消息传递图神经网络

一、引言

在开篇中我们介绍了，为节点生成节点表征是图计算任务成功的关键。本节中，

- 首先我们将学习**图神经网络生成节点表征的范式-消息传递（Message Passing）范式。**
- 接着我们将初步分析PyG为我们提供的实现图神经网络的`MessagePassing`基类。
- 然后我们以继承`MessagePassing`基类的`GCNConv`类为例，学习如何通过继承`MessagePassing`基类实现一个简单的图神经网络。
- 再接着我们将对`MessagePassing`基类进行剖析。
- 最后我们将学习在继承`MessagePassing`基类的子类中覆写`message(),aggreate(),message_and_aggreate()`和`update()`方法的规范。

二、 消息传递范式介绍

用$\mathbf{x}_i^{(k-1)} \in \mathbb{R}^F$表示$(k-1)$层中节点$i$的节点属性，$\mathbf{e}_{j,i} \in \mathbb{R}^D$ 表示从节点j到节点i的边的属性，消息传递图神经网络可以描述为

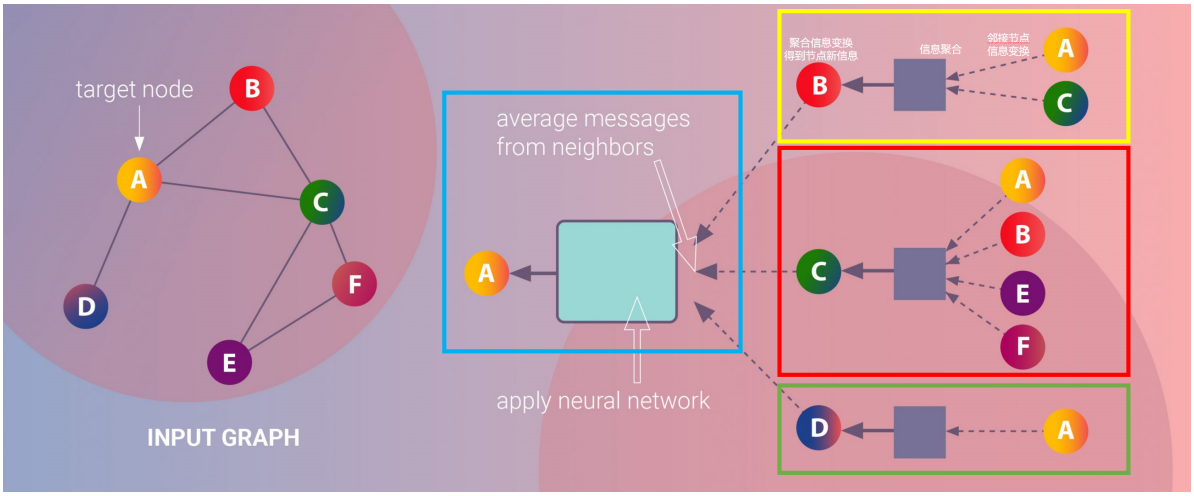
$$\mathbf{x}_i^{(k)} = \gamma^{(k)} \left(\mathbf{x}_i^{(k-1)}, \square_{j \in \mathcal{N}(i)} \, \phi^{(k)} \left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i} \right) \right), \tag{1}$$

其中$\square$表示可微分的、具有排列不变性（函数输出结果与输入参数的排列无关）的函数。具有排列不变性的函数有，和函数、均值函数和最大值函数。γ和ϕ表示可微分的函数，如MLPs（多层感知器）。*此处内容来源于CREATING MESSAGE PASSING NETWORKS。*

神经网络的生成节点表征的操作称为节点嵌入（Node Embedding），节点表征也可以称为节点嵌入。**为了统一此次组队学习中的表述**，我们**规定节点嵌入只代指神经网络生成节点表征的操作。**

下方图片展示了**基于消息传递范式的生成节点表征的过程**：

1. 图中黄色方框部分展示了**邻居节点信息传递到中心节点的过程**：B节点的邻接节点（A,C）的信息经过**变换**后**聚合**到B节点，接着**B节点信息与邻居节点聚合信息一起经过变换**得到B节点的**新的节点信息**。分别如红色和绿色方框部分所示，遵循同样的过程，C、D节点的信息被更新。实际上，同样的过程在所有节点上都进行了一遍，所有节点的信息都更新了一遍。
2. 这样的邻居节点信息传递到中心节点的过程会进行多次。如图中蓝色方框部分所示，A节点的邻接节点（B,C,D）的已经更新过一次的节点信息经过变换、聚合、再变换得到了A节点**第二次更新**的节点信息。**多次更新**后的节点信息就作为节点表征。



