GNN - II

- Graph Neural Networks

Graph neural networks and its variants

- Graph convolutional network (GCN)
- Graph attention network (GAT)
- Line graph neural network (LGNN)

Graph convolutional network (GCN)

- a scalable approach for semi-supervised learning on graph-structured data that is based on convolutional neural networks.
- a localized first-order approximation of spectral graph convolutions.
- scales linearly in the number of graph edges and learns hidden layer representations that encode both local graph structure and features of nodes.

Graph convolutional network (GCN)

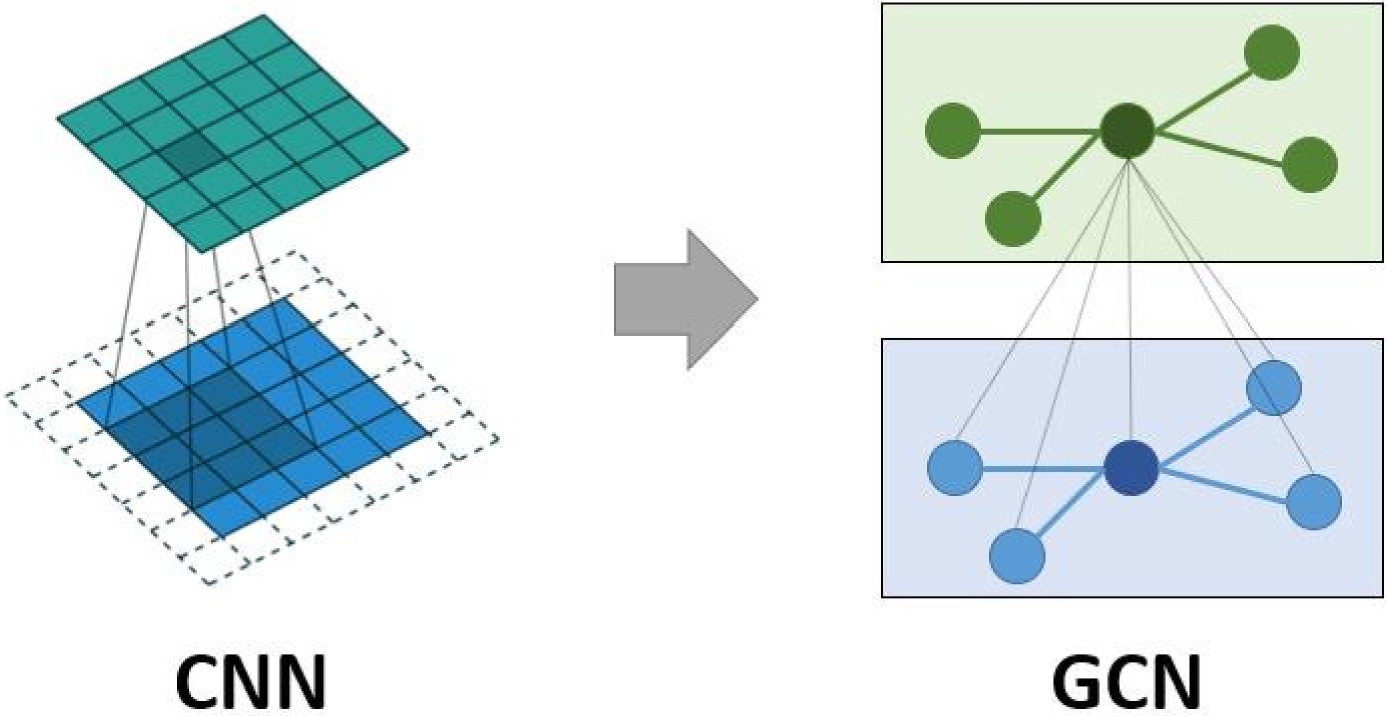
In spectral graph theory, a spectral convolution on a graph is defined as the multiplication of a signal $s \in R^N$ with a filter g in the Fourier domain, 。。。

Graph convolutional network (GCN)

We consider a multi-layer Graph Convolutional Network (GCN) with the following layer-wise propagation rule (spectral propagation):

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$$

Graph convolutional network (GCN)



$$H^{(l+1)} = \sigma\Big(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\Big)$$

1. 对邻居节点加权求和： $H^{l+1} = AH^{l}$

2. 对邻居和自己加权求和： $H^{l+1} = \hat{A}H^{l} \quad \hat{A} = A + I$

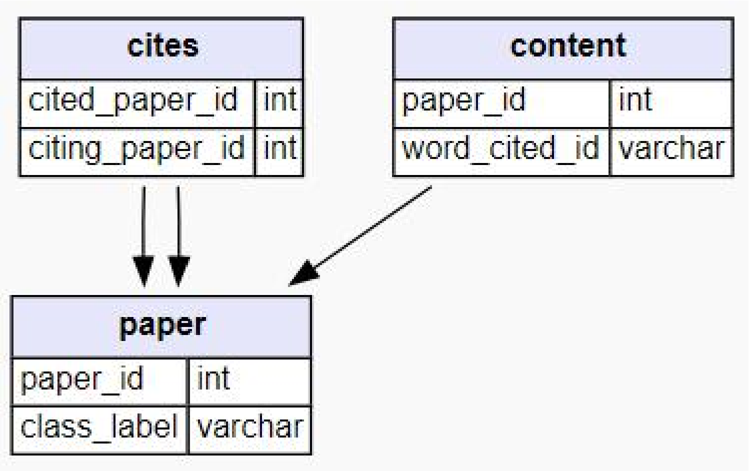
3. 加完后做平均： $H^{l+1} = \hat{D}^{-1}\hat{A}H^{l}$

4. 感觉太简单，加个track： $H^{l+1} = \hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}H^{l}$

5. DNN的权重矩阵也要： $H^{l+1} = \hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}H^{l}W^{l}$

6. 怎么能少了激活函数： $H^{l+1} = \sigma(\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}H^{l}W^{l})$

CORA Dataset



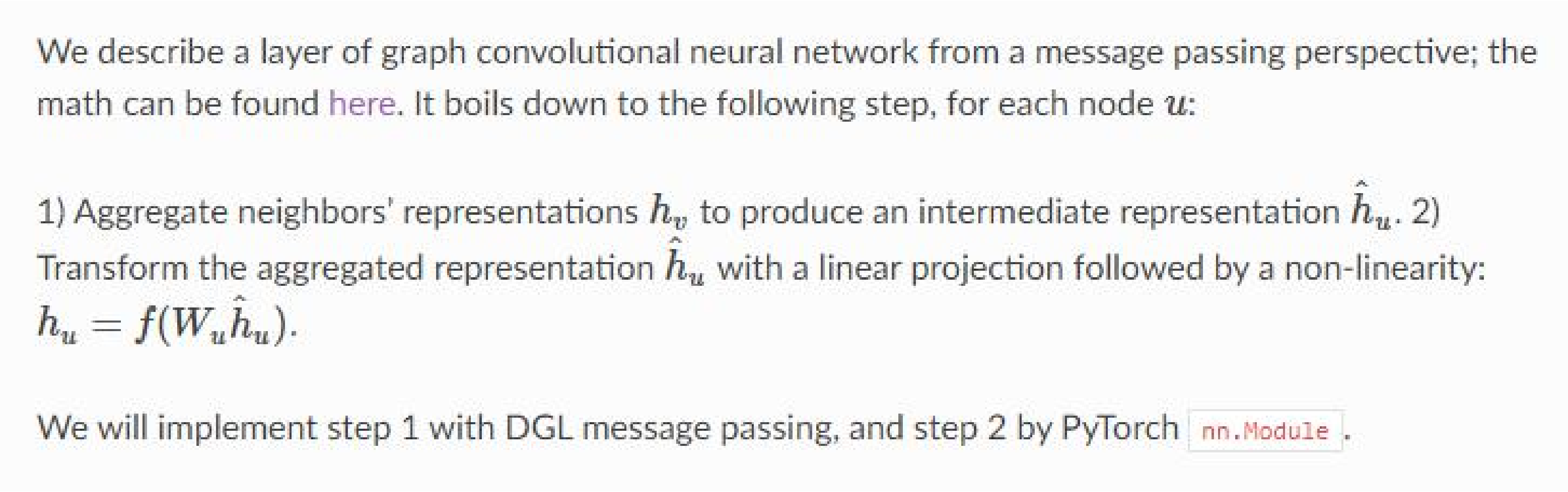
cites	
cited_paper_id	int
citing_paper_id	int

content	
paper_id	int
word_cited_id	varchar

paper	
paper_id	int
class_label	varchar

The Cora dataset consists of 2708 scientific publications classified into one of seven classes. Each publication in the dataset is described by a 0/1-valued word vector indicating the absence/presence of the corresponding word from the dictionary. The dictionary consists of 1433 unique words.

GCN from the perspective of message passing

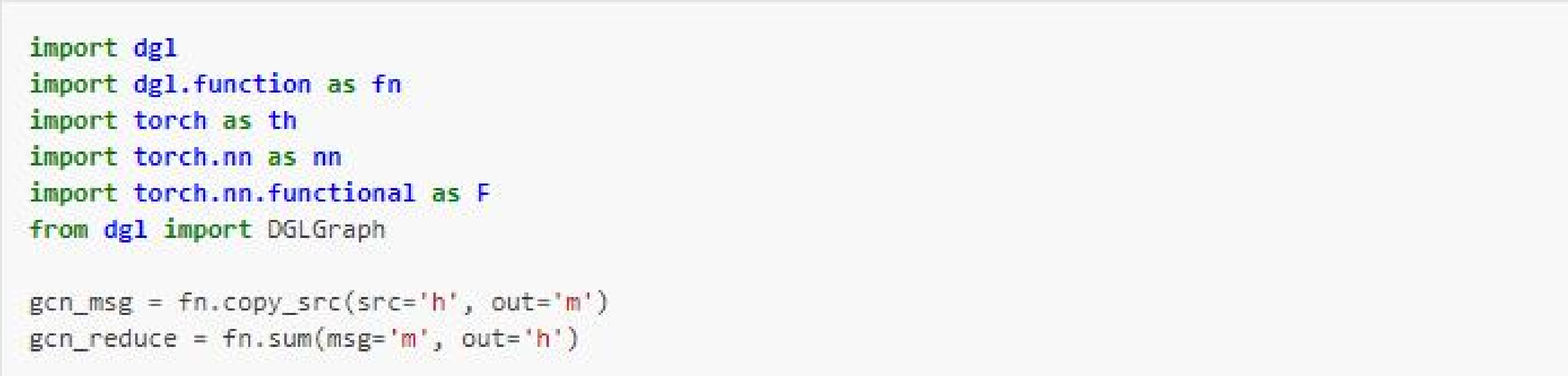


We describe a layer of graph convolutional neural network from a message passing perspective; the math can be found here. It boils down to the following step, for each node u:

1) Aggregate neighbors' representations h_v to produce an intermediate representation $\hat{h}_u$. 2) Transform the aggregated representation $\hat{h}_u$ with a linear projection followed by a non-linearity: $h_u = f(W_u \hat{h}_u)$.

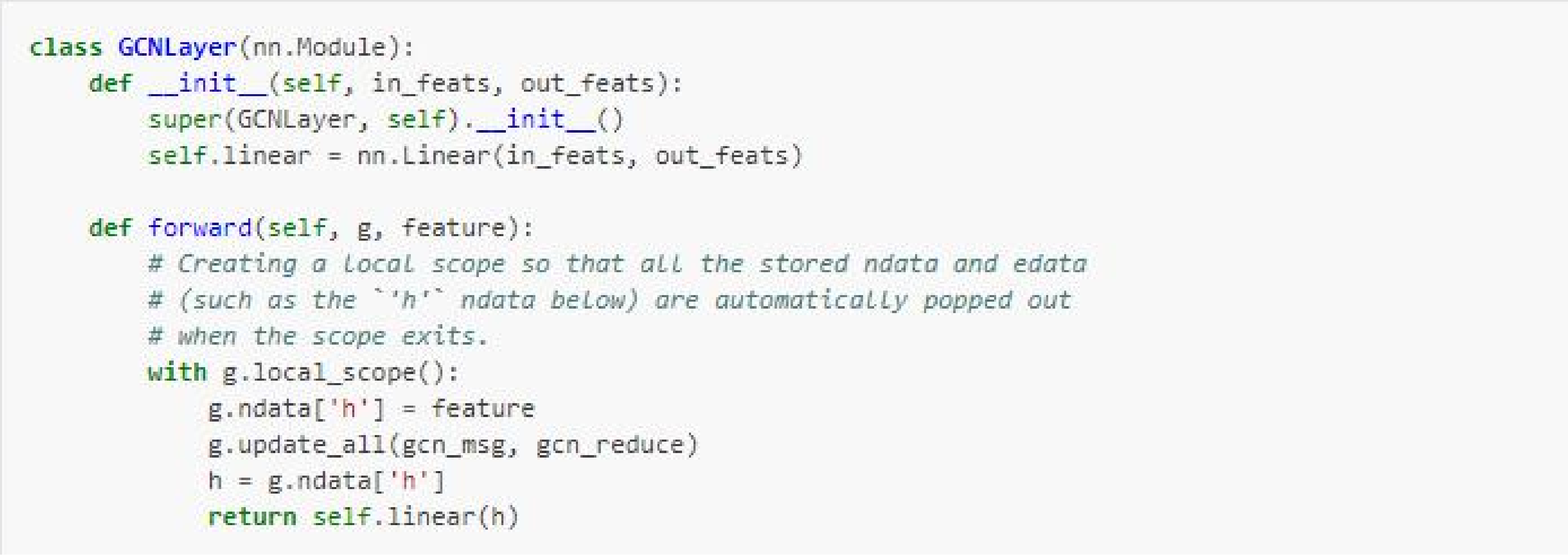
We will implement step 1 with DGL message passing, and step 2 by PyTorch `nn.Module`.

