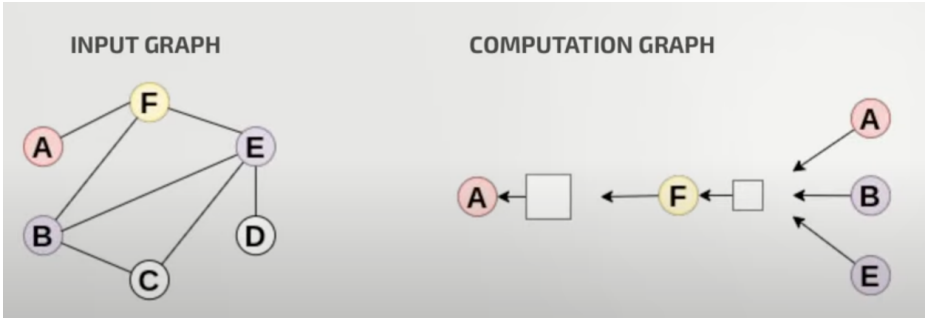
1. Recap.

Computation graph ⇒ The neighbor of a node defines its computational graph.



✗ adjacency matrix

⇒ (in neural networks)

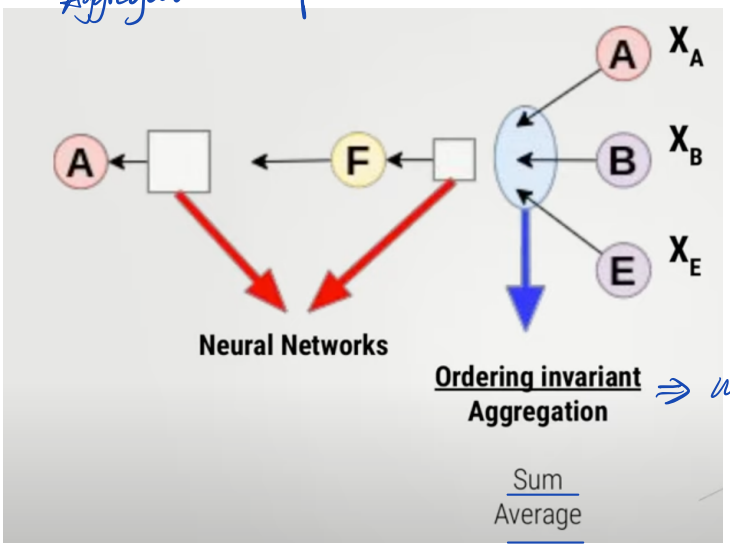
why?

- different sizes.
- Not invariant to nodes ordering.

Special case ⇒ Isomorphic.

plot leads to different adjacency matrix.

Aggregation Steps:



⇒ no inherent order.

GAT (Graph Attention layer)

Input: a set of node features $h = \{\bar{h}_1, \bar{h}_2, \cdots, \bar{h}_n\}$ $\bar{h}_i \in \mathbb{R}^F$

Output: a new set of node features $h' = \{\bar{h}'_1, \bar{h}'_2, \cdots, \bar{h}'_n\}$ $\bar{h}'_i \in \mathbb{R}^{F'}$

① apply a parameterized linear transformation to every node.

$W \cdot \bar{h}_i$ $\quad W \in \mathbb{R}^{F' \times F}$. $\quad W \cdot \bar{h}_i$.

$W \in \mathbb{R}^{F' \times F}$

$\bar{h}_i \in \mathbb{R}^F$

② Self attention.

$a: \mathbb{R}^{F'} \times \mathbb{R}^{F'} \Rightarrow \mathbb{R}$

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

⇓

specify the importance of node j's features to node i.

- W is a weight matrix ⇒ transform $\bar{h}$ to higher dim
- e_{ij} ⇒ attention coefficient.

