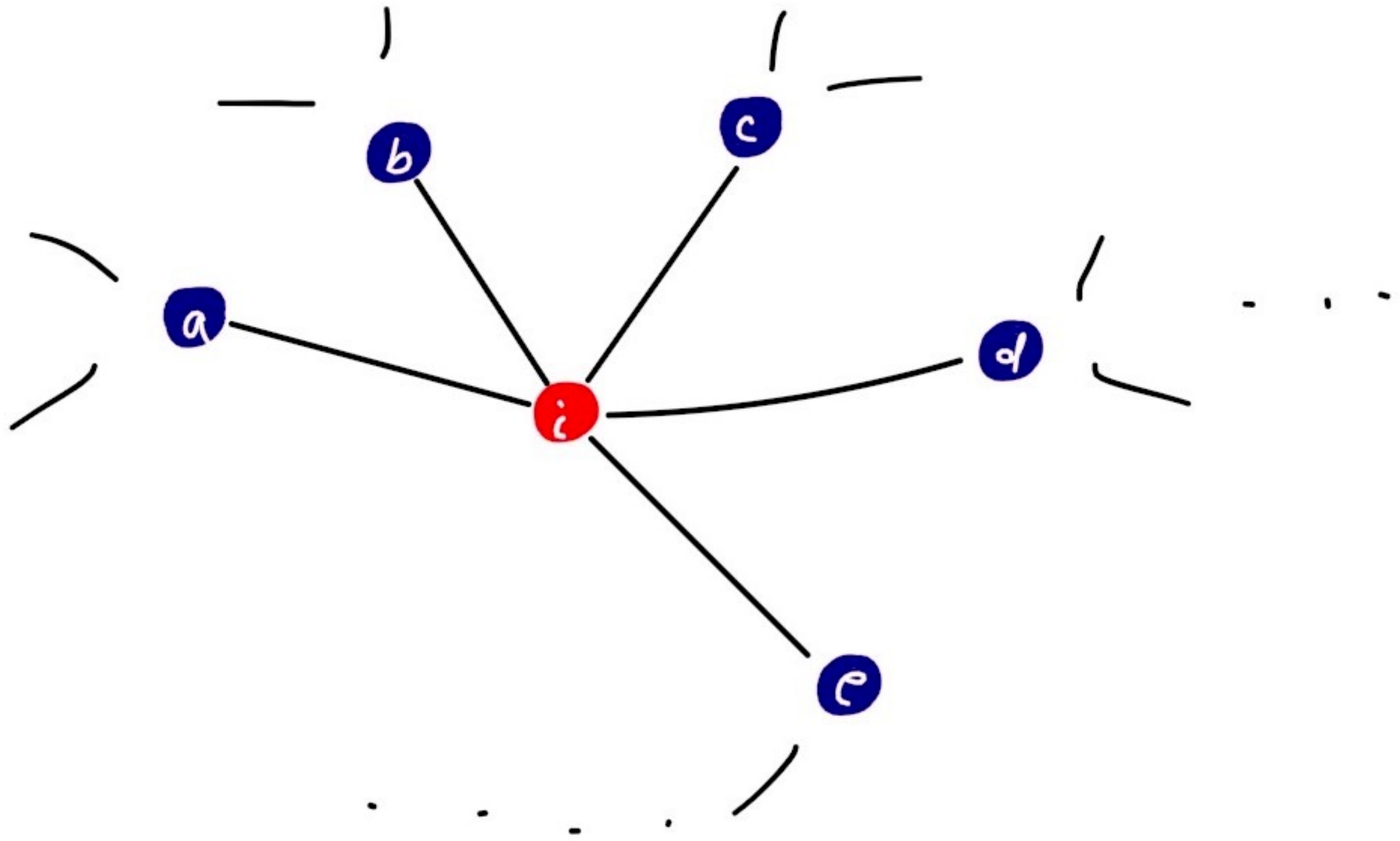
# Graph Attention Networks (GAN) in PyG
ICLR'18




★ How much the features of $a-e$ are important to node $i$? → we'd like to learn this

a set of node features $h = \{\bar{h}_1, \ldots, \bar{h}_n\},\ \bar{h}_i \in R^F$ → $\boxed{\text{graph attention Layer}}$ → a new set of node features $h' = \{\bar{h}'_1, \ldots, \bar{h}'_n\},\ \bar{h}'_i \in R^{F'}$