GCN from the perspective of message passing



```
import dgl
import dgl.function as fn
import torch as th
import torch.nn as nn
import torch.nn.functional as F
from dgl import DGLGraph

gcn_msg = fn.copy_src(src='h', out='m')
gcn_reduce = fn.sum(msg='m', out='h')
```

GCN from the perspective of message passing



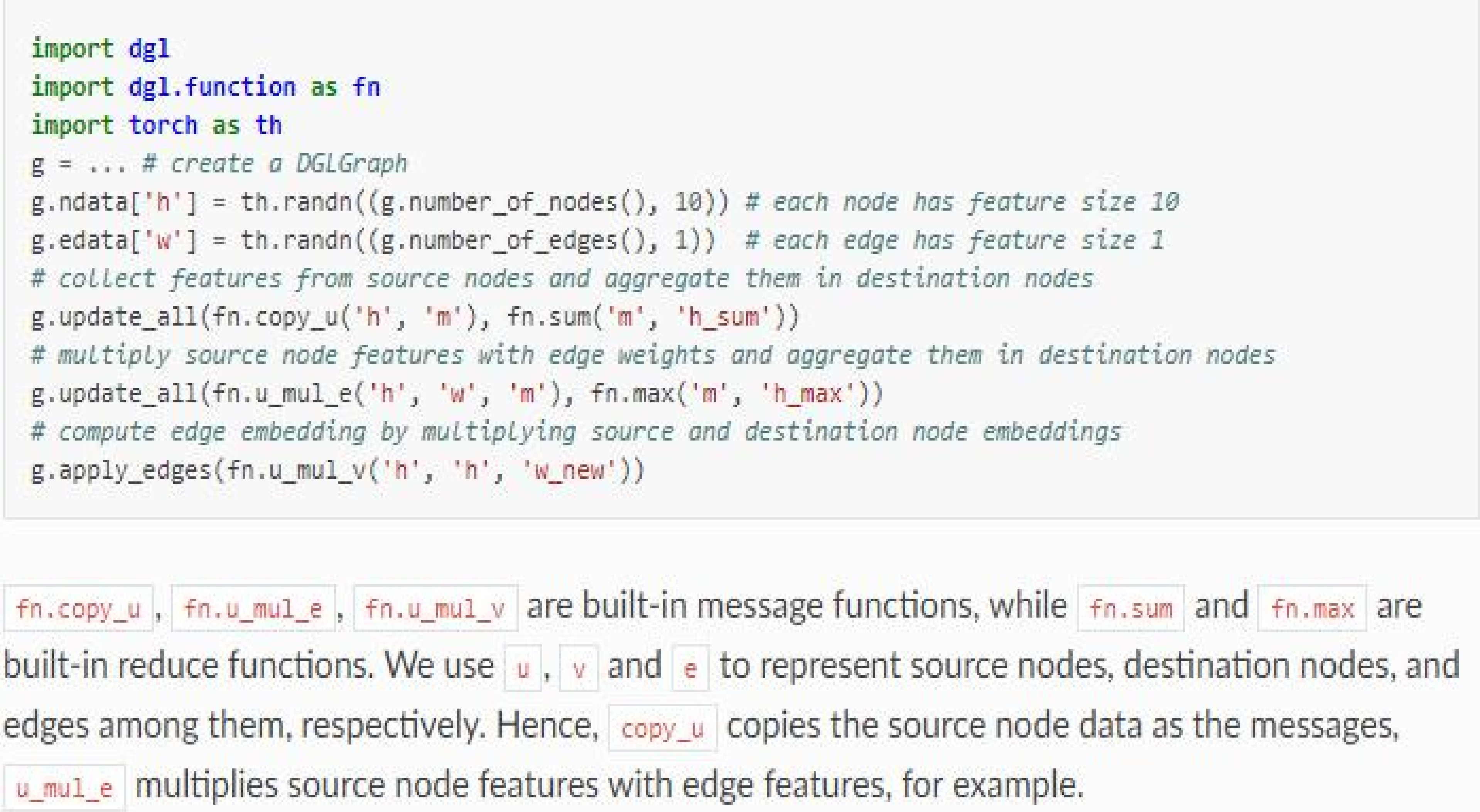
```python
class GCNLayer(nn.Module):
    def __init__(self, in_feats, out_feats):
        super(GCNLayer, self).__init__()
        self.linear = nn.Linear(in_feats, out_feats)

    def forward(self, g, feature):
        # Creating a local scope so that all the stored ndata and edata
        # (such as the `'h'` ndata below) are automatically popped out
        # when the scope exits.
        with g.local_scope():
            g.ndata['h'] = feature
            g.update_all(gcn_msg, gcn_reduce)
            h = g.ndata['h']
            return self.linear(h)
```

GCN from the perspective of message passing

In DGL, message passing is expressed by two APIs:

send(edges, message_func) for computing the messages along the given edges.
recv(nodes, reduce_func) for collecting the incoming messages, perform aggregation and so on.

Although the two-stage abstraction can cover all the models that are defined in the message passing paradigm, it is inefficient because it requires storing explicit messages. We fuse the two stages into one kernel so no explicit messages are generated and stored. To achieve this, we recommend using our built-in message and reduce functions so that DGL can analyze and map them to fused dedicated kernels.



```
import dgl
import dgl.function as fn
import torch as th
g = ... # create a DGLGraph
g.ndata['h'] = th.randn((g.number_of_nodes(), 10)) # each node has feature size 10
g.edata['w'] = th.randn((g.number_of_edges(), 1))  # each edge has feature size 1
# collect features from source nodes and aggregate them in destination nodes
g.update_all(fn.copy_u('h', 'm'), fn.sum('m', 'h_sum'))
# multiply source node features with edge weights and aggregate them in destination nodes
g.update_all(fn.u_mul_e('h', 'w', 'm'), fn.max('m', 'h_max'))
# compute edge embedding by multiplying source and destination node embeddings
g.apply_edges(fn.u_mul_v('h', 'h', 'w_new'))
```

`fn.copy_u` , `fn.u_mul_e` , `fn.u_mul_v` are built-in message functions, while `fn.sum` and `fn.max` are built-in reduce functions. We use `u` , `v` and `e` to represent source nodes, destination nodes, and edges among them, respectively. Hence, `copy_u` copies the source node data as the messages, `u_mul_e` multiplies source node features with edge features, for example.

Graph attention network (GAT)

- GAT extends the GCN functionality by deploying multi-head attention among neighborhood of a node. This greatly enhances the capacity and expressiveness of the model.

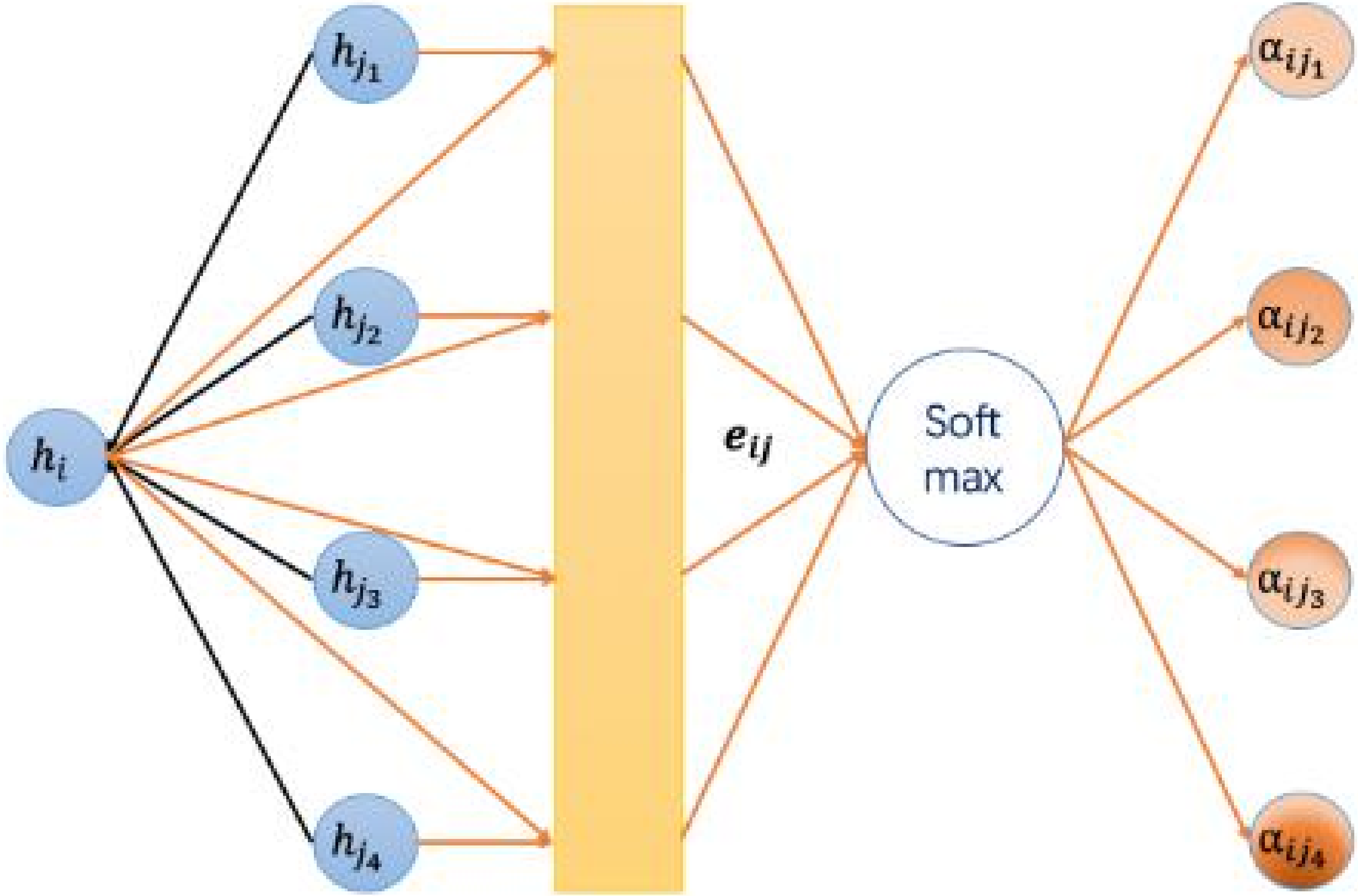
Introducing attention to GCN

- The key difference between GAT and GCN is how the information from the one-hop neighborhood is aggregated.
- For GCN, a graph convolution operation produces the normalized sum of the node features of neighbors.

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathrm{N}(i)} \frac{1}{c_{ij}} W^{(l)} h_j^{(l)}\right)$$

Introducing attention to GCN

- GAT introduces the attention mechanism as a substitute for the statically normalized convolution operation.
- Below are the equations to compute the node embedding $h^{(l+1)}_i$ of layer l+1 from the embeddings of layer l.

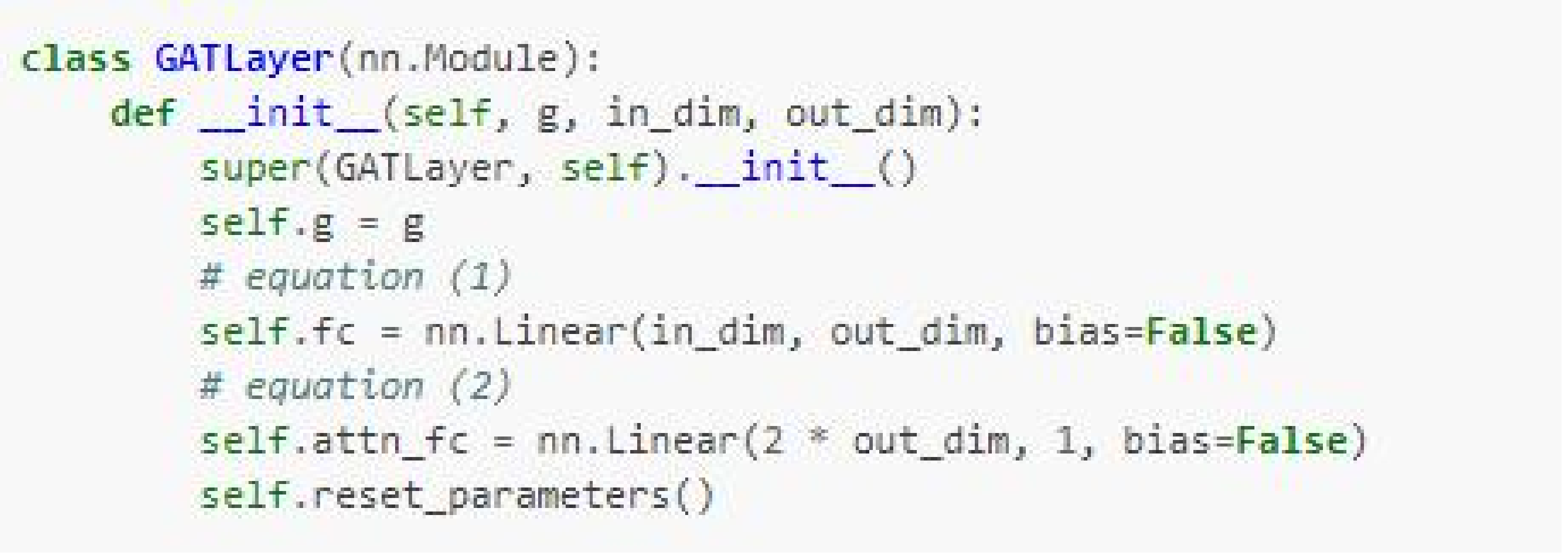

h_i
h_{j_1}
h_{j_2}
h_{j_3}
h_{j_4}
e_{ij}
Soft max
α_{ij_1}
α_{ij_2}
α_{ij_3}
α_{ij_4}

$$z_i^{(l)} = W^{(l)} h_i^{(l)}, \tag{1}$$

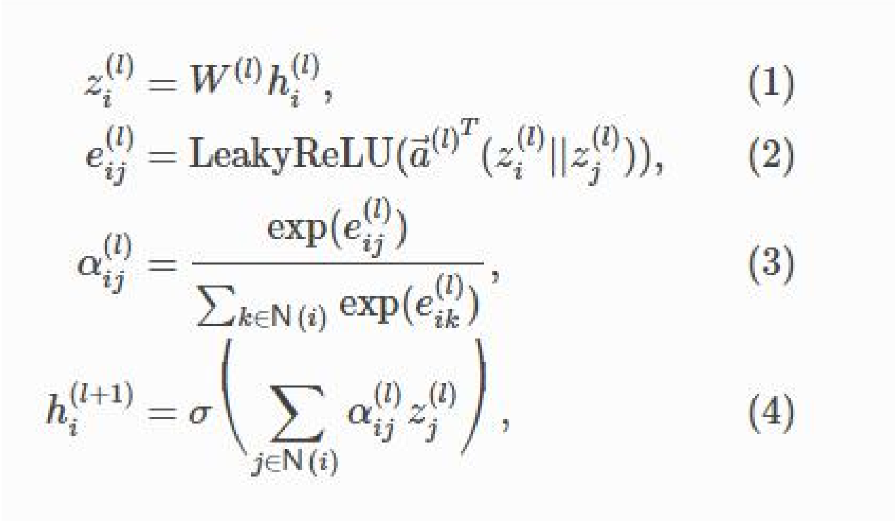
$$e_{ij}^{(l)} = \text{LeakyReLU}\left(\vec{a}^{(l)^T}\left(z_i^{(l)} \| z_j^{(l)}\right)\right), \tag{2}$$

$$\alpha_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k \in \mathrm{N}(i)} \exp(e_{ik}^{(l)})}, \tag{3}$$

