Memory-efficient aggregations Tutorial 6

Gabriele Santin

https://gabrielesantin.github.io/

TABLE OF CONTENTS

01 Sparse matrices and SparseTensor

02 Use of SparseTensor in MessagePassing

01 Sparse matrices and SparseTensor

… notebook …

File Edit View Insert Cell Kernel Widgets Help Trusted

Run Code

Tutorial6: Memory-efficient aggregations

Outline

- Representation of sparse matrices
- The `SparseTensor` class
- Use of `SparseTensor` in PyG

Official resources:

- PyG documentation

02 Use of SparseTensor in MessagePassing

Message passing: an example layer

$$\mathbf{x}_i' = \sum_{j \in \mathcal{N}(i)} \mathrm{MLP}(\mathbf{x}_j - \mathbf{x}_i),$$

02 Use of SparseTensor in MessagePassing

Message passing: an example layer

$$\mathbf{x}'_i = \sum_{j \in \mathcal{N}(i)} \mathrm{MLP}(\mathbf{x}_j - \mathbf{x}_i),$$

```python
from torch_geometric.nn import MessagePassing

x = ...            # Node features of shape [num_nodes, num_features]
edge_index = ...   # Edge indices of shape [2, num_edges]

class MyConv(MessagePassing):
    def __init__(self):
        super().__init__(aggr="add")

    def forward(self, x, edge_index):
        return self.propagate(edge_index, x=x)

    def message(self, x_i, x_j):
        return MLP(x_j - x_i)
```

02 Use of SparseTensor in MessagePassing

Message passing: an example layer

$$\mathbf{x}_i' = \sum_{j \in \mathcal{N}(i)} \mathrm{MLP}(\mathbf{x}_j - \mathbf{x}_i),$$

```python
from torch_geometric.nn import MessagePassing

x = ...            # Node features of shape [num_nodes, num_features]
edge_index = ...   # Edge indices of shape [2, num_edges]

class MyConv(MessagePassing):
    def __init__(self):
        super().__init__(aggr="add")

    def forward(self, x, edge_index):
        return self.propagate(edge_index, x=x)

    def message(self, x_i, x_j):
        return MLP(x_j - x_i)
```

02 Use of SparseTensor in MessagePassing

```python
from torch_geometric.nn import MessagePassing

x = ...           # Node features of shape [num_nodes, num_features]
edge_index = ...  # Edge indices of shape [2, num_edges]

class MyConv(MessagePassing):
    def __init__(self):
        super().__init__(aggr="add")

    def forward(self, x, edge_index):
        return self.propagate(edge_index, x=x)

    def message(self, x_i, x_j):
        return MLP(x_j - x_i)
```

02 Use of SparseTensor in MessagePassing

```python
from torch_geometric.nn import MessagePassing

x = ...              # Node features of shape [num_nodes, num_features]
edge_index = ...     # Edge indices of shape [2, num_edges]

class MyConv(Mes
    def __init__
        super().

    def forward(
        return s

    def message(
        return M
```

```python
from torch_scatter import scatter

x = ...              # Node features of shape [num_nodes, num_features]
edge_index = ...     # Edge indices of shape [2, num_edges]

x_j = x[edge_index[0]]  # Source node features [num_edges, num_features]
x_i = x[edge_index[1]]  # Target node features [num_edges, num_features]

msg = MLP(x_j - x_i)  # Compute message for each edge

# Aggregate messages based on target node indices
out = scatter(msg, edge_index[1], dim=0, dim_size=x.size(0), reduce="add")
```

02 Use of SparseTensor in MessagePassing

```python
from torch_geometric.nn import MessagePassing

x = ...              # Node features of shape [num_nodes, num_features]
edge_index = ...     # Edge indices of shape [2, num_edges]

class MyConv(Mes
    def __init__
        super().

    def forward(
        return s

    def message(
        return M
```

```python
from torch_scatter import scatter

x = ...              # Node features of shape [num_nodes, num_features]
edge_index = ...     # Edge indices of shape [2, num_edges]

x_j = x[edge_index[0]]  # Source node features [num_edges, num_features]
x_i = x[edge_index[1]]  # Target node features [num_edges, num_features]

msg = MLP(x_j - x_i)  # Compute message for each edge

# Aggregate messages based on target node indices
out = scatter(msg, edge_index[1], dim=0, dim_size=x.size(0), reduce="add")
```

gather

02 Use of SparseTensor in MessagePassing

```python
from torch_geometric.nn import MessagePassing

x = ...              # Node features of shape [num_nodes, num_features]
edge_index = ...     # Edge indices of shape [2, num_edges]

class MyConv(Mes
    def __init__
        super().

    def forward(
        return s

    def message(
        return M
```

```python
from torch_scatter import scatter

x = ...              # Node features of shape [num_nodes, num_features]
edge_index = ...     # Edge indices of shape [2, num_edges]

x_j = x[edge_index[0]]  # Source node features [num_edges, num_features]
x_i = x[edge_index[1]]  # Target node features [num_edges, num_features]

msg = MLP(x_j - x_i)  # Compute message for each edge

# Aggregate messages based on target node indices
out = scatter(msg, edge_index[1], dim=0, dim_size=x.size(0), reduce="add")
```

gather

scatter

02 Use of SparseTensor in MessagePassing

gather

- Collect all features and apply the message
- It requires the explicit storage of `x_i`, `x_j`
  (lots of possible duplicates, especially on dense graphs)

```
x_j = x[edge_index[0]]
x_i = x[edge_index[1]]
```

