Note

Click here to download the full example code

Write your own GNN module

Sometimes, your model goes beyond simply stacking existing GNN modules. For example, you would like to invent a new way of aggregating neighbor information by considering node importance or edge weights.

By the end of this tutorial you will be able to

- Understand DGL's message passing APIs.
- Implement GraphSAGE convolution module by your own.

This tutorial assumes that you already know the basics of training a GNN for node classification.

(Time estimate: 10 minutes)

```
import os

os.environ["DGLBACKEND"] = "pytorch"
import dgl
import dgl.function as fn
import torch
import torch.nn as nn
import torch.nn.functional as F
```

Message passing and GNNs

DGL follows the *message passing paradigm* inspired by the Message Passing Neural Network proposed by Gilmer et al. Essentially, they found many GNN models can fit into the following framework:

$$m_{u\to v}^{(l)} = M^{(l)}\left(h_v^{(l-1)}, h_u^{(l-1)}, e_{u\to v}^{(l-1)}\right)$$

$$m_v^{(l)} = \sum_{u\in\mathcal{N}(v)} m_{u\to v}^{(l)}$$

$$h_v^{(l)} = U^{(l)}\left(h_v^{(l-1)}, m_v^{(l)}\right)$$

where DGL calls $M^{(l)}$ the *message function*, $\sum$ the *reduce function* and $U^{(l)}$ the *update function*. Note that $\sum$ here can represent any function and is not necessarily a summation.

For example, the GraphSAGE convolution (Hamilton et al., 2017) takes the following mathematical form:

$$h_{\mathcal{N}(v)}^k \leftarrow \text{Average}\{h_u^{k-1}, \forall u \in \mathcal{N}(v)\}$$

$$h_v^k \leftarrow \text{ReLU}\left(W^k \cdot \text{CONCAT}(h_v^{k-1}, h_{\mathcal{N}(v)}^k)\right)$$

You can see that message passing is directional: the message sent from one node u to other node v is not necessarily the same as the other message sent from node v to node u in the opposite direction.

Although DGL has builtin support of GraphSAGE via `dgl.nn.SAGEConv`, here is how you can implement GraphSAGE convolution in DGL by your own.

```python
class SAGEConv(nn.Module):
    """Graph convolution module used by the GraphSAGE model.

    Parameters
    ----------
    in_feat : int
        Input feature size.
    out_feat : int
        Output feature size.
    """

    def __init__(self, in_feat, out_feat):
        super(SAGEConv, self).__init__()
        # A linear submodule for projecting the input and neighbor feature to the output.
        self.linear = nn.Linear(in_feat * 2, out_feat)

    def forward(self, g, h):
        """Forward computation

        Parameters
        ----------
        g : Graph
            The input graph.
        h : Tensor
            The input node feature.
        """
        with g.local_scope():
            g.ndata["h"] = h
            # update_all is a message passing API.
            g.update_all(
                message_func=fn.copy_u("h", "m"),
                reduce_func=fn.mean("m", "h_N"),
            )
            h_N = g.ndata["h_N"]
            h_total = torch.cat([h, h_N], dim=1)
            return self.linear(h_total)
```

The central piece in this code is the `g.update_all` function, which gathers and averages the neighbor features. There are three concepts here:

- Message function `fn.copy_u('h', 'm')` that copies the node feature under name `'h'` as *messages* with name `'m'` sent to neighbors.
- Reduce function `fn.mean('m', 'h_N')` that averages all the received messages under name `'m'` and saves the result as a new node feature `'h_N'`.
- `update_all` tells DGL to trigger the message and reduce functions for all the nodes and edges.

Afterwards, you can stack your own GraphSAGE convolution layers to form a multi-layer GraphSAGE network.

```
class Model(nn.Module):
    def __init__(self, in_feats, h_feats, num_classes):
        super(Model, self).__init__()
        self.conv1 = SAGEConv(in_feats, h_feats)
        self.conv2 = SAGEConv(h_feats, num_classes)

    def forward(self, g, in_feat):
        h = self.conv1(g, in_feat)
        h = F.relu(h)
        h = self.conv2(g, h)
        return h
```

