Creating message passing networks

Generalizing the convolution operator to irregular domains is typically expressed as a *neighborhood aggregation* or *message passing* scheme. With $\mathbf{x}_i^{(k-1)} \in \mathbb{R}^F$ denoting node features of node i in layer $(k-1)$ and $\mathbf{e}_{i,j} \in \mathbb{R}^D$ denoting (optional) edge features from node i to node j, message passing graph neural networks can be described as

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

where $\square$ denotes a differentiable, permutation invariant function, *e.g.*, sum, mean or max, and γ and ϕ denote differentiable functions such as MLPs.

- The "MessagePassing" base class
- Implementing the GCN layer
- Implementing the edge convolution

The "MessagePassing" base class

PyTorch Geometric provides the `torch_geometric.nn.MessagePassing` base class, which helps in creating such kinds of message passing graph neural networks by automatically taking care of message propagation. The user only has to define the functions ϕ , *i.e.* `message()`, and γ , *.i.e.* `update()`, as well as the aggregation scheme to use, *.i.e.* `aggr='add'`, `aggr='mean'` or `aggr='max'`.

This is done with the help of the following methods:

- `torch_geometric.nn.MessagePassing(aggr="add", flow="source_to_target")`: Defines the aggregation scheme to use (`"add"`, `"mean"` or `"max"`) and the flow direction of message passing (either `"source_to_target"` or `"target_to_source"`).
- `torch_geometric.nn.MessagePassing.propagate(edge_index, size=None, **kwargs)`: The initial call to start propagating messages. Takes in the edge indices and all additional data which is needed to construct messages and to update node embeddings. Note that `propagate()` is not limited to exchange messages in symmetric adjacency matrices of shape `[N, N]` only, but can also exchange messages in general sparse assignment matrices, *.e.g.*, bipartite graphs, of shape `[N, M]` by passing `size=(N, M)` as an additional argument. If set to `None`, the assignment matrix is assumed to be symmetric. For bipartite graphs with two independent sets of nodes and indices, and each set holding its own information, this split can be marked by passing the information as a tuple, *e.g.* `x=(x_row, x_col)` and to indicate the memberships to the different sets.

- `torch_geometric.nn.MessagePassing.message()`: Constructs messages to node i in analogy to ϕ for each edge in $(j,i) \in \mathcal{E}$ if `flow="source_to_target"` and $(i,j) \in \mathcal{E}$ if `flow="target_to_source"`. Can take any argument which was initially passed to `propagate()`. In addition, features can be mapped to the respective nodes i and j by appending `_i` or `_j` to the variable name, *.e.g.* `x_i` and `x_j`.
- `torch_geometric.nn.MessagePassing.update()`: Updates node embeddings in analogy to γ for each node $i \in \mathcal{V}$. Takes in the output of aggregation as first argument and any argument which was initially passed to `propagate()`.

Let us verify this by re-implementing two popular GNN variants, the GCN layer from Kipf and Welling and the EdgeConv layer from Wang et al..

Implementing the GCN layer

The GCN layer is mathematically defined as

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

where neighboring node features are first transformed by a weight matrix $\mathbf{\Theta}$, normalized by their degree, and finally summed up. This formula can be divided into the following steps:

1. Add self-loops to the adjacency matrix.
2. Linearly transform node feature matrix.
3. Normalize node features in ϕ.
4. Sum up neighboring node features (`"add"` aggregation).
5. Return new node embeddings in γ.

Steps 1-2 are typically computed before message passing takes place. Steps 3-5 can be easily processed using the `torch_geometric.nn.MessagePassing` base class. The full layer implementation is shown below:

```python
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')  # "Add" aggregation.
        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 3-5: Start propagating messages.
        return self.propagate(edge_index, size=(x.size(0), x.size(0)), x=x)

    def message(self, x_j, edge_index, size):
        # x_j has shape [E, out_channels]

        # Step 3: Normalize node features.
        row, col = edge_index
        deg = degree(row, size[0], dtype=x_j.dtype)
        deg_inv_sqrt = deg.pow(-0.5)
        norm = deg_inv_sqrt[row] * deg_inv_sqrt[col]

        return norm.view(-1, 1) * x_j

    def update(self, aggr_out):
        # aggr_out has shape [N, out_channels]

        # Step 5: Return new node embeddings.
        return aggr_out
```

`GCNConv` inherits from `torch_geometric.nn.MessagePassing` with `"add"` propagation. All the logic of the layer takes place in `forward()`. Here, we first add self-loops to our edge indices using the `torch_geometric.utils.add_self_loops()` function (step 1), as well as linearly transform node features by calling the `torch.nn.Linear` instance (step 2).