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathrm{N}(i)} \alpha_{ij}^{(l)} z_j^{(l)}\right), \tag{4}$$



```
class GATLayer(nn.Module):
    def __init__(self, g, in_dim, out_dim):
        super(GATLayer, self).__init__()
        self.g = g
        # equation (1)
        self.fc = nn.Linear(in_dim, out_dim, bias=False)
        # equation (2)
        self.attn_fc = nn.Linear(2 * out_dim, 1, bias=False)
        self.reset_parameters()
```



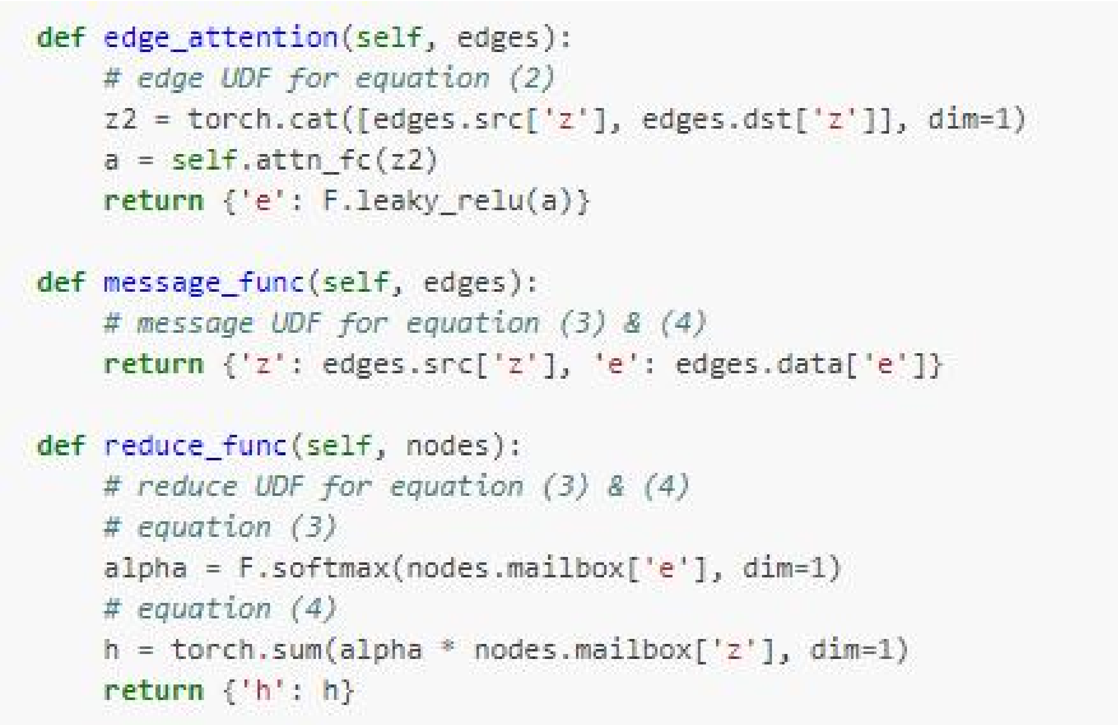
$$z_i^{(l)} = W^{(l)} h_i^{(l)}, \tag{1}$$

$$e_{ij}^{(l)} = \text{LeakyReLU}(\vec{a}^{(l)^T}(z_i^{(l)} || z_j^{(l)})), \tag{2}$$

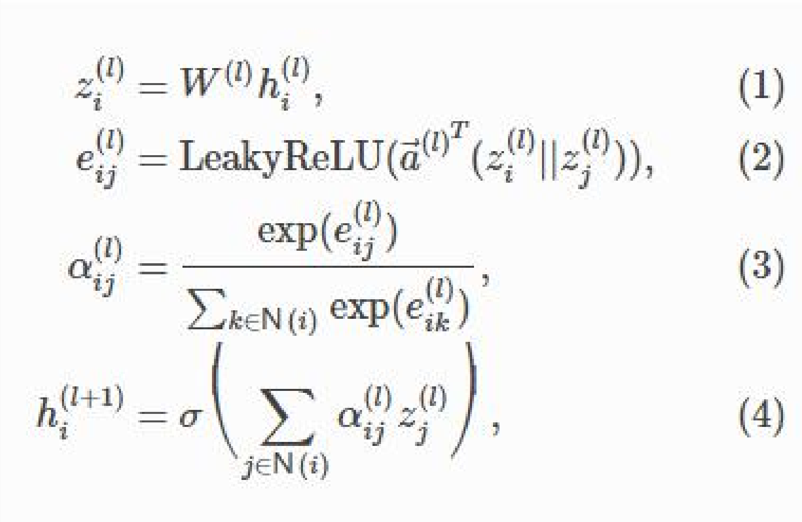
$$\alpha_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k \in \mathrm{N}(i)} \exp(e_{ik}^{(l)})}, \tag{3}$$

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathrm{N}(i)} \alpha_{ij}^{(l)} z_j^{(l)}\right), \tag{4}$$

Introducing attention to GCN



```
def edge_attention(self, edges):
    # edge UDF for equation (2)
    z2 = torch.cat([edges.src['z'], edges.dst['z']], dim=1)
    a = self.attn_fc(z2)
    return {'e': F.leaky_relu(a)}

def message_func(self, edges):
    # message UDF for equation (3) & (4)
    return {'z': edges.src['z'], 'e': edges.data['e']}

def reduce_func(self, nodes):
    # reduce UDF for equation (3) & (4)
    # equation (3)
    alpha = F.softmax(nodes.mailbox['e'], dim=1)
    # equation (4)
    h = torch.sum(alpha * nodes.mailbox['z'], dim=1)
    return {'h': h}
```

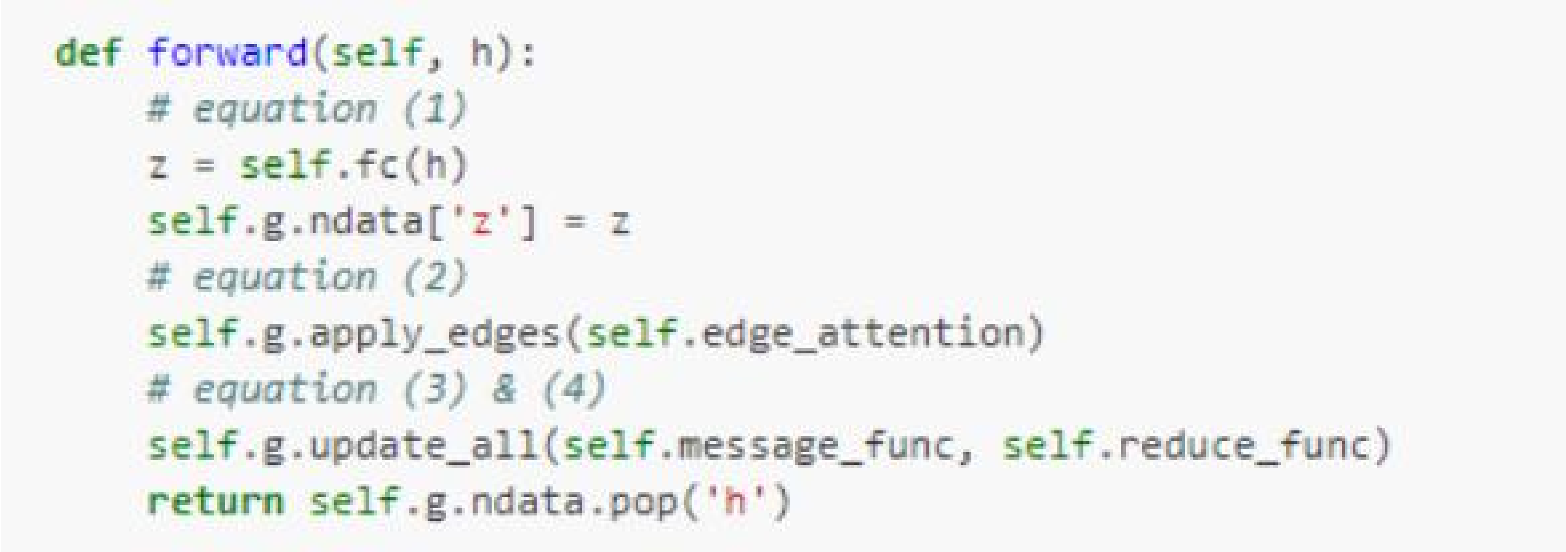


$$z_i^{(l)} = W^{(l)} h_i^{(l)}, \tag{1}$$

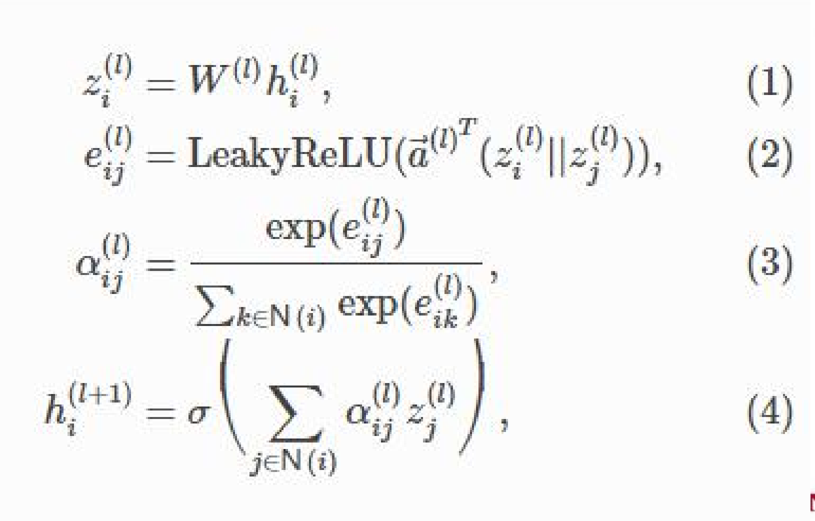
$$e_{ij}^{(l)} = \mathrm{LeakyReLU}(\vec{a}^{(l)^T}(z_i^{(l)} || z_j^{(l)})), \tag{2}$$

$$\alpha_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k \in \mathrm{N}(i)} \exp(e_{ik}^{(l)})}, \tag{3}$$

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathrm{N}(i)} \alpha_{ij}^{(l)} z_j^{(l)}\right), \tag{4}$$



```python
def forward(self, h):
    # equation (1)
    z = self.fc(h)
    self.g.ndata['z'] = z
    # equation (2)
    self.g.apply_edges(self.edge_attention)
    # equation (3) & (4)
    self.g.update_all(self.message_func, self.reduce_func)
    return self.g.ndata.pop('h')
```



$$z_i^{(l)} = W^{(l)} h_i^{(l)}, \tag{1}$$

$$e_{ij}^{(l)} = \text{LeakyReLU}(\vec{a}^{(l)^T}(z_i^{(l)} || z_j^{(l)})), \tag{2}$$

$$\alpha_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k \in \mathrm{N}(i)} \exp(e_{ik}^{(l)})}, \tag{3}$$

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathrm{N}(i)} \alpha_{ij}^{(l)} z_j^{(l)}\right), \tag{4}$$

Introducing attention to GCN

- GAT is just a different aggregation function with attention over features of neighbors, instead of a simple mean aggregation.

$$h_i^{(l+1)} = \sigma\left(\sum_{j\in \mathrm{N}(i)} \frac{1}{c_{ij}} W^{(l)} h_j^{(l)}\right)$$

$$\Longrightarrow$$

$$z_i^{(l)} = W^{(l)} h_i^{(l)}, \tag{1}$$

$$e_{ij}^{(l)} = \mathrm{LeakyReLU}(\vec{a}^{(l)^T}(z_i^{(l)} || z_j^{(l)})), \tag{2}$$

$$\alpha_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k\in \mathrm{N}(i)} \exp(e_{ik}^{(l)})}, \tag{3}$$