① apply a parameterized linear transformation to each node

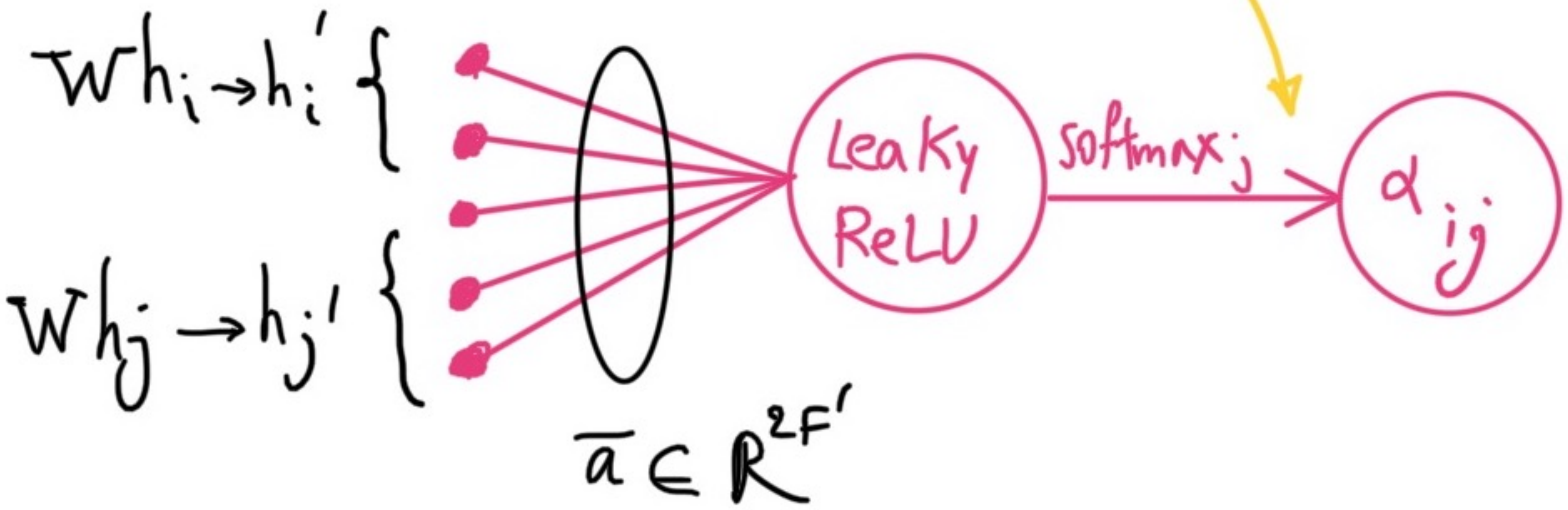
$$W \cdot \bar{h}_i \ , \ W \in R^{F' \times F}$$

② do self attention $\quad a : R^{F'} \times R^{F'} \to R$

importance of node $j$'s features to node $i$. ← $e_{ij} = a(W \cdot \bar{h}_i \ , \ W \cdot \bar{h}_j)$

③ normalize. $\alpha_{ij} = \text{softmax}_j(e_{ij}) = \dfrac{\exp(e_{ij})}{\underbrace{\sum_{k \in N(i)} \exp(e_{i,k})}_{\text{sum of neighbors of } i}}$

④ attention mechanism $a$ : a single-layer FFNN.




