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

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

where $\bigoplus$ denotes a differentiable, permutation invariant function, *e.g.*, sum, mean or max, and γ and ϕ denote differentiable functions such as MLPs (Multi Layer Perceptrons).

The "MessagePassing" Base Class

PyG provides the `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:

- `MessagePassing(aggr="add", flow="source_to_target", node_dim=-2)` : 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"`). Furthermore, the `node_dim` attribute indicates along which axis to propagate.
- `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 exchanging messages in square adjacency matrices of shape `[N, N]` only, but can also exchange messages in general sparse assignment matrices, *e.g.*, b latest of shape `[N, M]` by passing `size=(N, M)` as an additional argument. If set to `None` , the

assignment matrix is assumed to be a square matrix. 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_N, x_M)` .

- `MessagePassing.message(...)` : Constructs messages to node i in analogy to ϕ for each edge $(j, i) \in \mathcal{E}$ 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, tensors passed to `propagate()` 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` . Note that we generally refer to i as the central nodes that aggregates information, and refer to j as the neighboring nodes, since this is the most common notation.
- `MessagePassing.update(aggr_out, ...)` : 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

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

where neighboring node features are first transformed by a weight matrix $\mathbf{W}$, normalized by their degree, and finally summed up. Lastly, we apply the bias vector $\mathbf{b}$ to the aggregated output. This formula can be divided into the following steps:

1. Add self-loops to the adjacency matrix.
2. Linearly transform node feature matrix.
3. Compute normalization coefficients.
4. Normalize node features in ϕ.
5. Sum up neighboring node features (`"add"` aggregation).
6. Apply a final bias vector.

Steps 1-3 are typically computed before message passing takes place. Steps 4-5 can be easily processed using the `MessagePassing` base class. The full layer implementation is shown below:

latest

```python
import torch
from torch.nn import Linear, Parameter
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().__init__(aggr='add')  # "Add" aggregation (Step 5).
        self.lin = Linear(in_channels, out_channels, bias=False)
        self.bias = Parameter(torch.empty(out_channels))

        self.reset_parameters()

    def reset_parameters(self):
        self.lin.reset_parameters()
        self.bias.data.zero_()

    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)
        deg_inv_sqrt[deg_inv_sqrt == float('inf')] = 0
        norm = deg_inv_sqrt[row] * deg_inv_sqrt[col]

        # Step 4-5: Start propagating messages.
        out = self.propagate(edge_index, x=x, norm=norm)

        # Step 6: Apply a final bias vector.
        out = out + self.bias

        return out

    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` inherits from `MessagePassing` with `"add"` propagation. All the logic of the layer takes place in its `forward()` method. 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).

latest

The normalization coefficients are derived by the node degrees $\deg(i)$ for each node i which gets transformed to $1/(\sqrt{\deg(i)} \cdot \sqrt{\deg(j)})$ for each edge $(j,i) \in \mathcal{E}$. The result is saved in the tensor `norm` of shape `[num_edges, ]` (step 3).

We then call `propagate()`, which internally calls `message()`, `aggregate()` and `update()`. We pass the node embeddings `x` and the normalization coefficients `norm` as additional arguments for message propagation.

In the `message()` function, we need to normalize the neighboring node features `x_j` by `norm`. Here, `x_j` denotes a *lifted* tensor, which contains the source node features of each edge, *i.e.*, the neighbors of each node. Node features can be automatically lifted by appending `_i` or `_j` to the variable name. In fact, any tensor can be converted this way, as long as they hold source or destination node features.

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 an MLP. In analogy to the GCN layer, we can use the `MessagePassing` class to implement this layer, this time using the `"max"` aggregation:

latest

```
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().__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, 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)
```

Inside the **`message()`** function, we use **`self.mlp`** to transform both the target node features **`x_i`** and the relative source node features **`x_j - x_i`** for each edge $(j, i) \in \mathcal{E}$.

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

```
from torch_geometric.nn import knn_graph

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

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

Exercises

Imagine we are given the following `Data` object:

```
import torch
from torch_geometric.data import Data

edge_index = torch.tensor([[0, 1],
                           [1, 0],
                           [1, 2],
                           [2, 1]], dtype=torch.long)
x = torch.tensor([[-1], [0], [1]], dtype=torch.float)

data = Data(x=x, edge_index=edge_index.t().contiguous())
```

Try to answer the following questions related to `GCNConv` :

1. What information does `row` and `col` hold?
2. What does `degree()` do?
3. Why do we use `degree(col, ...)` rather than `degree(row, ...)` ?
4. What does `deg_inv_sqrt[col]` and `deg_inv_sqrt[row]` do?
5. What information does `x_j` hold in the `message()` function? If `self.lin` denotes the identity function, what is the exact content of `x_j` ?
6. Add an `update()` function to `GCNConv` that adds transformed central node features to the aggregated output.

Try to answer the following questions related to `EdgeConv` :

1. What is `x_i` and `x_j - x_i` ?
2. What does `torch.cat([x_i, x_j - x_i], dim=1)` do? Why `dim = 1` ?