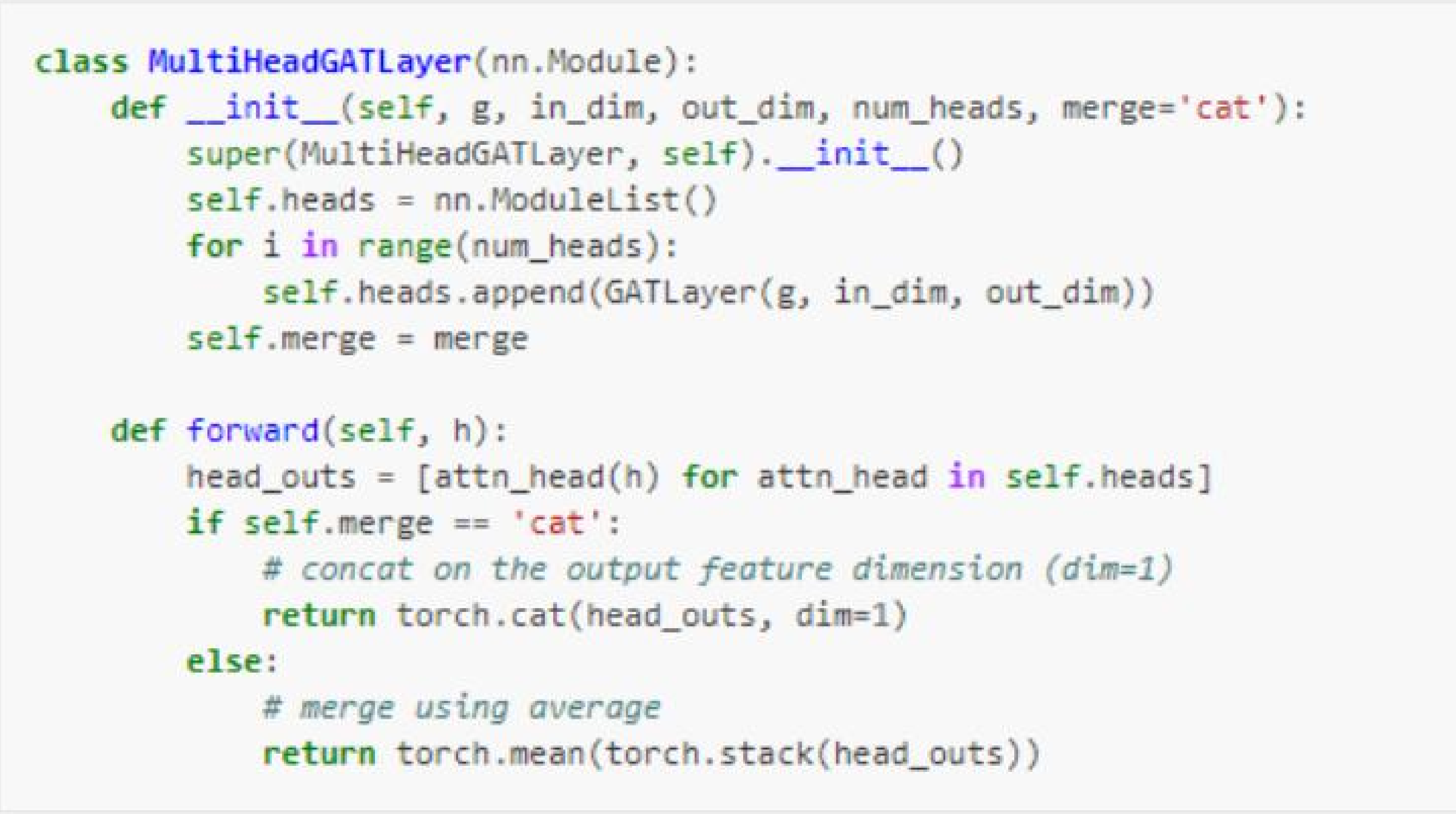
$$h_i^{(l+1)} = \sigma\left(\sum_{j\in \mathrm{N}(i)} \alpha_{ij}^{(l)} z_j^{(l)}\right), \tag{4}$$

Multi-head attention

- Analogous to multiple channels in ConvNet, GAT introduces multi-head attention to enrich the model capacity and to stabilize the learning process. Each attention head has its own parameters and their outputs can be merged in two ways:

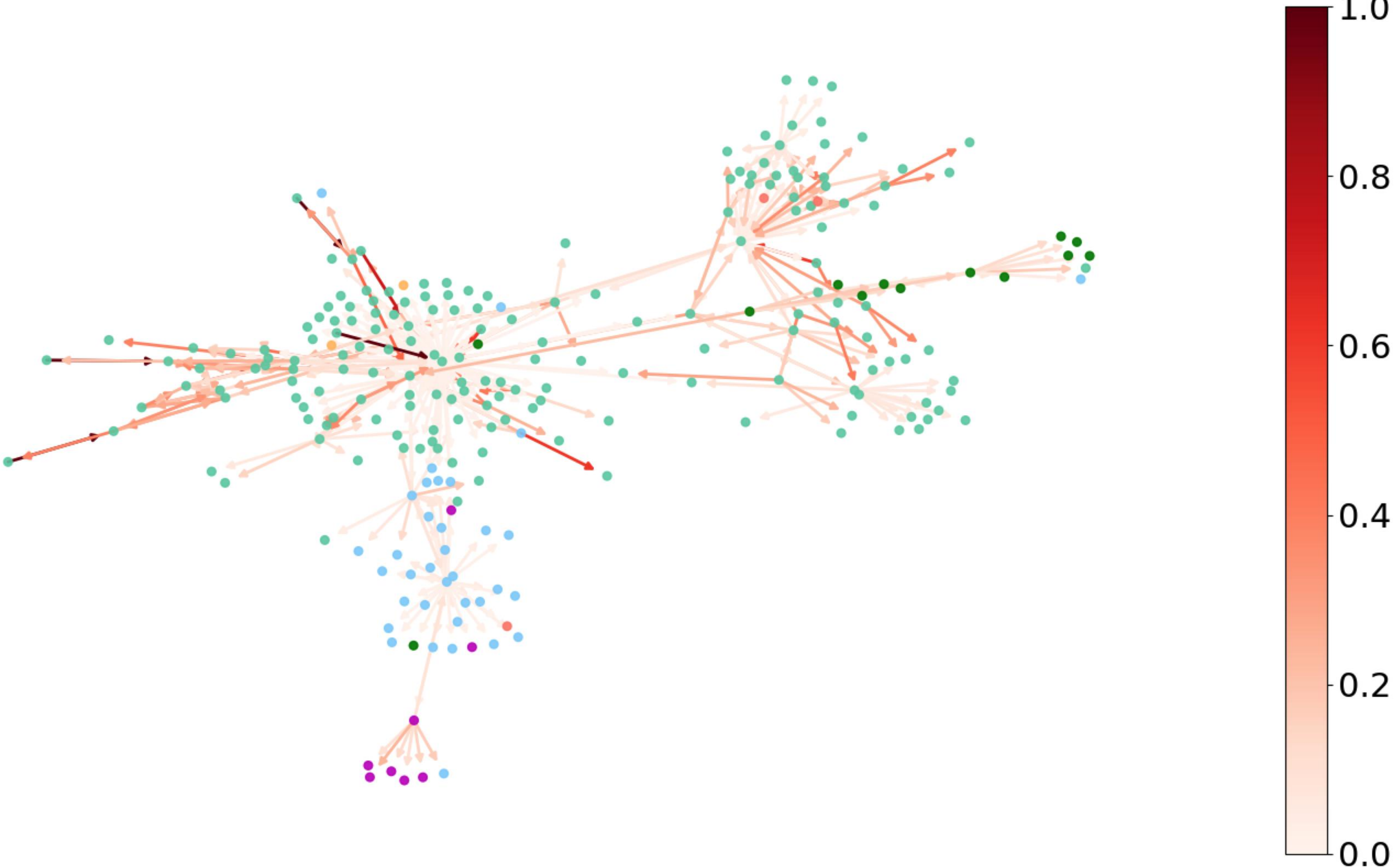
$$\text{concatenation} : h_i^{(l+1)} = ||_{k=1}^{K} \sigma \left(\sum_{j \in \mathrm{N}(i)} \alpha_{ij}^{k} W^{k} h_j^{(l)} \right)$$

$$\text{average} : h_i^{(l+1)} = \sigma \left(\frac{1}{K} \sum_{k=1}^{K} \sum_{j \in \mathrm{N}(i)} \alpha_{ij}^{k} W^{k} h_j^{(l)} \right)$$



```python
class MultiHeadGATLayer(nn.Module):
    def __init__(self, g, in_dim, out_dim, num_heads, merge='cat'):
        super(MultiHeadGATLayer, self).__init__()
        self.heads = nn.ModuleList()
        for i in range(num_heads):
            self.heads.append(GATLayer(g, in_dim, out_dim))
        self.merge = merge

    def forward(self, h):
        head_outs = [attn_head(h) for attn_head in self.heads]
        if self.merge == 'cat':
            # concat on the output feature dimension (dim=1)
            return torch.cat(head_outs, dim=1)
        else:
            # merge using average
            return torch.mean(torch.stack(head_outs))
```

final



Line graph neural network (LGNN)

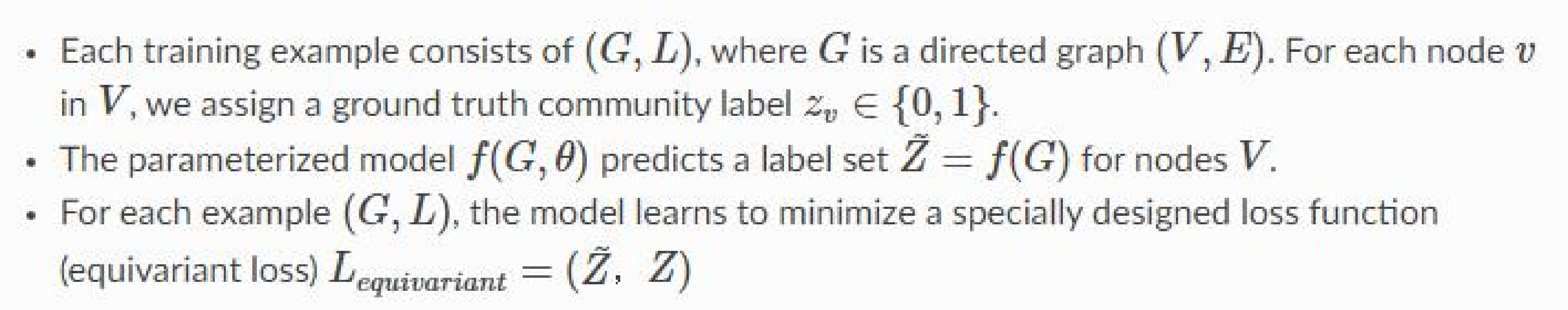
- This network focuses on community detection by inspecting graph structures. It uses representations of both the original graph and its line-graph companion. In addition to demonstrating how an algorithm can harness multiple graphs, this implementation shows how you can judiciously mix simple tensor operations and sparse-matrix tensor operations, along with message-passing with DGL.

Supervised community detection

- In a community detection task, you cluster similar nodes instead of labeling them. The node similarity is typically described as having higher inner density within each cluster.
- What’ s the difference between community detection and node classification? Comparing to node classification, community detection focuses on retrieving cluster information in the graph, rather than assigning a specific label to a node.

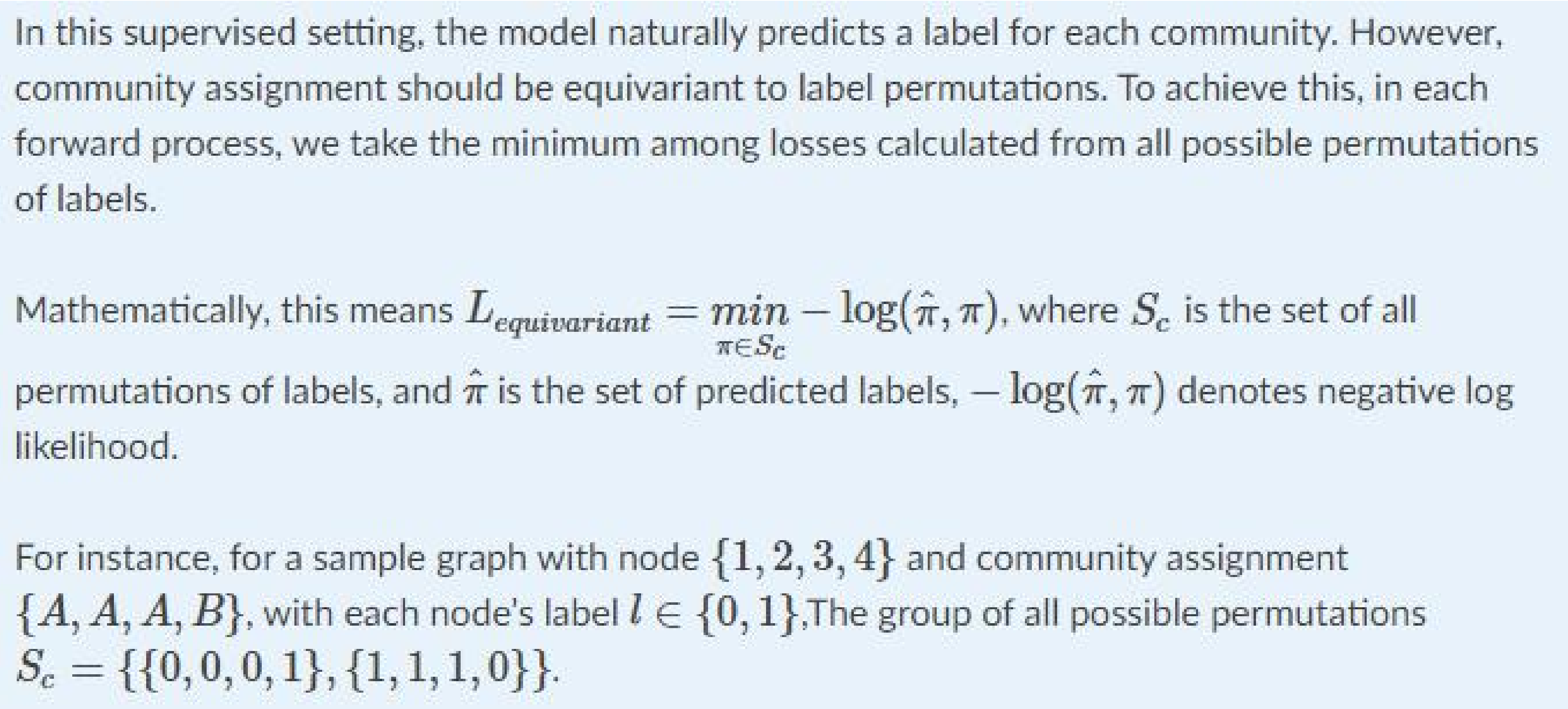
Community detection in a supervised setting

The community detection problem could be tackled with both supervised and unsupervised approaches. You can formulate community detection in a supervised setting as follows:



- Each training example consists of (G, L), where G is a directed graph (V, E). For each node v in V, we assign a ground truth community label $z_v \in \{0, 1\}$.
- The parameterized model $f(G, \theta)$ predicts a label set $\tilde{Z} = f(G)$ for nodes V.
- For each example (G, L), the model learns to minimize a specially designed loss function (equivariant loss) $L_{equivariant} = (\tilde{Z},\ Z)$

Community detection in a supervised setting



In this supervised setting, the model naturally predicts a label for each community. However, community assignment should be equivariant to label permutations. To achieve this, in each forward process, we take the minimum among losses calculated from all possible permutations of labels.