Leaky ReLU = $\max(0.2x, x)$

Complete formula:

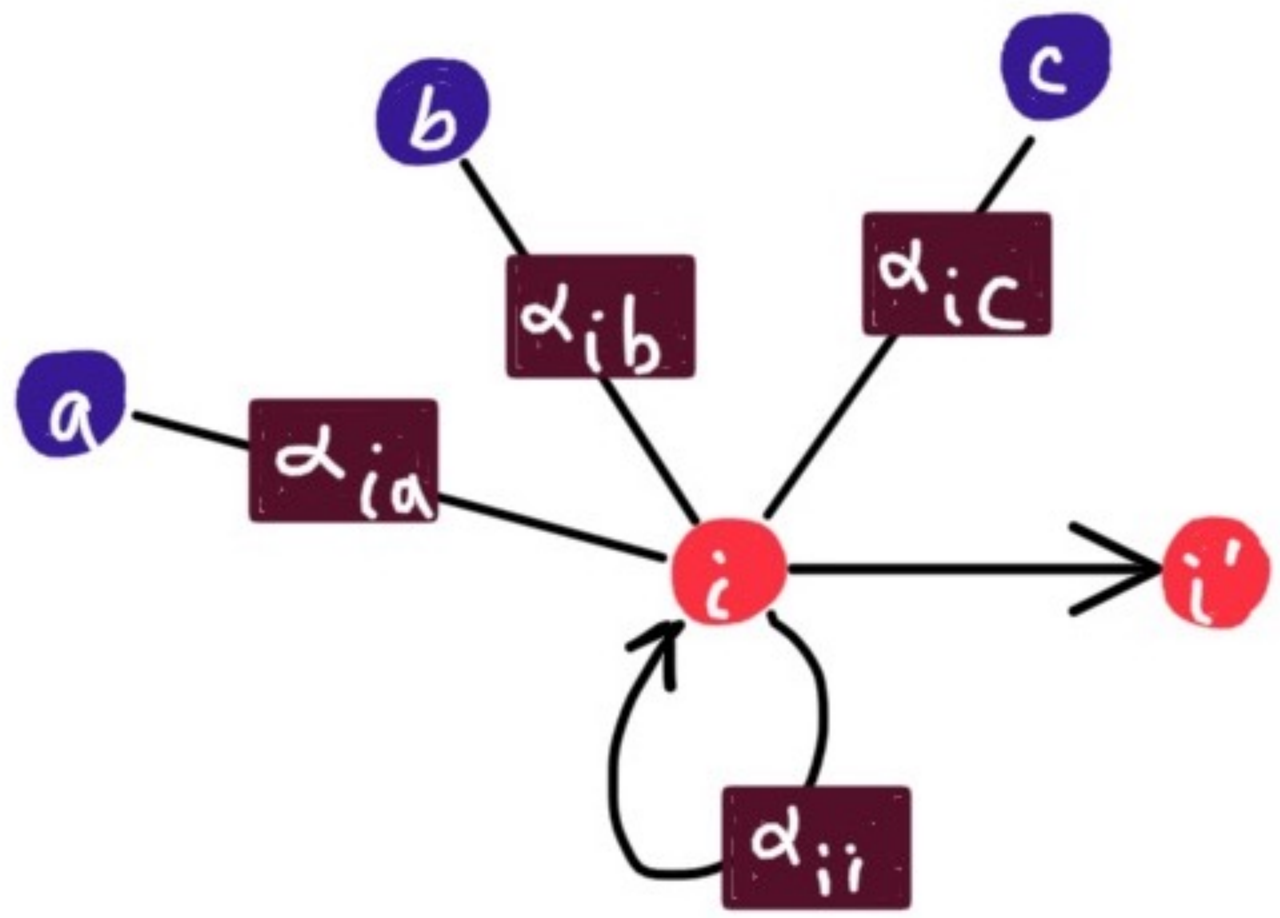
$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(\bar{a}^T [Wh_i ; Wh_j]))}{\sum_{k \in N(i)} \exp(\text{LeakyReLU}(\bar{a}^T [Wh_i ; Wh_k]))}$$

$\downarrow$

$\alpha_{ij} \in \mathbb{R}$

⑤ apply.

$$h_i' = \sigma\Big(\sum_{j \in N(i)} \alpha_{ij} W h_j\Big)$$




⑥ multi-head attention

$$h_i' = \Big\Vert_{k=1}^{K} \sigma\Big(\sum_{j \in N(i)} \alpha_{ij}^{k} W^{k} h_j\Big)$$

$\Vert$: concat

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

average on the last layer of the network

# Message Passing

$$x_i^{(k)} = \gamma^{(k)}\left(x_i^{(k-1)}, \square_{j\in N(i)}\, \phi^{(k)}\left(x_i^{(k-1)}, x_j^{(k-1)}, e_{ji}\right)\right)$$

MLP

feature repr of node i at the k-th layer

order-invariant function. For every neighbor j of node i. sum, mean, ...