We then proceed to call `propagate()`, which internally calls the `message()` and `update()` functions. As additional arguments for message propagation, we pass the node embeddings `x`.

In the `message()` function, we need to normalize the neighboring node features `x_j`. Here, `x_j` denotes a *mapped* tensor, which contains the neighboring node features of each edge. Node features can be automatically mapped by appending `_i` or `_j` to the variable name. In fact, any tensor can be mapped this way, as long as they have N entries in its first dimension.

The neighboring node features are normalized by computing node degrees $\deg(i)$ for each node i and saving $1/(\sqrt{\deg(i)} \cdot \sqrt{\deg(j)})$ in `norm` for each edge $(i,j) \in \mathcal{E}$.

In the `update()` function, we simply return the output of the aggregation.

That is all that it takes to create a simple message passing layer. You can use this layer as a building block for deep architectures. Initializing and calling it is straightforward:

```
conv = GCNConv(16, 32)
x = conv(x, edge_index)
```

Implementing the edge convolution

The edge convolutional layer processes graphs or point clouds and is mathematically defined as

$$\mathbf{x}_i^{(k)} = \max_{j \in \mathcal{N}(i)} h_{\mathbf{\Theta}} \left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)} - \mathbf{x}_i^{(k-1)} \right),$$

where $h_{\mathbf{\Theta}}$ denotes a MLP. Analogous to the GCN layer, we can use the `torch_geometric.nn.MessagePassing` class to implement this layer, this time using the `"max"` aggregation:

```
import torch
from torch.nn import Sequential as Seq, Linear, ReLU
from torch_geometric.nn import MessagePassing

class EdgeConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(EdgeConv, self).__init__(aggr='max') #  "Max" aggregation.
        self.mlp = Seq(Linear(2 * in_channels, out_channels),
                       ReLU(),
                       Linear(out_channels, out_channels))

    def forward(self, x, edge_index):
        # x has shape [N, in_channels]
        # edge_index has shape [2, E]

        return self.propagate(edge_index, size=(x.size(0), x.size(0)), x=x)

    def message(self, x_i, x_j):
        # x_i has shape [E, in_channels]
        # x_j has shape [E, in_channels]

        tmp = torch.cat([x_i, x_j - x_i], dim=1)  # tmp has shape [E, 2 * in_channels]
        return self.mlp(tmp)

    def update(self, aggr_out):
        # aggr_out has shape [N, out_channels]

        return aggr_out
```

Inside the `message()` function, we use `self.mlp` to transform both the source node features `x_i` and the relative target node features `x_j - x_i` for each edge.

The edge convolution is actual a dynamic convolution, which recomputes the graph for each layer using nearest neighbors in the feature space. Luckily, PyTorch Geometric comes with a GPU accelerated batch-wise k-NN graph generation method named `torch_geometric.nn.knn_graph()`:

```
from torch_geometric.nn import knn_graph

class DynamicEdgeConv(EdgeConv):
    def __init__(self, in_channels, out_channels, k=6):
        super(DynamicEdgeConv, self).__init__(in_channels, out_channels)
        self.k = k

    def forward(self, x, batch=None):
        edge_index = knn_graph(x, self.k, batch, loop=False)
        return super(DynamicEdgeConv, self).forward(x, edge_index)
```

Here, `knn_graph()` computes a nearest neighbor graph, which is further used to call the `forward()` method of `EdgeConv`.

This leaves us with a clean interface for initializing and calling this layer:

```
conv = DynamicEdgeConv(3, 128, k=6)
x = conv(pos, batch)
```