Mathematically, this means $L_{equivariant} = \underset{\pi \in S_c}{min} - \log(\hat{\pi}, \pi)$, where S_c is the set of all permutations of labels, and $\hat{\pi}$ is the set of predicted labels, $-\log(\hat{\pi}, \pi)$ denotes negative log likelihood.

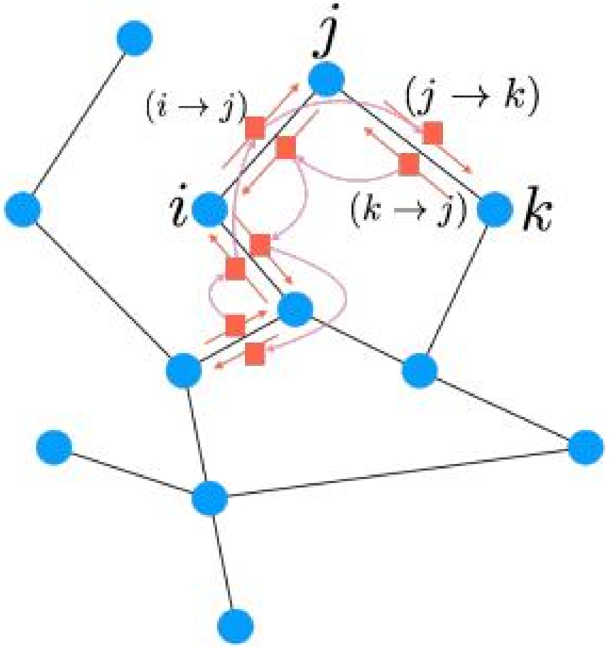
For instance, for a sample graph with node $\{1, 2, 3, 4\}$ and community assignment $\{A, A, A, B\}$, with each node's label $l \in \{0, 1\}$.The group of all possible permutations $S_c = \{\{0, 0, 0, 1\}, \{1, 1, 1, 0\}\}$.

Line graph neural network key ideas

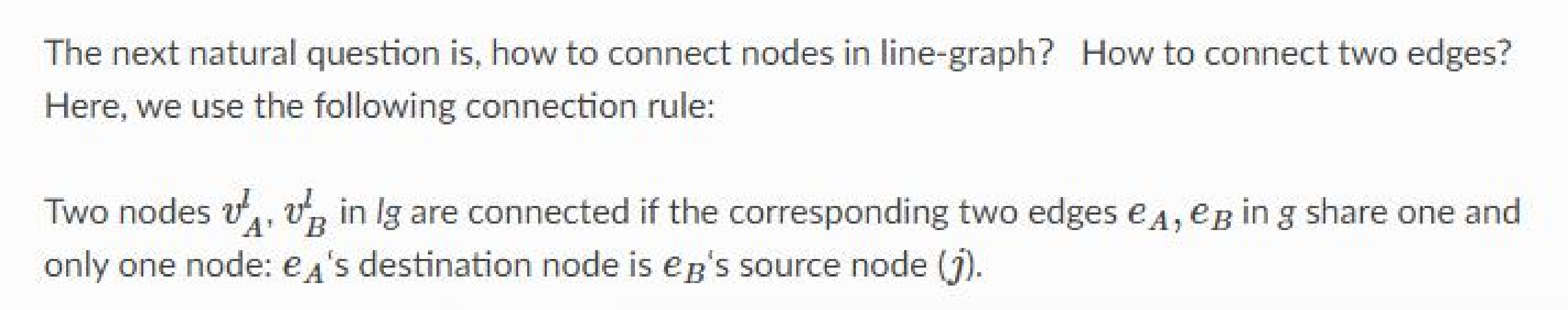
An key innovation in this topic is the use of a line graph. In models of line graph, message passing happens not only on the original graph, e.g. the binary community subgraph from Cora, but also on the line graph associated with the original graph.

What is a line-graph?

Specifically, a line-graph L(G) turns an edge of the original graph G into a node. This is illustrated with the graph below.

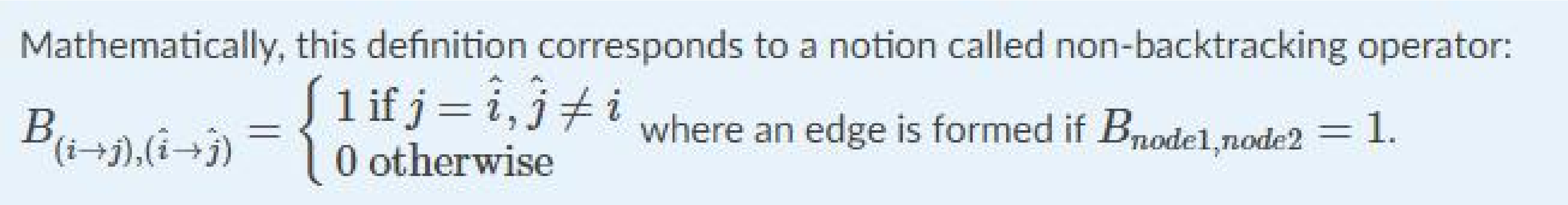


What is a line-graph?



The next natural question is, how to connect nodes in line-graph? How to connect two edges? Here, we use the following connection rule:

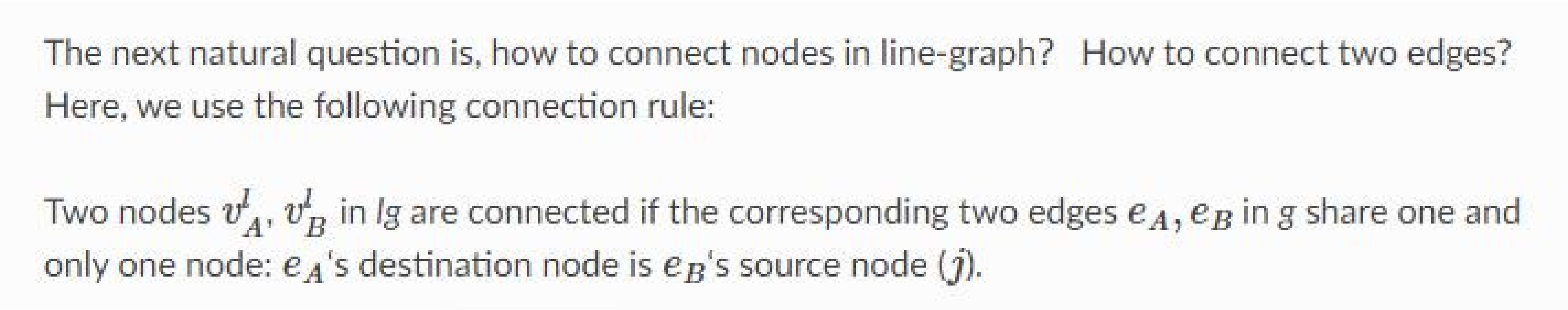
Two nodes v^l_A, v^l_B in *lg* are connected if the corresponding two edges e_A, e_B in *g* share one and only one node: e_A's destination node is e_B's source node (j).



Mathematically, this definition corresponds to a notion called non-backtracking operator:

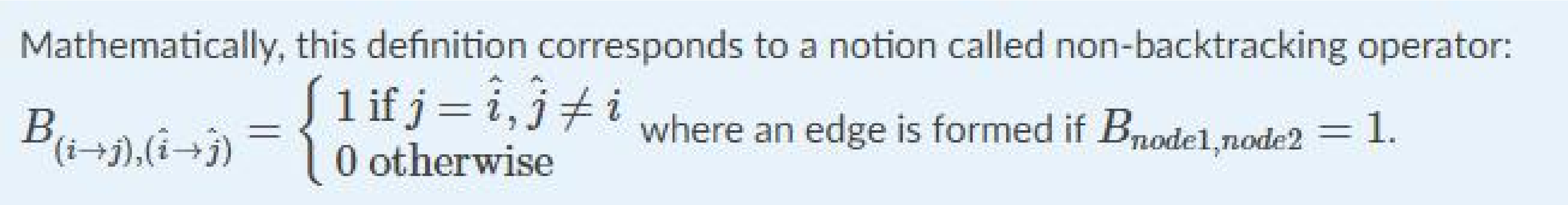
$B_{(i\to j),(\hat{i}\to\hat{j})} = \begin{cases} 1 \text{ if } j=\hat{i}, \hat{j}\neq i \\ 0 \text{ otherwise} \end{cases}$ where an edge is formed if $B_{node1,node2} = 1$.

What is a line-graph?



The next natural question is, how to connect nodes in line-graph? How to connect two edges? Here, we use the following connection rule:

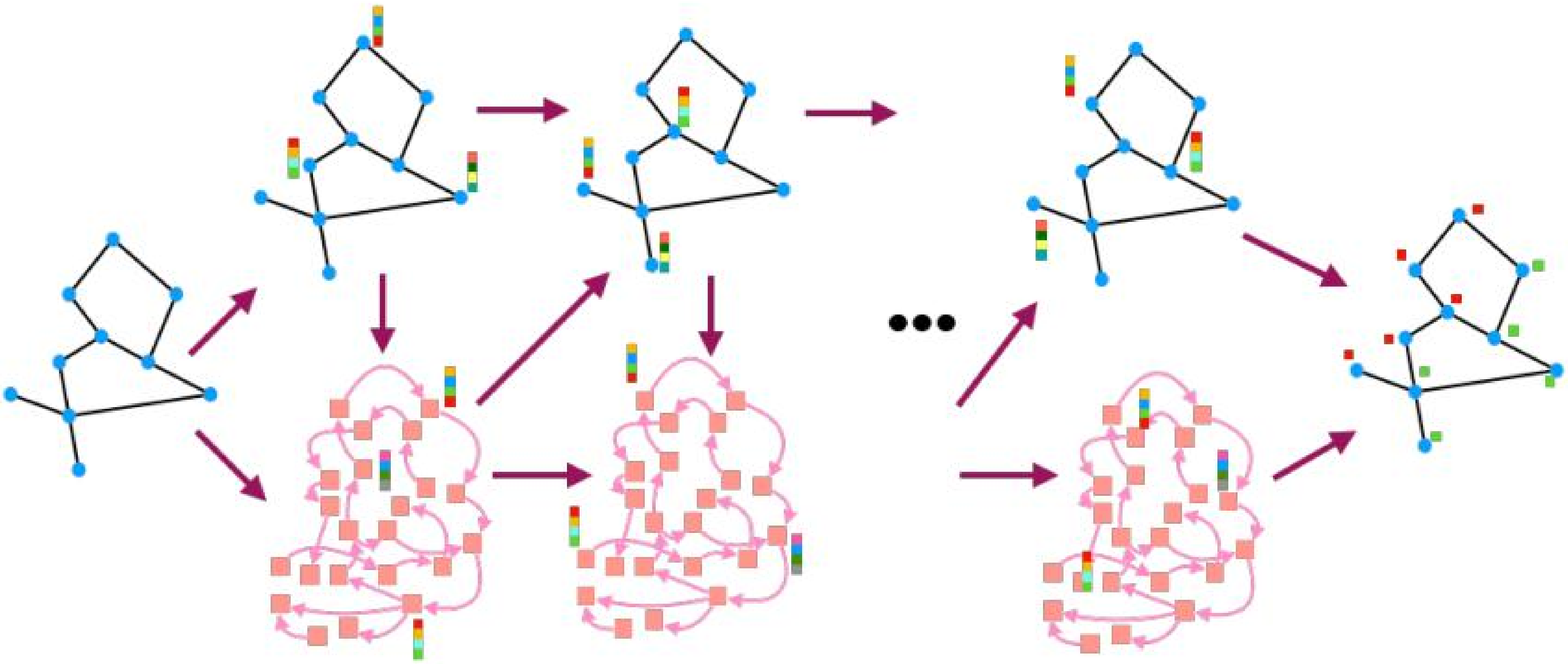
Two nodes v^l_A, v^l_B in *lg* are connected if the corresponding two edges e_A, e_B in *g* share one and only one node: e_A's destination node is e_B's source node (j).