MLP

feature repr of node i at (k-1)th layer

features of edge (i,j) if applicable

* PyTorch Geometric provides MessagePassing class.

aggregate() : aggr msgs from neighbors
message() : construct msg from node j to i ($\phi$)
propagate() : propagate messages
update() : update node embeddings ($\gamma$)

## Simple MessagePassing Usage

```
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...
```

Compute the message

# GCN

$$x_i^{(k)} = \underbrace{\sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{\deg(i)} \cdot \sqrt{\deg(j)}}}_{\square} \cdot \left( \underbrace{\Theta}_{\phi} \cdot x_j^{(k-1)} \right)$$

## steps to implement

1. add self-loops
2. a linear transformation to node feature matrix
3. Compute normalization coefficient
4. normalize node features
5. sum up neighboring node features

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

      N x in_channels                 2 x E
    def forward(self, x, edge_idx):
    [1] edge_idx, _ = add_self_loops(edge_idx, num_nodes=x.size(0))
    [2] x = self.lin(x)
```

```
[3] row, col = edge_idx
    deg = degree(col, x.size(0), dtype = x.dtype)
    deg_inv_sq = deg.pow(-0.5)
    norm = deg_inv_sq[row] * deg_inv_sq[col]

[4][5]  return self.propagate(edge_idx, x=x, norm=norm)

                       E x out_channels
def message(self, x_j, norm):
[4] return norm.view(-1,1) * x_j
```

# GAN

## Simplified GAN layer :

```
class GATLayer (nn.Module):
    def __init__(self, in_feats, out_feats, dropout, alpha,
                 concat = True):
        super(GATLayer, self).__init__()
        # get all input args ...

        self.W = nn.Parameter(torch.zeros((in_feats, out_feats)))
        nn.init.xavier_uniform_(self.W.data, gain=1.414)

        self.a = nn.Parameter(torch.zeros((2*out_feats, 1)))
        nn.init.xavier_uniform_(self.a.data, gain=1.414)