消息传递图神经网络是指遵循“消息传递范式”的图神经网络，此类图神经网络实现了上述的节点信息更新过程。

注：未经过训练的图神经网络生成的节点表征还不是好的节点表征，好的节点表征可用于衡量节点之间的相似性。通过监督学习对图神经网络做很好的训练，图神经网络才可以生成好的节点表征。我们将在第5节介绍此部分内容。

三、`MessagePassing`基类初步分析

Pytorch Geometric(PyG)提供了`MessagePassing`基类，它封装了“消息传递”的运行流程。通过继承`MessagePassing`基类，可以方便地构造消息传递图神经网络，构造一个最简单的消息传递图神经网络类，我们只需定义`message()`**方法（ϕ）**、`update()`**方法（γ）**，以及使用的**消息聚合方案**（`aggr="add"`、`aggr="mean"`或`aggr="max"`）。这一切是在以下方法的帮助下完成的：

- `MessagePassing(aggr="add", flow="source_to_target", node_dim=-2)`（对象初始化方法）：
  - `aggr`：定义要使用的聚合方案（"add"、"mean "或 "max"）；
  - `flow`：定义消息传递的流向（"source_to_target "或 "target_to_source"）；
  - `node_dim`：定义沿着哪个轴线传播。
  - 注：`MessagePassing(……)`等同于 `MessagePassing.__init__(……)`
- `MessagePassing.propagate(edge_index, size=None, **kwargs)`：
  - 开始传递消息的起始调用，在此方法中`message`、`update`等方法被调用。
  - 它以`edge_index`（边的端点的索引）和`flow`（消息的流向）以及一些额外的数据为参数。
  - 请注意，`propagate()`不仅限于基于形状为`[N, N]`的对称邻接矩阵进行"消息传递过程"，基于非对称的邻居矩阵进行消息传递（例如图为二部图时），需要传递参数`size=(N, M)`。
  - 如果设置`size=None`，则认为邻接矩阵是对称的。
- `MessagePassing.message(...)`：
  - 首先确定要给节点i传递消息的边的集合：
    - 如果`flow="source_to_target"`，则是$(j,i) \in \mathcal{E}$的边的集合；
    - 如果`flow="target_to_source"`，则是$(i,j) \in \mathcal{E}$的边的集合。
  - 接着为各条边创建要传递给节点i的消息，即实现ϕ函数。
  - `MessagePassing.message(...)`方法可以接收传递给`MessagePassing.propagate(edge_index, size=None, **kwargs)`方法的所有参数，我们在`message()`方法的参数列表里定义要接收的参数，如我们要接收`x,y,z`参数，则我们应定义`message(x,y,z)`方法。
  - 并且传递给`propagate()`方法的参数，如果是节点的属性的话，可以被拆分成属于中心节点的部分和属于邻接节点的部分，只需在变量名后面加上`_i`或`_j`。例如，我们自己定义的`meassage`方法包含参数`x_i`，那么首先`propagate()`方法将节点属性拆分成中心节点

属性和邻接节点属性，接着`propagate()`方法调用`message`方法并传递中心节点属性给参数`x_i`。而如果我们自己定义的`meassage`方法包含参数`x_j`，那么`propagate()`方法会传递邻接节点属性给参数`x_j`。

    - 我们用i表示“消息传递”中的中心节点，用j表示“消息传递”中的邻接节点。

- `MessagePassing.aggregate(...)`：
    - 将从源节点传递过来的消息聚合在目标节点上，一般可选的聚合方式有`sum`，`mean`和`max`。
- `MessagePassing.message_and_aggregate(...)`：
    - 在一些场景里，邻接节点信息变换和邻接节点信息聚合这两项操作可以融合在一起，那么我们可以在此方法里定义这两项操作，从而让程序运行更加高效。
- `MessagePassing.update(aggr_out, ...)`：
    - 为每个节点$i \in \mathcal{V}$更新节点表征，即实现γ函数。此方法以`aggregate`方法的输出为第一个参数，并接收所有传递给`propagate()`方法的参数。

以上内容来源于[The “MessagePassing” Base Class](#)。

四、`MessagePassing`子类实例

我们以继承`MessagePassing`基类的`GCNConv`类为例，学习如何通过继承`MessagePassing`基类实现一个简单的图神经网络。

GCNConv的数学定义为

$$\mathbf{x}_i^{(k)} = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{\deg(i)} \cdot \sqrt{\deg(j)}} \cdot \left(\mathbf{\Theta} \cdot \mathbf{x}_j^{(k-1)} \right), \tag{2}$$