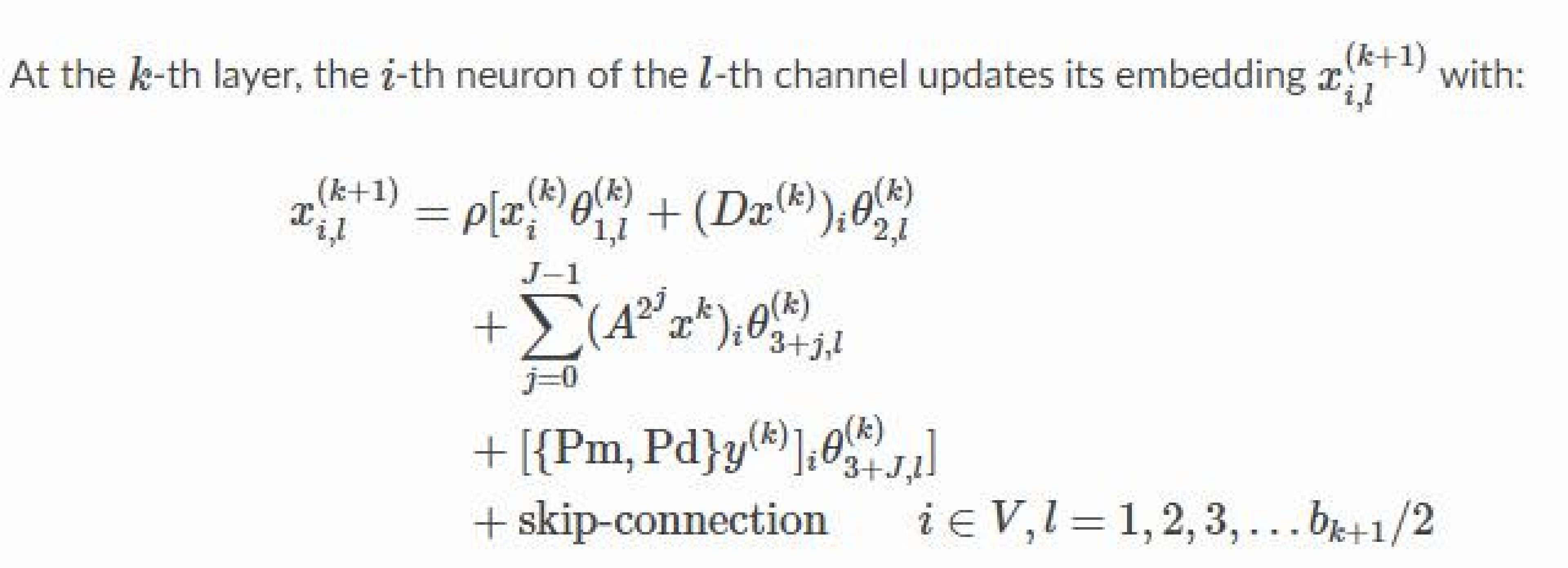
Mathematically, this definition corresponds to a notion called non-backtracking operator:

$B_{(i\to j),(\hat{i}\to\hat{j})} = \begin{cases} 1 \text{ if } j=\hat{i}, \hat{j}\neq i \\ 0 \text{ otherwise} \end{cases}$ where an edge is formed if $B_{node1,node2} = 1$.

One layer in LGNN, algorithm structure



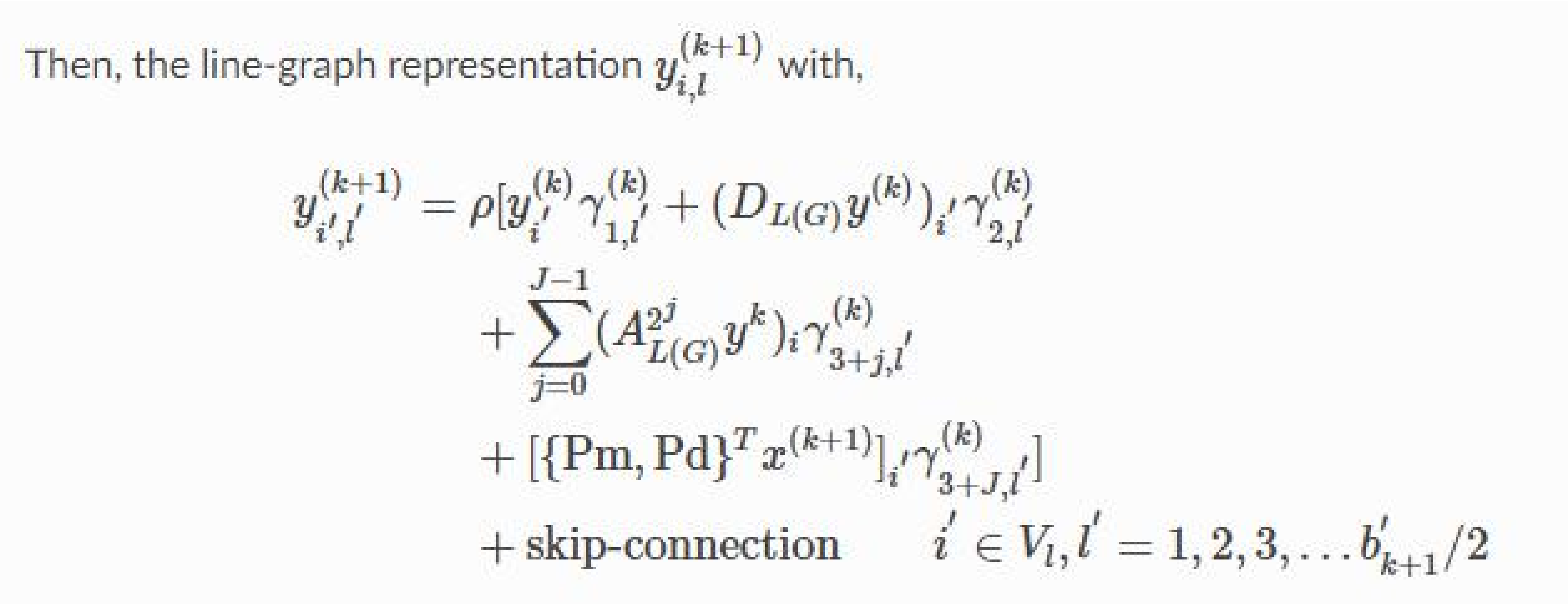
One layer in LGNN, algorithm structure



At the k-th layer, the i-th neuron of the l-th channel updates its embedding $x_{i,l}^{(k+1)}$ with:

$$
\begin{aligned}
x_{i,l}^{(k+1)} = \rho[x_i^{(k)}\theta_{1,l}^{(k)} &+ (Dx^{(k)})_i\theta_{2,l}^{(k)} \\
&+ \sum_{j=0}^{J-1}(A^{2^j}x^k)_i\theta_{3+j,l}^{(k)} \\
&+ [\{\mathrm{Pm},\mathrm{Pd}\}y^{(k)}]_i\theta_{3+J,l}^{(k)}] \\
&+ \text{skip-connection} \qquad i \in V, l = 1,2,3,\ldots b_{k+1}/2
\end{aligned}
$$

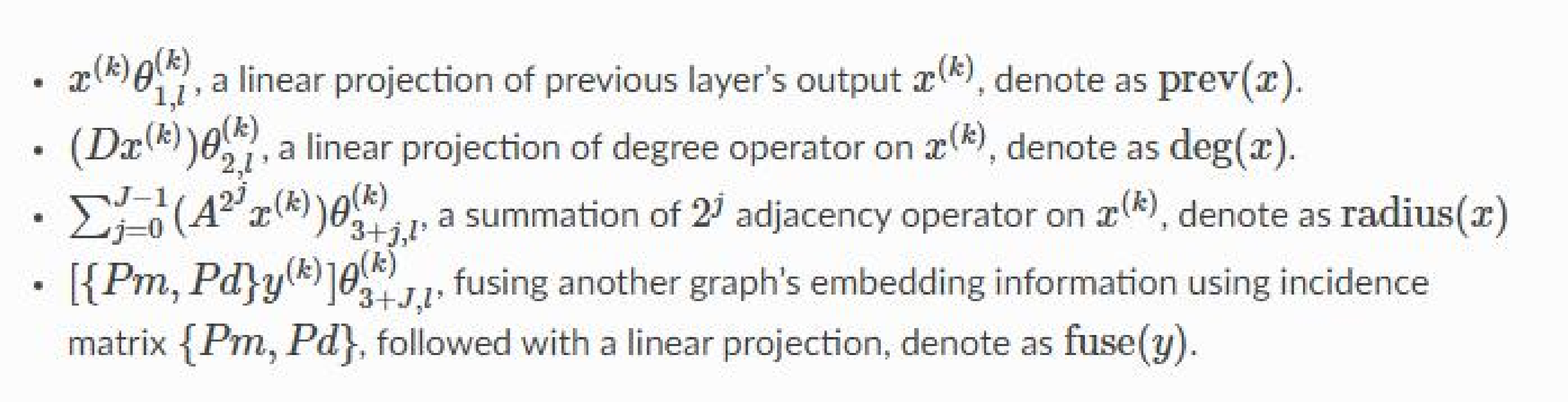
One layer in LGNN, algorithm structure



Then, the line-graph representation $y_{i,l}^{(k+1)}$ with,

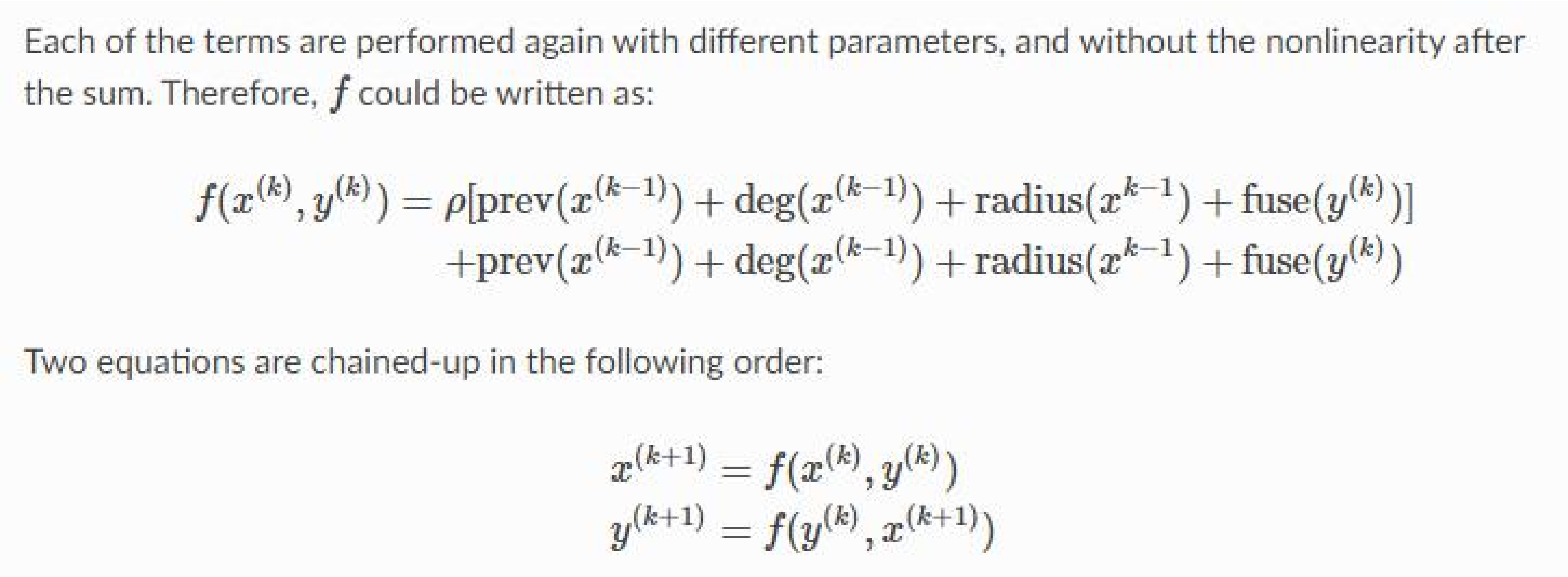
$$\begin{aligned} y_{i',l'}^{(k+1)} = \rho[&y_{i'}^{(k)}\gamma_{1,l'}^{(k)} + (D_{L(G)}y^{(k)})_{i'}\gamma_{2,l'}^{(k)} \\ &+ \sum_{j=0}^{J-1}(A_{L(G)}^{2^j}y^{k})_i\gamma_{3+j,l'}^{(k)} \\ &+ [\{\mathrm{Pm},\mathrm{Pd}\}^T x^{(k+1)}]_{i'}\gamma_{3+J,l'}^{(k)}] \\ &+ \text{skip-connection} \qquad i' \in V_l, l' = 1,2,3,\dots b'_{k+1}/2 \end{aligned}$$

Implement LGNN in DGL



- $x^{(k)}\theta^{(k)}_{1,l}$, a linear projection of previous layer's output $x^{(k)}$, denote as $\text{prev}(x)$.
- $(Dx^{(k)})\theta^{(k)}_{2,l}$, a linear projection of degree operator on $x^{(k)}$, denote as $\text{deg}(x)$.
- $\sum_{j=0}^{J-1}(A^{2^j}x^{(k)})\theta^{(k)}_{3+j,l}$, a summation of 2^j adjacency operator on $x^{(k)}$, denote as $\text{radius}(x)$
- $[\{Pm, Pd\}y^{(k)}]\theta^{(k)}_{3+J,l}$, fusing another graph's embedding information using incidence matrix $\{Pm, Pd\}$, followed with a linear projection, denote as $\text{fuse}(y)$.