        self.leakyrelu = nn.LeakyReLU(self.alpha)
```

true for all but last layer

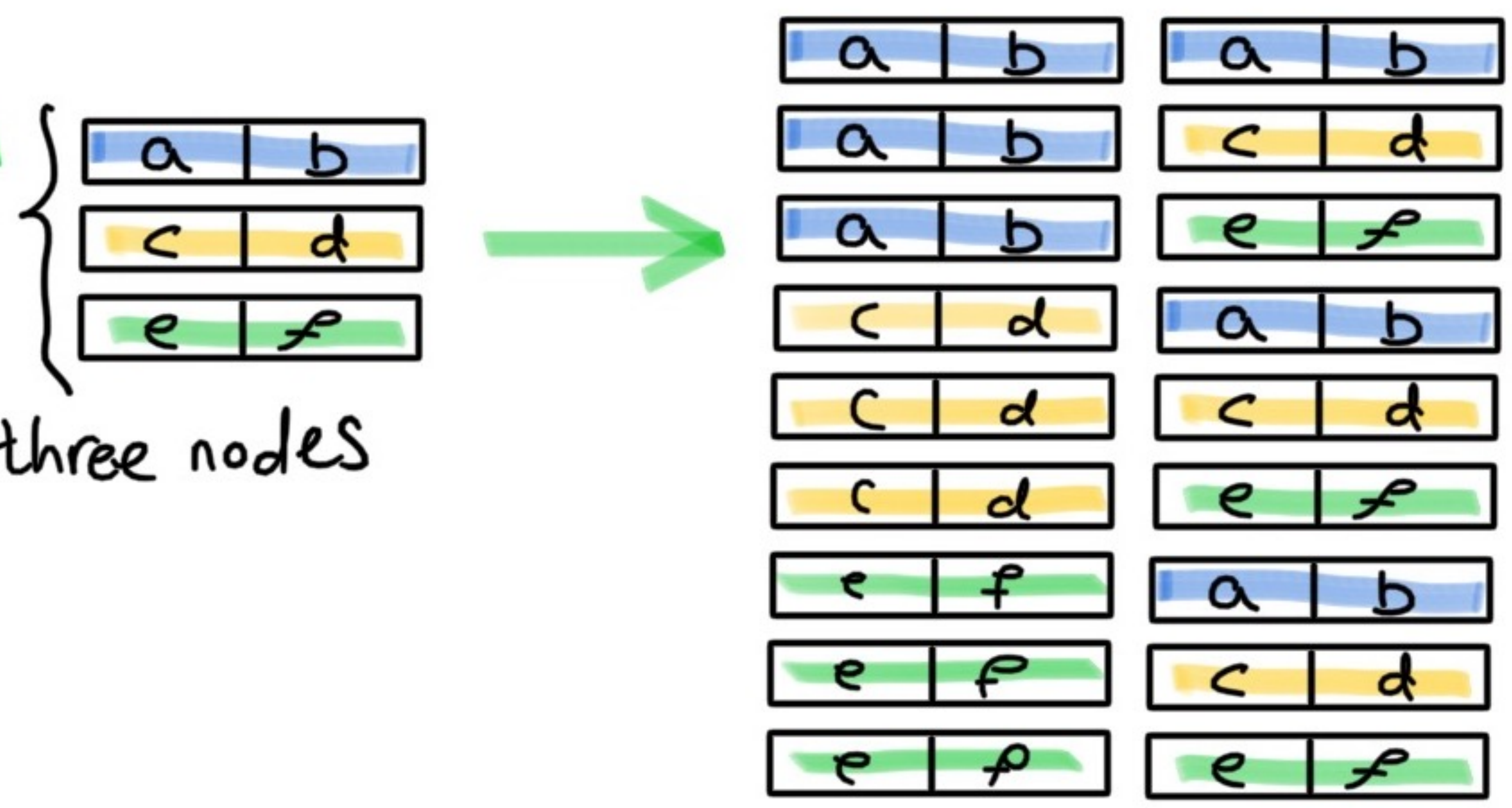
slope of LeakyReLU

```
def forward(self, input, adj):
    h = torch.mm(input, self.W)
```

→ linear transformation

attn mechanism

```
    a_inp = torch.cat([h.repeat(1,N).view(N*N,-1),
                       h.repeat(N,1)], dim=1)
              .view(N, -1, 2*self.out_feats)
```




three nodes

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

masked attn

```
    zero_vec = -9e15 * torch.ones_like(e)
    attn = torch.where(adj>0, e, zero_vec)

    attn = F.softmax(attn, dim=1)
    attn = F.dropout(attn, self.dropout, training=self.training)
    h_prime = torch.matmul(attn, h)

    if self.concat: return F.elu(h_prime)
    else: return h_prime
```

Fortunately, we don't need to implement GAN layers.

```
from torch_geometric.data import Data
from torch_geometric.nn import GATConv
from torch_geometric.datasets import Planetoid

dataset = Planetoid(root='tmp/', name='Cora')

class GAT(torch.nn.Module):
    def __init__(GAT, self).__init__()
        self.hid = 8
        self.in_head = 8
        self.out_head = 1

        self.Conv1 = GATConv(dataset.num_features, self.hid,
                             heads=self.in_head, dropout=.6)
        self.Conv2 = GATConv(self.hid * self.in_head,
                             dataset.num_classes,
                             Concat=False, heads=self.out_head,
                             dropout=.6)
```

```
def forward(self, data):
    x, edge_idx = data.x, data.edge_index

    x = f.dropout(x, p=.6, training=self.training)
    x = self.conv1(x, edge_idx)
    x = F.elu(x)
    x = f.dropout(x, p=.6, training=self.training)
    x = self.conv2(x, edge_idx)

    return F.log_softmax(x, dim=1)

model = GAT().to(device)
data = dataset[0].to(device)
opt = torch.optim.Adam(model.parameters(), lr=.005,
                       weight_decay=5e-4)
```

```
for epoch in range(epochs):
    model.train()
    opt.zero_grad()
    out = model(data)
    loss = F.nll_loss(out[data.train_mask],
                      data.y[data.train_mask])
    loss.backward()
    opt.step()
```