Training loop

The following code for data loading and training loop is directly copied from the introduction tutorial.

```python
import dgl.data

dataset = dgl.data.CoraGraphDataset()
g = dataset[0]


def train(g, model):
    optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
    all_logits = []
    best_val_acc = 0
    best_test_acc = 0

    features = g.ndata["feat"]
    labels = g.ndata["label"]
    train_mask = g.ndata["train_mask"]
    val_mask = g.ndata["val_mask"]
    test_mask = g.ndata["test_mask"]
    for e in range(200):
        # Forward
        logits = model(g, features)

        # Compute prediction
        pred = logits.argmax(1)

        # Compute loss
        # Note that we should only compute the losses of the nodes in the training set,
        # i.e. with train_mask 1.
        loss = F.cross_entropy(logits[train_mask], labels[train_mask])

        # Compute accuracy on training/validation/test
        train_acc = (pred[train_mask] == labels[train_mask]).float().mean()
        val_acc = (pred[val_mask] == labels[val_mask]).float().mean()
        test_acc = (pred[test_mask] == labels[test_mask]).float().mean()

        # Save the best validation accuracy and the corresponding test accuracy.
        if best_val_acc < val_acc:
            best_val_acc = val_acc
            best_test_acc = test_acc

        # Backward
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        all_logits.append(logits.detach())

        if e % 5 == 0:
            print(
                "In epoch {}, loss: {:.3f}, val acc: {:.3f} (best {:.3f}), test acc: {:.3f}
(best {:.3f})".format(
                    e, loss, val_acc, best_val_acc, test_acc, best_test_acc
                )
            )


model = Model(g.ndata["feat"].shape[1], 16, dataset.num_classes)
train(g, model)
```

Out:

```
  NumNodes: 2708
  NumEdges: 10556
  NumFeats: 1433
  NumClasses: 7
  NumTrainingSamples: 140
  NumValidationSamples: 500
  NumTestSamples: 1000
Done loading data from cached files.
In epoch 0, loss: 1.952, val acc: 0.114 (best 0.114), test acc: 0.103 (best 0.103)
In epoch 5, loss: 1.881, val acc: 0.124 (best 0.124), test acc: 0.137 (best 0.137)
In epoch 10, loss: 1.743, val acc: 0.524 (best 0.524), test acc: 0.519 (best 0.519)
In epoch 15, loss: 1.527, val acc: 0.540 (best 0.542), test acc: 0.530 (best 0.530)
In epoch 20, loss: 1.240, val acc: 0.586 (best 0.586), test acc: 0.570 (best 0.570)
In epoch 25, loss: 0.912, val acc: 0.634 (best 0.634), test acc: 0.636 (best 0.636)
In epoch 30, loss: 0.600, val acc: 0.680 (best 0.680), test acc: 0.686 (best 0.686)
In epoch 35, loss: 0.357, val acc: 0.706 (best 0.706), test acc: 0.732 (best 0.732)
In epoch 40, loss: 0.199, val acc: 0.716 (best 0.720), test acc: 0.747 (best 0.737)
In epoch 45, loss: 0.111, val acc: 0.716 (best 0.720), test acc: 0.755 (best 0.737)
In epoch 50, loss: 0.065, val acc: 0.724 (best 0.724), test acc: 0.757 (best 0.757)
In epoch 55, loss: 0.041, val acc: 0.718 (best 0.724), test acc: 0.757 (best 0.757)
In epoch 60, loss: 0.028, val acc: 0.718 (best 0.724), test acc: 0.759 (best 0.757)
```

More customization

In DGL, we provide many built-in message and reduce functions under the `dgl.function` package. You can find more details in the API doc.

These APIs allow one to quickly implement new graph convolution modules. For example, the following implements a new `SAGEConv` that aggregates neighbor representations using a weighted average. Note that `edata` member can hold edge features which can also take part in message passing.

```
class WeightedSAGEConv(nn.Module):
    """Graph convolution module used by the GraphSAGE model with edge weights.

    Parameters
    ----------
    in_feat : int
        Input feature size.
    out_feat : int
        Output feature size.
    """

    def __init__(self, in_feat, out_feat):
        super(WeightedSAGEConv, self).__init__()
        # A linear submodule for projecting the input and neighbor feature to the output.
        self.linear = nn.Linear(in_feat * 2, out_feat)

    def forward(self, g, h, w):
        """Forward computation

        Parameters
        ----------
        g : Graph
            The input graph.
        h : Tensor
            The input node feature.
        w : Tensor
            The edge weight.
        """
        with g.local_scope():
            g.ndata["h"] = h
            g.edata["w"] = w
            g.update_all(
                message_func=fn.u_mul_e("h", "w", "m"),
                reduce_func=fn.mean("m", "h_N"),
            )
            h_N = g.ndata["h_N"]
            h_total = torch.cat([h, h_N], dim=1)
            return self.linear(h_total)
```