其中，相邻节点的特征首先通过权重矩阵$\mathbf{\Theta}$进行转换，然后按端点的度进行归一化处理，最后进行求和。这个公式可以分为以下几个步骤：

1. 向邻接矩阵添加自环边。
2. 线性转换节点特征矩阵。
3. 计算归一化系数。
4. 归一化j中的节点特征。

5. 将相邻节点特征相加（"求和 "聚合）。

步骤1-3通常是在消息传递发生之前计算的。步骤4-5可以使用`MessagePassing`基类轻松处理。该层的全部实现如下所示。

```
import torch
from torch_geometric.nn import MessagePassing
from torch_geometric.utils import add_self_loops, degree

class GCNConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(GCNConv, self).__init__(aggr='add', flow='source_to_target')
        # "Add" aggregation (Step 5).
        # flow='source_to_target' 表示消息从源节点传播到目标节点
        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

```

GCNConv继承了MessagePassing并以"求和"作为领域节点信息聚合方式。该层的所有逻辑都发生在其forward()方法中。在这里，我们首先使用torch_geometric.utils.add_self_loops()函数向我们的边索引添加自循环边（步骤1），以及通过调用torch.nn.Linear实例对节点特征进行线性变换（步骤2）。propagate()方法也在forward方法中被调用，propagate()方法被调用后节点间的信息传递开始执行。

归一化系数是由每个节点的节点度得出的，它被转换为每条边的节点度。结果被保存在形状[num_edges,]的张量norm中（步骤3）。

在message()方法中，我们需要通过norm对邻接节点属性x_j进行归一化处理。

通过以上内容的学习，我们便掌握了**创建一个仅包含一次"消息传递过程"的图神经网络的方法**。如下方代码所示，我们可以很方便地初始化和调用它：

```
from torch_geometric.datasets import Planetoid

dataset = Planetoid(root='dataset/Cora', name='Cora')
data = dataset[0]

net = GCNConv(data.num_features, 64)
h_nodes = net(data.x, data.edge_index)
print(h_nodes.shape)

```

通过串联多个这样的简单图神经网络，我们就可以构造复杂的图神经网络模型。我们将在第5节介绍复杂图神经网络模型的构建。

以上内容来源于Implementing the GCN Layer。

五、MessagePassing基类剖析

在`__init__()`方法中，我们看到程序会检查子类是否实现了`message_and_aggregate()`方法，并将检查结果赋值给`fuse`属性。

```
class MessagePassing(torch.nn.Module):
    def __init__(self, aggr: Optional[str] = "add", flow: str = "source_to_target", node_dim: int = -2):
        super(MessagePassing, self).__init__()
        # 此处省略n行代码
        # Support for "fused" message passing.
        self.fuse = self.inspector.implements('message_and_aggregate')
        # 此处省略n行代码

```

”消息传递过程“是从`propagate`方法被调用开始执行的。

```
class MessagePassing(torch.nn.Module):
    # 此处省略n行代码
    def propagate(self, edge_index: Adj, size: Size = None, **kwargs):
        # 此处省略n行代码
        # Run "fused" message and aggregation (if applicable).
        if (isinstance(edge_index, SparseTensor) and self.fuse and not self.__explain__):
            coll_dict = self.__collect__(self.__fused_user_args__, edge_index, size, kwargs)

            msg_aggr_kwargs = self.inspector.distribute('message_and_aggregate', coll_dict)
            out = self.message_and_aggregate(edge_index, **msg_aggr_kwargs)

            update_kwargs = self.inspector.distribute('update', coll_dict)
            return self.update(out, **update_kwargs)
```

```
        # Otherwise, run both functions in separation.
        elif isinstance(edge_index, Tensor) or not
self.fuse:
            coll_dict =
self.__collect__(self.__user_args__, edge_index, size,
kwargs)

            msg_kwargs =
self.inspector.distribute('message', coll_dict)
            out = self.message(**msg_kwargs)
            # 此处省略n行代码
            aggr_kwargs =
self.inspector.distribute('aggregate', coll_dict)
            out = self.aggregate(out, **aggr_kwargs)

            update_kwargs =
self.inspector.distribute('update', coll_dict)
            return self.update(out, **update_kwargs)

