F'$$



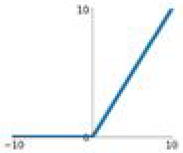
$$\bar{a} \in \mathbf{R}^{2F'}$$

Graph Attention layer

4) Meccanismo di attenzione

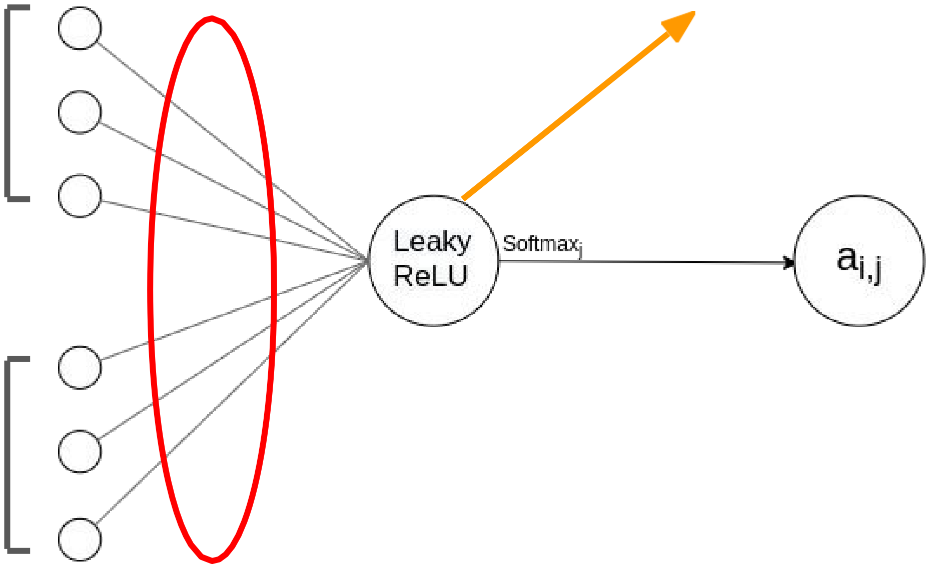
$$a$$

ReLU
$\max(0, x)$



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

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



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

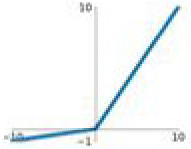
Graph Attention layer

4) Meccanismo di attenzione

a

Leaky ReLU

max(0.2x, x)

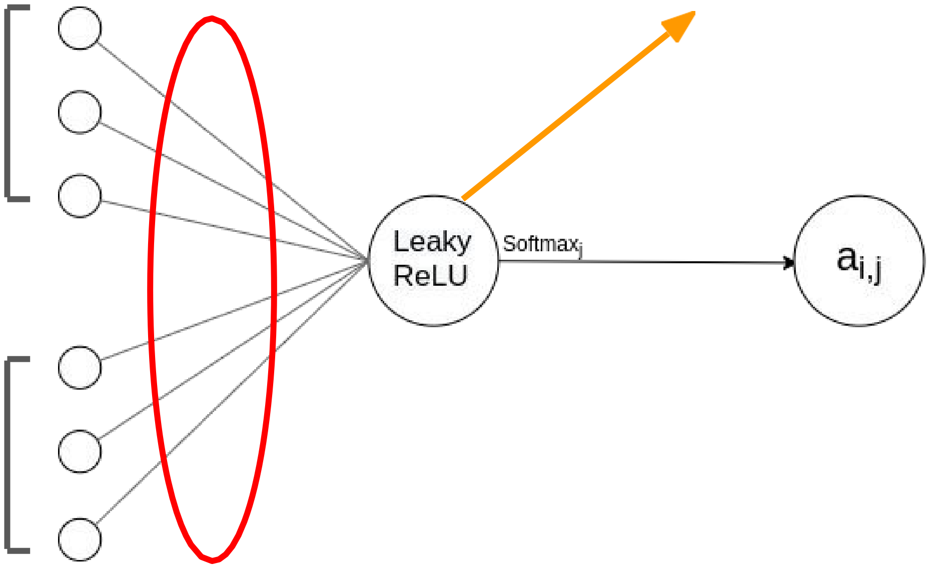


$$\mathbf{W}h_i \rightarrow h'_i$$

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

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

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



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

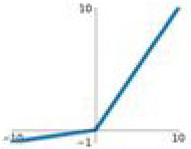
Graph Attention layer

4) Meccanismo di attenzione

a

Leaky ReLU

max(0.2x, x)