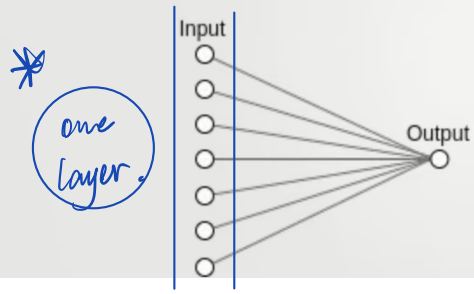
3) Normalization

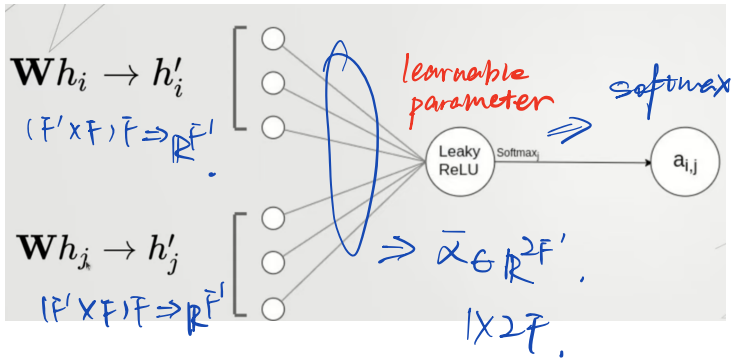
$$\alpha_{i,j} = softmax_j(e_{i,j}) = \frac{exp(e_{i,j})}{\sum_{k \in N(i)} exp(e_{i,k})}$$

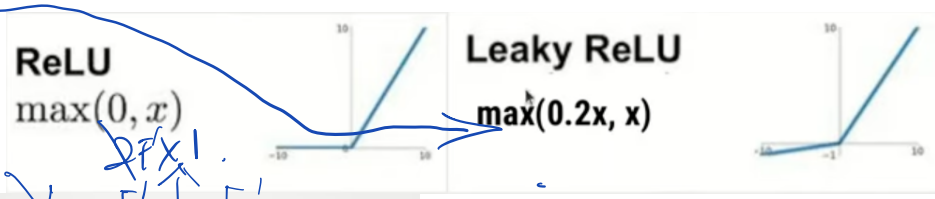
* The attention coefficient are typically normalized across all choices of j using softmax. so that they can sum to one.

4) Attention mechanism a

Is a single-layer feed forward neural network







1

1×1 $1 \times 2F'$. $2F' \times 1$. F' F'

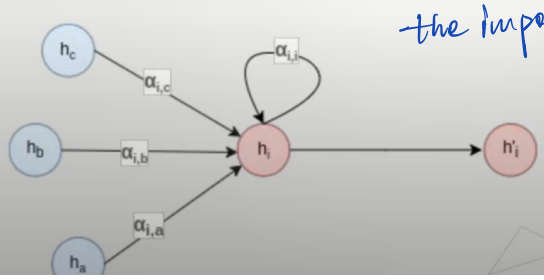
$$\alpha_{i,j} = \frac{exp(LeakyReLU(\bar{a}^T [\overbrace{\mathbf{W}h_i}^{(F' \times F)F} || \overbrace{\mathbf{W}h_j}^{(F' \times F)F}]))}{\sum_{k \in N(i)} exp(LeakyReLU(\bar{a}^T [\mathbf{W}h_i || \mathbf{W}h_k]))}$$

= real #

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

→ multi-head attention. repeat is several times.

learn $\alpha_{i,j}$ to tell the importance of feature of the node.



use for inner-layer → concatenate k times.

use for final layer. → do avg across k times.

Concatenation

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

Average

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

- On the final (prediction) layer of the network

Pros of using GATS

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)

```python
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)
```

5) **Sum up neighboring node features**

use aggregation { sum, avg }

ordering invariant method

1) **Add self loops** use the build-in function

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

3) **Compute normalization coefficients** :

$$\frac{1}{\sqrt{deg(i)} \cdot \sqrt{deg(j)}}$$

Jupyter Notebook.

①
```python
#Base case
class GATLayer(nn.Module):
  def __init__(self):
    super(GATLayer, self).__init__()#initialize the class, call "super" specifying your aggregation

  def forward(self, input, adj):
    print(" ")
```

initialize the class

② Linear transformation.

$W \in \mathbb{R}^{F' \times F}$ $\quad \vec{h_i} \in \mathbb{R}^F$. $\quad \vec{h_i'} = W \cdot \vec{h_i} \in \mathbb{R}^{(F' \times F) \times F}$.

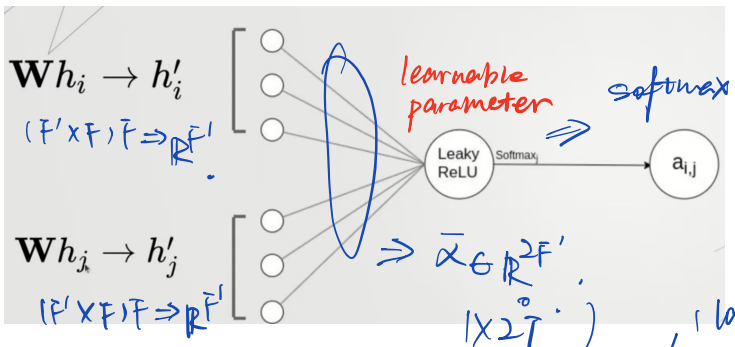
$\Rightarrow \vec{h_i'} \in \mathbb{R}^{F'}$

```python
in_features = 5
out_features = 2
nb_nodes = 3

W = nn.Parameter(torch.zeros(size=(in_features,out_features)))
nn.init.xavier_uniform_(W.data,gain=1.414)
input = torch.rand(nb_nodes,in_features)

h = torch.mm(input,W)
N = h.size()[0]

print(h.shape)
```

⇒ xavier parameter initializator.