```

参数简介：

- `edge_index`：边端点索引，它可以是`Tensor`类型或`SparseTensor`类型。
  - 当flow="source_to_target"时，节点`edge_index[0]`的信息将被传递到节点`edge_index[1]`，
  - 当flow="target_to_source"时，节点`edge_index[1]`的信息将被传递到节点`edge_index[0]`
- `size`：邻接节点的数量与中心节点的数量。
  - 对于普通图，邻接节点的数量与中心节点的数量都是N，我们可以不给size传参数，即让size取值为默认值None。
  - 对于二部图，邻接节点的数量与中心节点的数量分别记为M, N，于是我们需要给size参数传一个元组`(M, N)`。
- `kwargs`：图其他属性或额外的数据。

`propagate()`方法首先检查`edge_index`是否为`SparseTensor`类型以及是否子类实现了`message_and_aggregate()`方法，如是就执行子类的`message_and_aggregate`方法；否则依次执行子类的`message(),aggregate(),update()`三个方法。

六、message方法的覆写

前面我们介绍了，传递给propagate()方法的参数，如果是节点的属性的话，可以被拆分成属于中心节点的部分和属于邻接节点的部分，只需在变量名后面加上_i或_j。现在我们有一个额外的节点属性，节点的度，deg，我们希望meassge方法还能接收中心节点的度，我们对前面GCNConv的message方法进行改造得到新的GCNConv类：

```
import torch
from torch_geometric.nn import MessagePassing
from torch_geometric.utils import add_self_loops, degree

class GCNConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(GCNConv, self).__init__(aggr='add', flow='source_to_target')
        # "Add" aggregation (Step 5).
        # flow='source_to_target' 表示消息从源节点传播到目标节点
        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, deg=deg.view((-1, 1)))

    def message(self, x_j, norm, deg_i):
        # x_j has shape [E, out_channels]
        # deg_i has shape [E, 1]
        # Step 4: Normalize node features.
        return norm.view(-1, 1) * x_j * deg_i


from torch_geometric.datasets import Planetoid

dataset = Planetoid(root='dataset/Cora', name='Cora')
data = dataset[0]

net = GCNConv(data.num_features, 64)
h_nodes = net(data.x, data.edge_index)
print(h_nodes.shape)

```

若一个数据可以被拆分成属于中心节点的部分和属于邻接节点的部分，其形状必须是`[num_nodes, num_features]`，因此在上方代码的第`29`行，我们执行了`deg.view((-1, 1))`操作，使得数据形状为`[num_nodes, 1]`，然后才将数据传给`propagate()`方法。

七、`aggregate`方法的覆写

在前面的例子的基础上，我们增加如下的`aggregate`方法。通过观察运行结果我们可以看到，我们覆写的`aggregate`方法被调用，同时在`super(GCNConv, self).__init__(aggr='add')`中传递给`aggr`参数的值被存储到了`self.aggr`属性中。

```
import torch
from torch_geometric.nn import MessagePassing
from torch_geometric.utils import add_self_loops, degree

```

```python
class GCNConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(GCNConv, self).__init__(aggr='add', flow='source_to_target')
        # "Add" aggregation (Step 5).
        # flow='source_to_target' 表示消息从源节点传播到目标节点
        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, deg=deg.view((-1, 1)))

    def message(self, x_j, norm, deg_i):
        # x_j has shape [E, out_channels]
        # deg_i has shape [E, 1]
        # Step 4: Normalize node features.
        return norm.view(-1, 1) * x_j * deg_i

    def aggregate(self, inputs, index, ptr, dim_size):
        print('self.aggr:', self.aggr)
```

```
        print("`aggregate` is called")
        return super().aggregate(inputs, index,
ptr=ptr, dim_size=dim_size)


from torch_geometric.datasets import Planetoid

dataset = Planetoid(root='dataset/Cora', name='Cora')
data = dataset[0]

net = GCNConv(data.num_features, 64)
h_nodes = net(data.x, data.edge_index)
print(h_nodes.shape)