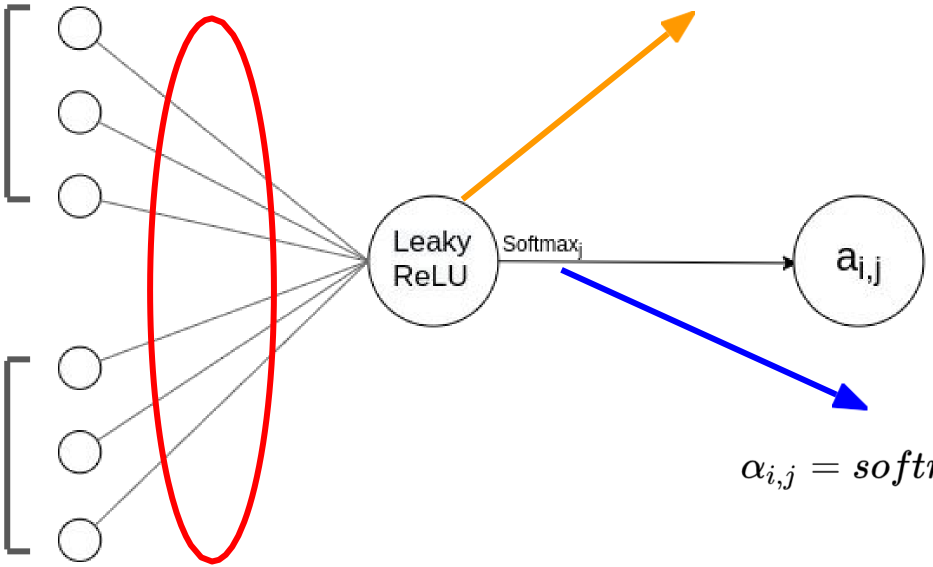


$$\mathbf{W}h_i \rightarrow h'_i$$

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

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

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



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

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

Graph Attention layer

4) Meccanismo di attenzione $\boldsymbol{a}$

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

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

$|| \rightarrow concatenation$

Graph Attention layer

4) Meccanismo di attenzione $\boldsymbol{a}$

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

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

$|| \rightarrow concatenation$

Graph Attention layer

4) Meccanismo di attenzione $\qquad a$

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

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

$|| \rightarrow concatenation$

Graph Attention layer

4) Meccanismo di attenzione a

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

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

$|| \rightarrow concatenation$

Graph Attention layer

4) Meccanismo di attenzione $\boldsymbol{a}$

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

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

$|| \rightarrow concatenation$

Graph Attention layer

4) Meccanismo di attenzione $\quad \boldsymbol{a}$

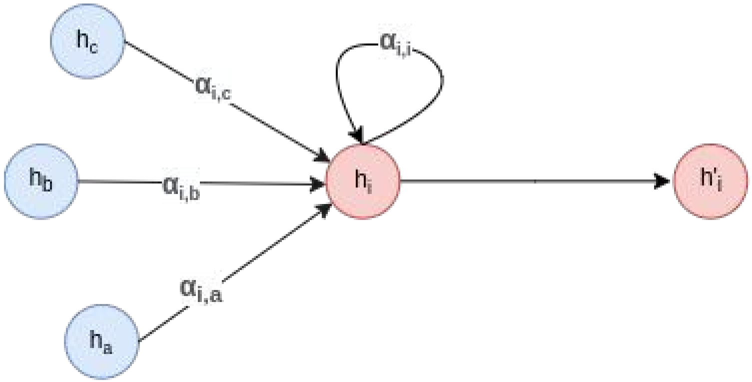
$$\alpha_{i,j} = \frac{\overbrace{exp(LeakyReLU(\overbrace{\overbrace{\bar{a}^T}^{(1 \times 2F')} \overbrace{[\overbrace{\mathbf{W}h_i}^{\substack{F' \\ (F' \times F)F}} || \overbrace{\mathbf{W}h_j}^{\substack{F' \\ (F' \times F)F}}]}^{(2F' \times 1)}}^{1}))}^{1}}{\sum_{k \in N(i)} exp(LeakyReLU(\bar{a}^T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