02 Use of SparseTensor in MessagePassing

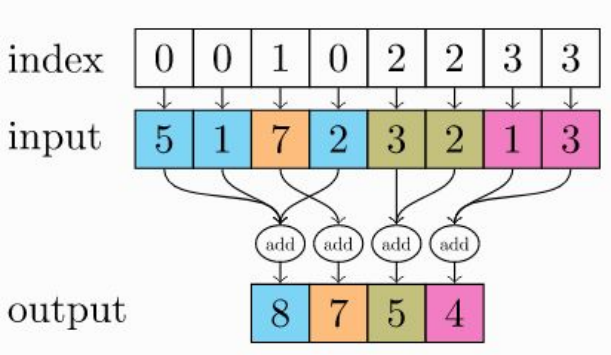
gather

- Collect all features and apply the message
- It requires the explicit storage of `x_i`, `x_j` (lots of possible duplicates, especially on dense graphs)

```
x_j = x[edge_index[0]]
x_i = x[edge_index[1]]
```

scatter

- Aggregate the message according to the target node index
- It requires a lot of aggregation operations



02 Use of SparseTensor in MessagePassing

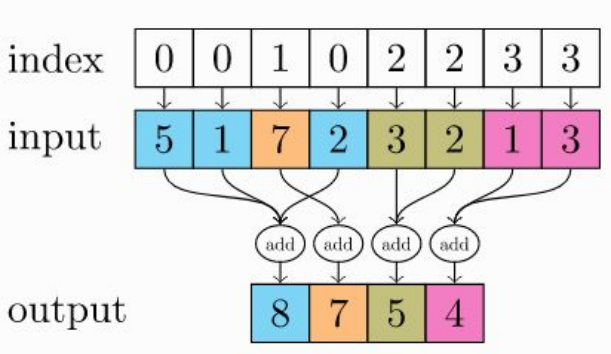
gather

- Collect all features and apply the message
- It requires the explicit storage of `x_i`, `x_j` (lots of possible duplicates, especially on dense graphs)

```
x_j = x[edge_index[0]]
x_i = x[edge_index[1]]
```

scatter

- Aggregate the message according to the target node index
- It requires a lot of aggregation operations



Sparse matrix multiplication as an efficient replacement of gather-scatter

02 Use of SparseTensor in MessagePassing

GINConv example:

$$\mathbf{x}'_i = \mathrm{MLP}\left((1+\epsilon)\cdot\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)}\mathbf{x}_j\right)$$

02 Use of SparseTensor in MessagePassing

GINConv example:

$$\mathbf{x}'_i = \mathrm{MLP}\left((1+\epsilon)\cdot\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)}\mathbf{x}_j\right) \rightarrow \mathbf{X}' = \mathrm{MLP}\left((1+\epsilon)\cdot\mathbf{X} + \mathbf{A}\mathbf{X}\right)$$

02 Use of SparseTensor in MessagePassing

GINConv example:

$$\mathbf{x}'_i = \mathrm{MLP}\left((1+\epsilon)\cdot\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)}\mathbf{x}_j\right) \rightarrow \mathbf{X}' = \mathrm{MLP}\left((1+\epsilon)\cdot\mathbf{X} + \mathbf{A}\mathbf{X}\right)$$

- **Lower memory** footprint
- **Faster execution** time
- Possible for most, but not all layers

02 Use of SparseTensor in MessagePassing

The method `message_and_aggregate()`:

- It combines the `message` and `aggregate` methods in one single call that uses sparse matrix operations.
- It is automatically called whenever implemented!

02 Use of SparseTensor in MessagePassing

The method `message_and_aggregate()`:

- It combines the `message` and `aggregate` methods in one single call that uses sparse matrix operations.
- It is automatically called whenever implemented!

02 Use of SparseTensor in MessagePassing

The method `message_and_aggregate()` :

- It combines the `message` and `aggregate` methods in one single call that uses sparse matrix operations.
- It is automatically called whenever implemented!

```python
from torch_sparse import matmul

class GINConv(MessagePassing):
    def __init__(self):
        super().__init__(aggr="add")

    def forward(self, x, edge_index):
        out = self.propagate(edge_index, x=x)
        return MLP((1 + eps) x + out)

    def message(self, x_j):
        return x_j

    def message_and_aggregate(self, adj_t, x):
        return matmul(adj_t, x, reduce=self.aggr)
```

02 Use of SparseTensor in MessagePassing

The method `message_and_aggregate()` :

- It combines the `message` and `aggregate` methods in one single call that uses sparse matrix operations.
- It is automatically called whenever implemented!

```
from torch_sparse import matmul

class GINConv(MessagePassing):
    def __init__(self):
        super().__init__(aggr="add")

    def forward(self, x, edge_index):
        out = self.propagate(edge_index, x=x)
        return MLP((1 + eps) x + out)

    def message(self, x_j):
        return x_j

    def message_and_aggregate(self, adj_t, x):
        return matmul(adj_t, x, reduce=self.aggr)
```

Sparse matrix-vector multiplication

THANKS

Questions?

gsantin@fbk.eu