```

八、message_and_aggregate方法的覆写

在一些案例中，“消息传递”与“消息聚合”可以融合在一起。对于这种情况，我们可以覆写message_and_aggregate方法，在message_and_aggregate方法中一块实现“消息传递”与“消息聚合”，这样能使程序的运行更加高效。

```
import torch
from torch_geometric.nn import MessagePassing
from torch_geometric.utils import add_self_loops,
degree
from torch_sparse import SparseTensor

class GCNConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(GCNConv, self).__init__(aggr='add',
flow='source_to_target')
        # "Add" aggregation (Step 5).
        # flow='source_to_target' 表示消息从源节点传播到目
标节点
        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.
        adjmat = SparseTensor(row=edge_index[0], col=edge_index[1],
value=torch.ones(edge_index.shape[1]))
        # 此处传的不再是edge_idex，而是SparseTensor类型的Adjancency Matrix
        return self.propagate(adjmat, x=x, norm=norm, deg=deg.view((-1, 1)))

    def message(self, x_j, norm, deg_i):
        # x_j has shape [E, out_channels]
        # deg_i has shape [E, 1]
        # Step 4: Normalize node features.
        return norm.view(-1, 1) * x_j * deg_i

    def aggregate(self, inputs, index, ptr, dim_size):
        print('self.aggr:', self.aggr)
        print("`aggregate` is called")
        return super().aggregate(inputs, index, ptr=ptr, dim_size=dim_size)

    def message_and_aggregate(self, adj_t, x, norm):
        print('`message_and_aggregate` is called')
```

```
        # 没有实现真实的消息传递与消息聚合的操作

from torch_geometric.datasets import Planetoid

dataset = Planetoid(root='dataset/Cora', name='Cora')
data = dataset[0]

net = GCNConv(data.num_features, 64)
h_nodes = net(data.x, data.edge_index)
# print(h_nodes.shape)

```

运行程序后我们可以看到，虽然我们同时覆写了`message`方法和`aggregate`方法，然而只有`message_and_aggregate`方法被执行。

九、覆写`update`方法

```
from torch_geometric.datasets import Planetoid
import torch
from torch_geometric.nn import MessagePassing
from torch_geometric.utils import add_self_loops, degree
from torch_sparse import SparseTensor


class GCNConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(GCNConv, self).__init__(aggr='add', flow='source_to_target')
        # "Add" aggregation (Step 5).
        # flow='source_to_target' 表示消息从源节点传播到目标节点
        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.
        adjmat = SparseTensor(row=edge_index[0], col=edge_index[1],
value=torch.ones(edge_index.shape[1]))
        # 此处传的不再是edge_idex，而是SparseTensor类型的Adjancency Matrix
        return self.propagate(adjmat, x=x, norm=norm, deg=deg.view((-1, 1)))

    def message(self, x_j, norm, deg_i):
        # x_j has shape [E, out_channels]
        # deg_i has shape [E, 1]
        # Step 4: Normalize node features.
        return norm.view(-1, 1) * x_j * deg_i

    def aggregate(self, inputs, index, ptr, dim_size):
        print('self.aggr:', self.aggr)
        print("`aggregate` is called")
        return super().aggregate(inputs, index, ptr=ptr, dim_size=dim_size)

    def message_and_aggregate(self, adj_t, x, norm):
        print('`message_and_aggregate` is called')
        # 没有实现真实的消息传递与消息聚合的操作

    def update(self, inputs, deg):
        print(deg)
        return inputs
```

```
54
55
56 dataset = Planetoid(root='dataset/Cora', name='Cora')
57 data = dataset[0]
58
59 net = GCNConv(data.num_features, 64)
60 h_nodes = net(data.x, data.edge_index)
61 # print(h_nodes.shape)
62
```

`update`方法接收聚合的输出作为第一个参数，此外还可以接收传递给`propagate`方法的任何参数。在上方的代码中，我们覆写的`update`方法接收了聚合的输出作为第一个参数，此外接收了传递给`propagate`的`deg`参数。

十、结语

消息传递范式是一种聚合邻接节点信息来更新中心节点信息的范式，它将卷积算子推广到了不规则数据领域，实现了图与神经网络的连接。**该范式包含这样三个步骤：(1)邻接节点信息变换、(2)邻接节点信息聚合到中心节点、(3)聚合信息变换。**因为简单且强大的特性，消息传递范式现被人们广泛地使用。基于此范式，我们可以定义聚合邻接节点信息来生成中心节点表征的图神经网络。在PyG中，`MessagePassing`基类是所有基于消息传递范式的图神经网络的基类，它大大地方便了我们对图神经网络的构建。

此节内容为我们打开了`MessagePassing`基类的黑箱子，通过此节内容的学习，我们可以了解`MessagePassing`基类实现"消息传递"的运行流程，可以掌握继承`MessagePassing`基类来构造自己的图神经网络类的规范。

再次强调，要掌握基于`MessagePassing`基类构建自己的图神经网络类的方法，我们不能仅停留于理论理解层面，还需要通过逐行代码调试，观察代码运行流程，最终掌握对`MessagePassing`基类的实际应用。

作业

1. 请总结`MessagePassing`基类的运行流程以及通过继承`MessagePassing`基类来构造自己的图神经网络类的规范。

参考资料