⇒ do matrix multiplication

③ Recall the learning process.



$\Rightarrow \vec{\alpha} \in \mathbb{R}^{2F'}$.

$1 \times 2F'$.

1 layer size $2F'$

```python
a = nn.Parameter(torch.zeros(size=(2*out_features,1)))
nn.init.xavier_uniform_(a.data,gain = 1.414)
print(a.shape)

leakyrelu = nn.LeakyReLU(0.2)
```

⇒ initialize it.

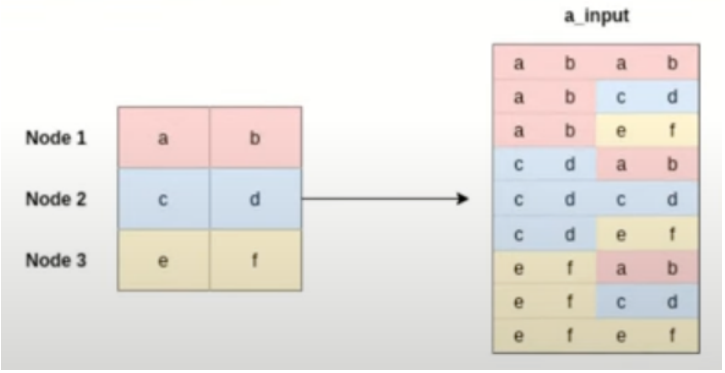
set leakyrelu.

torch.Size([4,1]) as. output-size = 2 $\quad 2 \times F' = 4$,

for one layer.

④ Compute all the possible combination

```python
a_input = torch.cat([h.repeat(1,N).view(N*N,-1), h.repeat(N,1)], dim=1).view(N,-1,2*out_features)
```



concat all possible combinations.

④ compute LeakyRelu

```
e = leakyrelu(torch.matmul(a_input,a).squeeze(2))
```

```
print(a_input.shape, a.shape)
print("")
print(torch.matmul(a_input,a).shape)
print("")
print(torch.matmul(a_input,a).squeeze(2).shape)
```

```
torch.Size([3, 3, 4]) torch.Size([4, 1])

torch.Size([3, 3, 1])
```

⇒ squeeze(2) remove the third col.

```
torch.Size([3, 3])
```

⇒ change the dim to 3×3 matrix.

⑤ Masked attention.

```
# Masked Attention
adj = torch.randint(2,(3,3))
zero_vec = -9e15*torch.ones_like(e)
print(zero_vec.shape)
```

```
attention = torch.where(adj>0, e, zero_vec)
print(adj, "\n", e, "\n", zero_vec)
attention
```

```
tensor([[1, 0, 1],
        [0, 0, 0],
        [0, 1, 1]])
 tensor([[-0.2220, -0.0950,  0.2007],
         [-0.3181, -0.1911, -0.0559],
         [-0.4623, -0.3354, -0.2002]], grad_fn=<LeakyReluBackward0>)
 tensor([[-9.0000e+15, -9.0000e+15, -9.0000e+15],
         [-9.0000e+15, -9.0000e+15, -9.0000e+15],
         [-9.0000e+15, -9.0000e+15, -9.0000e+15]])
tensor([[-2.2200e-01, -9.0000e+15,  2.0066e-01],
        [-9.0000e+15, -9.0000e+15, -9.0000e+15],
        [-9.0000e+15, -3.3538e-01, -2.0021e-01]], grad_fn=<WhereBackward0>)
```

<0, set to a extreme small number, in this case, -9.00^{+15}.

attention

⑥ normalization on attention, and calculate hprime.

```
] attention = F.softmax(attention, dim =1)
  h_prime = torch.matmul(attention,h)
```

```
] attention
```

```
tensor([[0.3959, 0.0000, 0.6041],
        [0.3333, 0.3333, 0.3333],
        [0.0000, 0.4663, 0.5337]], grad_fn=<SoftmaxBackward0>)
```

```
h_prime
```

```
tensor([[ 2.4552, -1.4964],
        [ 2.5126, -1.3958],
        [ 2.2673, -1.4275]], grad_fn=<MmBackward0>)
```