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

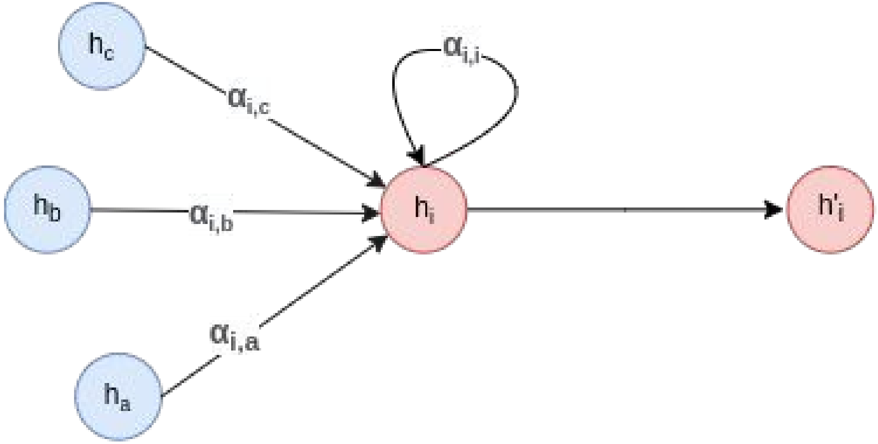
$|| \rightarrow concatenation$

5) Usiamolo

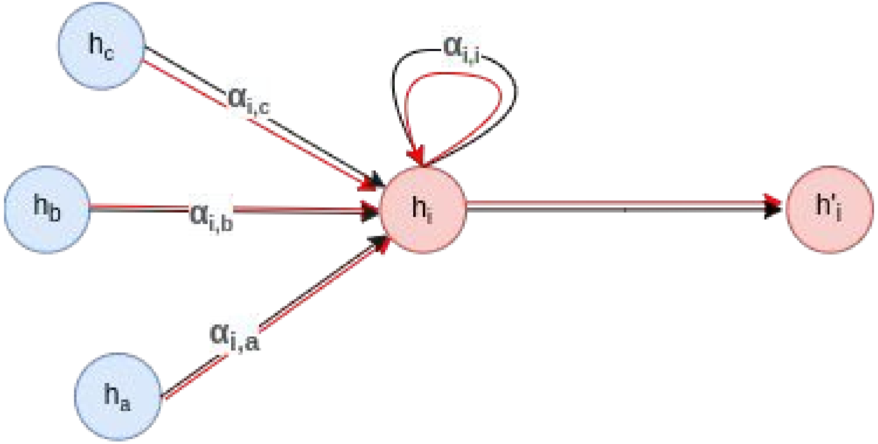
$$h'_i = \sigma\left(\sum_{j \in N(i)} \alpha_{i,j} \mathbf{W} h_j\right)$$