Because the graph in this dataset does not have edge weights, we manually assign all edge weights to one in the `forward()` function of the model. You can replace it with your own edge weights.

```
class Model(nn.Module):
    def __init__(self, in_feats, h_feats, num_classes):
        super(Model, self).__init__()
        self.conv1 = WeightedSAGEConv(in_feats, h_feats)
        self.conv2 = WeightedSAGEConv(h_feats, num_classes)

    def forward(self, g, in_feat):
        h = self.conv1(g, in_feat, torch.ones(g.num_edges(), 1).to(g.device))
        h = F.relu(h)
        h = self.conv2(g, h, torch.ones(g.num_edges(), 1).to(g.device))
        return h


model = Model(g.ndata["feat"].shape[1], 16, dataset.num_classes)
train(g, model)
```

Out:

```
In epoch 0, loss: 1.948, val acc: 0.162 (best 0.162), test acc: 0.149 (best 0.149)
In epoch 5, loss: 1.867, val acc: 0.432 (best 0.432), test acc: 0.435 (best 0.435)
In epoch 10, loss: 1.710, val acc: 0.264 (best 0.432), test acc: 0.253 (best 0.435)
In epoch 15, loss: 1.474, val acc: 0.336 (best 0.432), test acc: 0.326 (best 0.435)
In epoch 20, loss: 1.177, val acc: 0.448 (best 0.448), test acc: 0.430 (best 0.430)
In epoch 25, loss: 0.855, val acc: 0.564 (best 0.564), test acc: 0.580 (best 0.580)
In epoch 30, loss: 0.561, val acc: 0.684 (best 0.684), test acc: 0.680 (best 0.680)
In epoch 35, loss: 0.338, val acc: 0.726 (best 0.726), test acc: 0.722 (best 0.722)
In epoch 40, loss: 0.194, val acc: 0.732 (best 0.740), test acc: 0.732 (best 0.729)
In epoch 45, loss: 0.112, val acc: 0.732 (best 0.740), test acc: 0.738 (best 0.729)
In epoch 50, loss: 0.067, val acc: 0.738 (best 0.740), test acc: 0.739 (best 0.729)
In epoch 55, loss: 0.043, val acc: 0.734 (best 0.740), test acc: 0.743 (best 0.729)
In epoch 60, loss: 0.029, val acc: 0.728 (best 0.740), test acc: 0.740 (best 0.729)
In epoch 65, loss: 0.021, val acc: 0.730 (best 0.740), test acc: 0.738 (best 0.729)
In epoch 70, loss: 0.017, val acc: 0.726 (best 0.740), test acc: 0.737 (best 0.729)
In epoch 75, loss: 0.014, val acc: 0.728 (best 0.740), test acc: 0.738 (best 0.729)
In epoch 80, loss: 0.012, val acc: 0.728 (best 0.740), test acc: 0.738 (best 0.729)
In epoch 85, loss: 0.010, val acc: 0.728 (best 0.740), test acc: 0.736 (best 0.729)
In epoch 90, loss: 0.009, val acc: 0.726 (best 0.740), test acc: 0.736 (best 0.729)
In epoch 95, loss: 0.008, val acc: 0.726 (best 0.740), test acc: 0.733 (best 0.729)
In epoch 100, loss: 0.007, val acc: 0.724 (best 0.740), test acc: 0.733 (best 0.729)
```

Even more customization by user-defined function

DGL allows user-defined message and reduce function for the maximal expressiveness. Here is a user-defined message function that is equivalent to `fn.u_mul_e('h', 'w', 'm')`.

```python
def u_mul_e_udf(edges):
    return {"m": edges.src["h"] * edges.data["w"]}
```

`edges` has three members: `src`, `data` and `dst`, representing the source node feature, edge feature, and destination node feature for all edges.

You can also write your own reduce function. For example, the following is equivalent to the builtin `fn.mean('m', 'h_N')` function that averages the incoming messages:

```python
def mean_udf(nodes):
    return {"h_N": nodes.mailbox["m"].mean(1)}
```

In short, DGL will group the nodes by their in-degrees, and for each group DGL stacks the incoming messages along the second dimension. You can then perform a reduction along the second dimension to aggregate messages.

For more details on customizing message and reduce function with user-defined function, please refer to the API reference.

Best practice of writing custom GNN modules

DGL recommends the following practice ranked by preference:

- Use `dgl.nn` modules.
- Use `dgl.nn.functional` functions which contain lower-level complex operations such as computing a softmax for each node over incoming edges.
- Use `update_all` with builtin message and reduce functions.
- Use user-defined message or reduce functions.

What's next?

- Writing Efficient Message Passing Code.

```
# Thumbnail credits: Representation Learning on Networks, Jure Leskovec, WWW 2018
# sphinx_gallery_thumbnail_path = '_static/blitz_3_message_passing.png'
```

Total running time of the script: (0 minutes 13.880 seconds)

Download Python source code: 3_message_passing.py

Download Jupyter notebook: 3_message_passing.ipynb