Implement LGNN in DGL



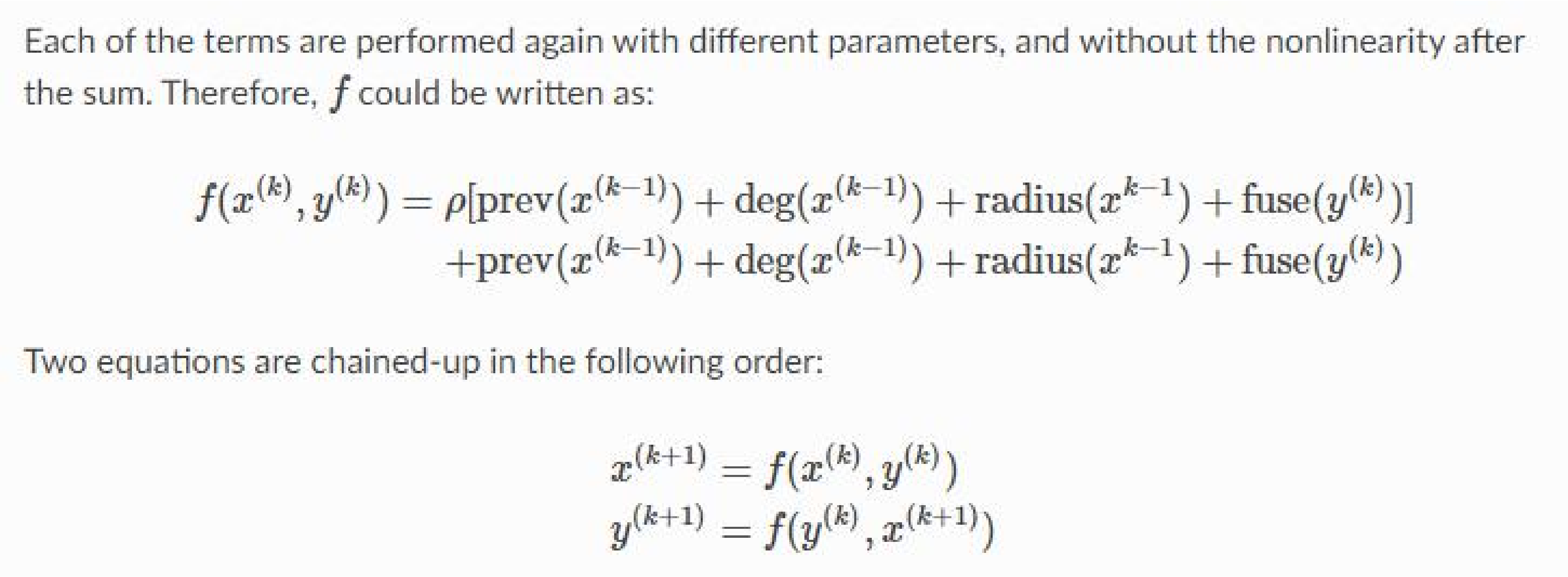
Each of the terms are performed again with different parameters, and without the nonlinearity after the sum. Therefore, f could be written as:

$$\begin{aligned} f(x^{(k)}, y^{(k)}) = \rho[&\text{prev}(x^{(k-1)}) + \text{deg}(x^{(k-1)}) + \text{radius}(x^{k-1}) + \text{fuse}(y^{(k)})] \\ &+\text{prev}(x^{(k-1)}) + \text{deg}(x^{(k-1)}) + \text{radius}(x^{k-1}) + \text{fuse}(y^{(k)}) \end{aligned}$$

Two equations are chained-up in the following order:

$$\begin{aligned} x^{(k+1)} &= f(x^{(k)}, y^{(k)}) \\ y^{(k+1)} &= f(y^{(k)}, x^{(k+1)}) \end{aligned}$$

Implementing prev and deg as tensor operation



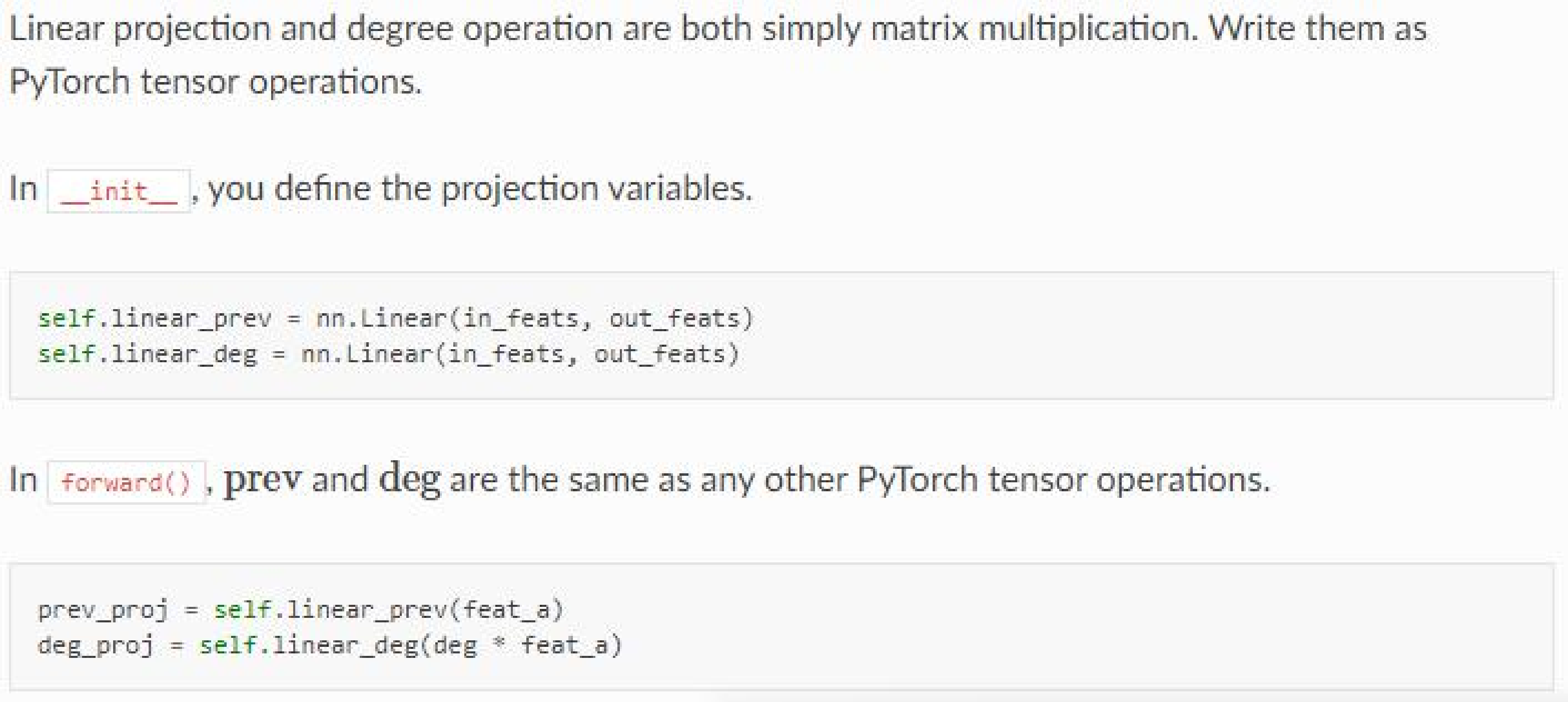
Each of the terms are performed again with different parameters, and without the nonlinearity after the sum. Therefore, f could be written as:

$$\begin{aligned} f(x^{(k)}, y^{(k)}) = \rho[&\mathrm{prev}(x^{(k-1)}) + \mathrm{deg}(x^{(k-1)}) + \mathrm{radius}(x^{k-1}) + \mathrm{fuse}(y^{(k)})] \\ &+\mathrm{prev}(x^{(k-1)}) + \mathrm{deg}(x^{(k-1)}) + \mathrm{radius}(x^{k-1}) + \mathrm{fuse}(y^{(k)}) \end{aligned}$$

Two equations are chained-up in the following order:

$$\begin{aligned} x^{(k+1)} &= f(x^{(k)}, y^{(k)}) \\ y^{(k+1)} &= f(y^{(k)}, x^{(k+1)}) \end{aligned}$$

Implementing prev and deg as tensor operation



Linear projection and degree operation are both simply matrix multiplication. Write them as PyTorch tensor operations.

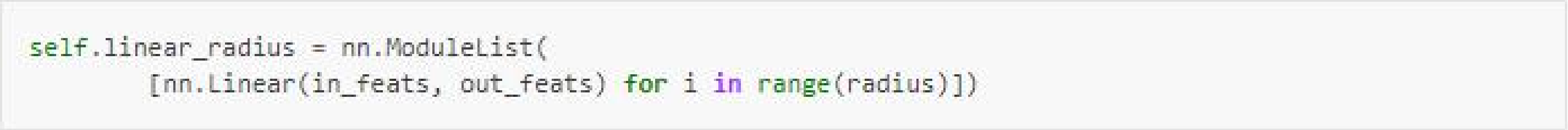
In `__init__`, you define the projection variables.

```
self.linear_prev = nn.Linear(in_feats, out_feats)
self.linear_deg = nn.Linear(in_feats, out_feats)
```

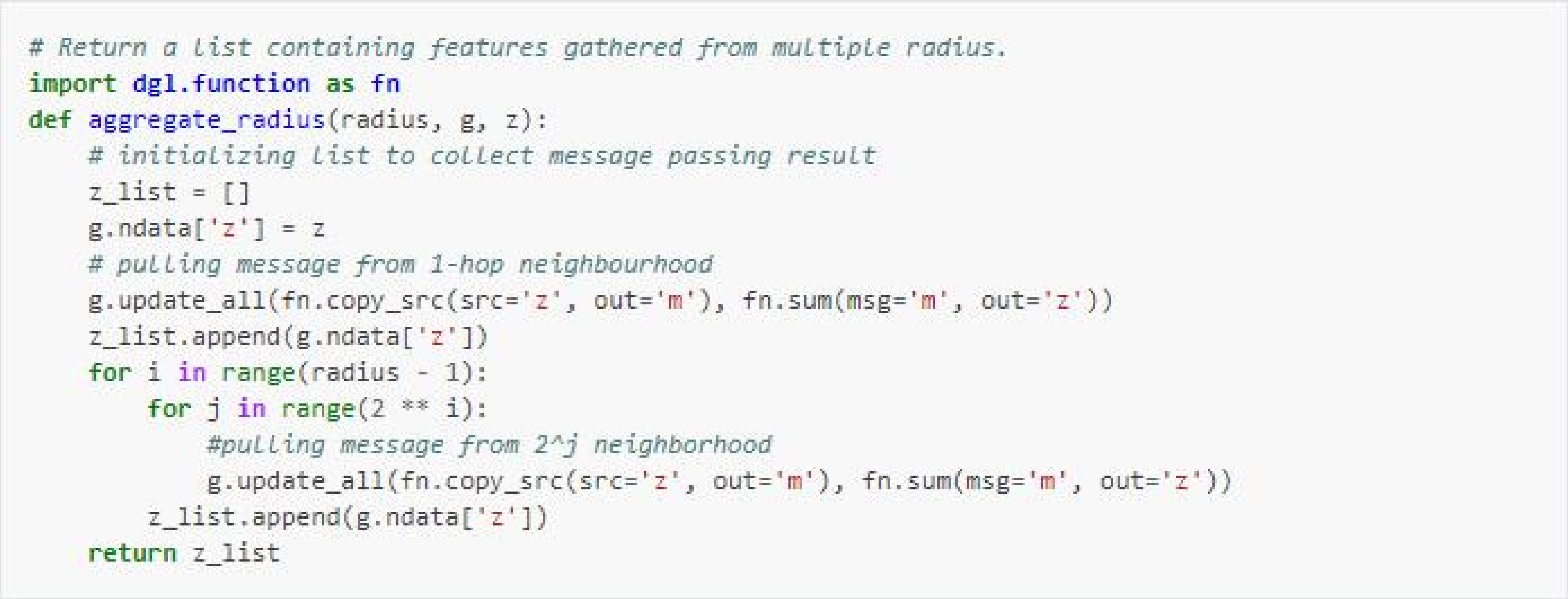
In `forward()`, prev and deg are the same as any other PyTorch tensor operations.

```
prev_proj = self.linear_prev(feat_a)
deg_proj = self.linear_deg(deg * feat_a)
```

Implementing radius as message passing in DGL

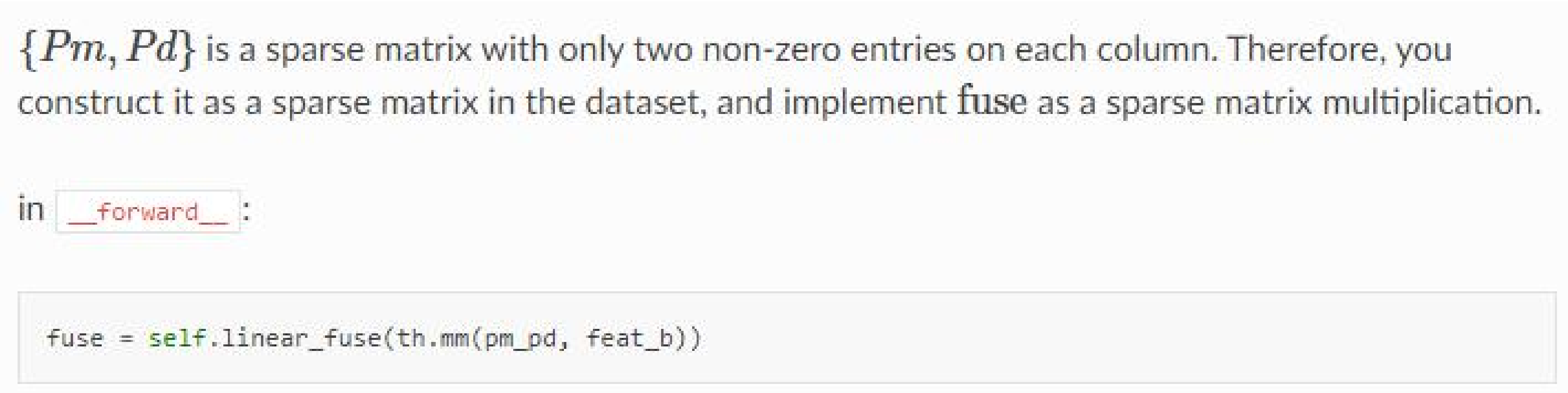


```
self.linear_radius = nn.ModuleList(
        [nn.Linear(in_feats, out_feats) for i in range(radius)])
```



```
# Return a list containing features gathered from multiple radius.
import dgl.function as fn
def aggregate_radius(radius, g, z):
    # initializing list to collect message passing result
    z_list = []
    g.ndata['z'] = z
    # pulling message from 1-hop neighbourhood
    g.update_all(fn.copy_src(src='z', out='m'), fn.sum(msg='m', out='z'))
    z_list.append(g.ndata['z'])
    for i in range(radius - 1):
        for j in range(2 ** i):
            #pulling message from 2^j neighborhood
            g.update_all(fn.copy_src(src='z', out='m'), fn.sum(msg='m', out='z'))
        z_list.append(g.ndata['z'])
    return z_list
```