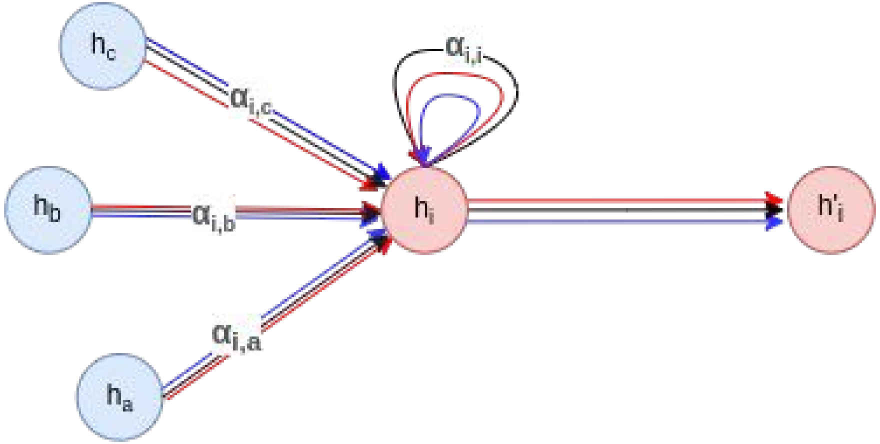
6) Multi-head attention



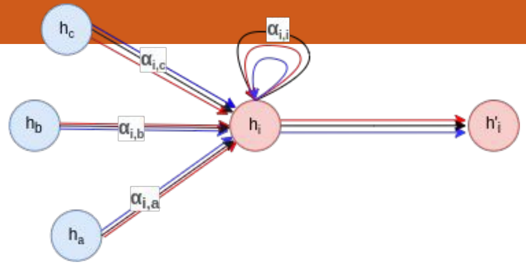
6) Multi-head attention



6) Multi-head attention



6) Multi-head attention



Concatenazione

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

Media

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

- Sullo strato finale (di predizione) della rete

Vantaggi delle GAT

- Efficiente dal punto di vista computazionale

Self-attention layer possono essere parallelizzato su tutti gli archi

Output feature possono essere parallelizzate tra i nodi

- Consente di assegnare importanza differente a nodi dello stesso vicinato

- Viene applicato in modo condiviso a tutti gli archi del grafo

Non è necessario avere l'intero grafo

Vantaggi delle GAT

- Efficiente dal punto di vista computazionale
  - Self-attention layer possono essere parallelizzato su tutti gli archi
  - Output feature possono essere parallelizzate tra i nodi
- Consente di assegnare importanza differente a nodi dello stesso vicinato
- Viene applicato in modo condiviso a tutti gli archi del grafo
  - **Non è necessario avere l'intero grafo**
- Applicabile
  - **Trasduttive learning (non dinamiche)**
  - **Inductive** learning (PPI)

Implementaizone con message passing

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

Rappresentazioni delle caratteristiche del nodo i al k-th layer

Funzione differenziabile
Es: MLP

Funzione differenziabile, invariante.
Per ogni j nelle vicinanze di i. es: sum, average, etc...

Funzione differenziabile
Es: MLP

- Feauture del nodo i al layer (k-1)
- Feauture del nodo j al layer (k-1)-th
- [opzionali] feature dell’arco (i,j)

Implementaizone con message passing

PyTorch Geometric MessagePassing base class.

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

Funzione differenziabile Es: MLP

update()

Aggregation

Sum, avg, concat

Funzione differenziabile Es: MLP

message()

Implementaizone con message passing

PyTorch Geometric MessagePassing base class.

Parametri

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.

Implementaizone con message passing

PyTorch Geometric MessagePassing base class.
Parametri

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.

Implementaizone con message passing

PyTorch Geometric MessagePassing base class.
Parametri

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.

Implementaizone con message passing

PyTorch Geometric MessagePassing base class.

Metodi

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

Aggregates messaggi dei vicini (sum, mean, max)

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

Costruisce il messaggio da j ad i in analogia con $\phi\Theta$

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

Propaga i messaggi

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

Agiorna gli embeddings similmente a $\gamma\Theta$

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

Implementaizone con message passing

Come usarla

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 ...
```

Implementaizone con message passing

Come usarla

GCNConv eredita da 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 ...
```

Inizializziamo la classe, “super” specifichiamo l’aggregazione (add,max,mean)

Forward e propagate

Calcolo del messaggio

Implementaizone con message passing

Eesempio

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

Implementaizone con message passing

Eesempio

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

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

Implementaizone con message passing

Eesempio

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

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

Implementaizone con message passing

Eesempio

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

In passi:

1. Aggiugiamo self-loop
2. Trasformazione lineare della matrice delle feature dei nodi
3. Coefficienti di normalizzazione
4. Normalizzazione delle feature
5. Somma delle feature del vicinato

Forward method

Message method

int

GCNConv eredita da 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) Riepilogare le feature dei nodi adiacenti

1) self loops

2) Una trasformazione lineare in una matrice di caratteristiche dei nodi

3) Calcolare i coefficienti di normalizzazione

4) Normalizzare le feature dei nodi