Implementing fuse as sparse matrix multiplication

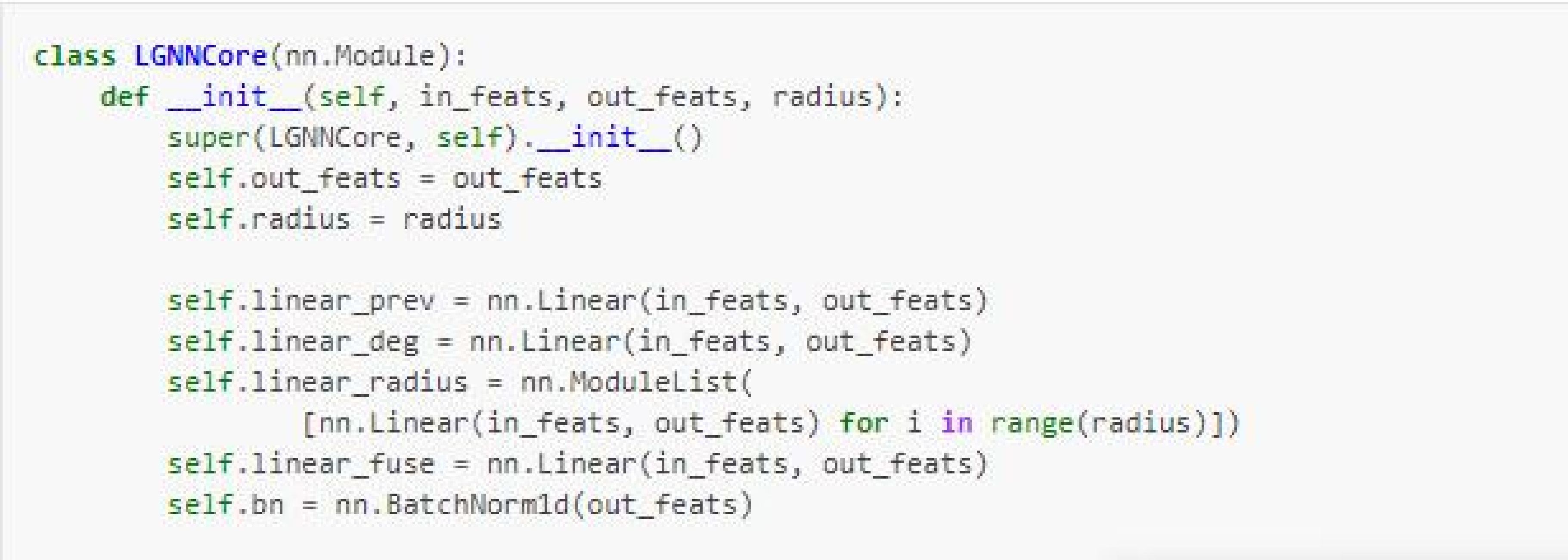


$\{Pm, Pd\}$ is a sparse matrix with only two non-zero entries on each column. Therefore, you construct it as a sparse matrix in the dataset, and implement fuse as a sparse matrix multiplication.

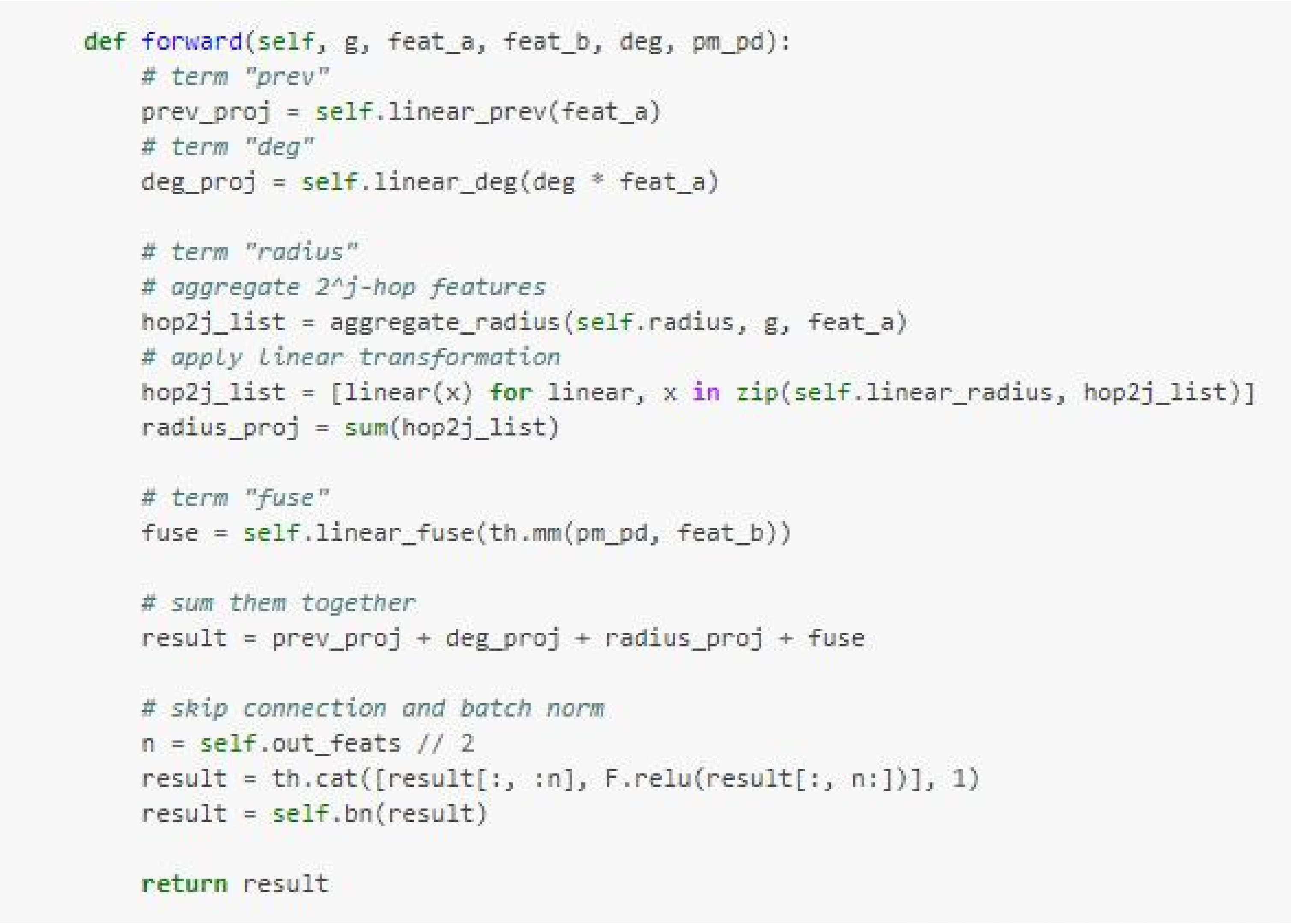
in `__forward__`:

```
fuse = self.linear_fuse(th.mm(pm_pd, feat_b))
```

the complete code for f(x,y)

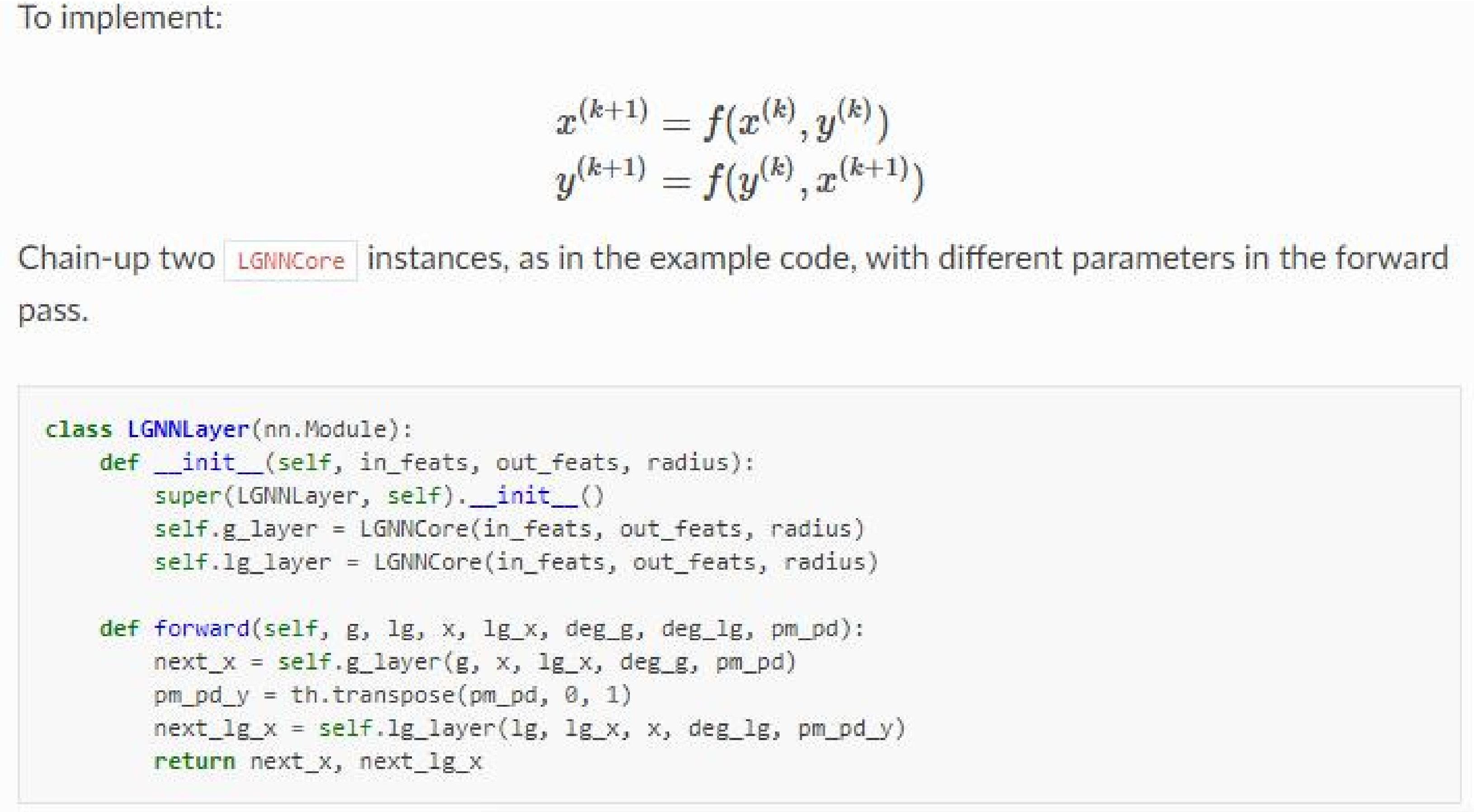


```
class LGNNCore(nn.Module):
    def __init__(self, in_feats, out_feats, radius):
        super(LGNNCore, self).__init__()
        self.out_feats = out_feats
        self.radius = radius

        self.linear_prev = nn.Linear(in_feats, out_feats)
        self.linear_deg = nn.Linear(in_feats, out_feats)
        self.linear_radius = nn.ModuleList(
                [nn.Linear(in_feats, out_feats) for i in range(radius)])
        self.linear_fuse = nn.Linear(in_feats, out_feats)
        self.bn = nn.BatchNorm1d(out_feats)
```



```
    def forward(self, g, feat_a, feat_b, deg, pm_pd):
        # term "prev"
        prev_proj = self.linear_prev(feat_a)
        # term "deg"
        deg_proj = self.linear_deg(deg * feat_a)

        # term "radius"
        # aggregate 2^j-hop features
        hop2j_list = aggregate_radius(self.radius, g, feat_a)
        # apply linear transformation
        hop2j_list = [linear(x) for linear, x in zip(self.linear_radius, hop2j_list)]
        radius_proj = sum(hop2j_list)

        # term "fuse"
        fuse = self.linear_fuse(th.mm(pm_pd, feat_b))

        # sum them together
        result = prev_proj + deg_proj + radius_proj + fuse

        # skip connection and batch norm
        n = self.out_feats // 2
        result = th.cat([result[:, :n], F.relu(result[:, n:])], 1)
        result = self.bn(result)

        return result
```

Chain-up LGNN abstractions as an LGNN layer



To implement:

$$x^{(k+1)} = f(x^{(k)}, y^{(k)})$$
$$y^{(k+1)} = f(y^{(k)}, x^{(k+1)})$$

Chain-up two `LGNNCore` instances, as in the example code, with different parameters in the forward pass.

```
class LGNNLayer(nn.Module):
    def __init__(self, in_feats, out_feats, radius):
        super(LGNNLayer, self).__init__()
        self.g_layer = LGNNCore(in_feats, out_feats, radius)
        self.lg_layer = LGNNCore(in_feats, out_feats, radius)

    def forward(self, g, lg, x, lg_x, deg_g, deg_lg, pm_pd):
        next_x = self.g_layer(g, x, lg_x, deg_g, pm_pd)
        pm_pd_y = th.transpose(pm_pd, 0, 1)
        next_lg_x = self.lg_layer(lg, lg_x, x, deg_lg, pm_pd_y)
        return next_x, next_lg_x
```