5.1 Node Classification/Regression

(中文版)

One of the most popular and widely adopted tasks for graph neural networks is node classification, where each node in the training/validation/test set is assigned a ground truth category from a set of predefined categories. Node regression is similar, where each node in the training/validation/test set is assigned a ground truth number.

Overview

To classify nodes, graph neural network performs message passing discussed in Chapter 2: Message Passing to utilize the node's own features, but also its neighboring node and edge features. Message passing can be repeated multiple rounds to incorporate information from larger range of neighborhood.

Writing neural network model

DGL provides a few built-in graph convolution modules that can perform one round of message passing. In this guide, we choose `dgl.nn.pytorch.SAGEConv` (also available in MXNet and Tensorflow), the graph convolution module for GraphSAGE.

Usually for deep learning models on graphs we need a multi-layer graph neural network, where we do multiple rounds of message passing. This can be achieved by stacking graph convolution modules as follows.

```
# Contruct a two-layer GNN model
import dgl.nn as dglnn
import torch.nn as nn
import torch.nn.functional as F
class SAGE(nn.Module):
    def __init__(self, in_feats, hid_feats, out_feats):
        super().__init__()
        self.conv1 = dglnn.SAGEConv(
            in_feats=in_feats, out_feats=hid_feats, aggregator_type='mean')
        self.conv2 = dglnn.SAGEConv(
            in_feats=hid_feats, out_feats=out_feats, aggregator_type='mean')

    def forward(self, graph, inputs):
        # inputs are features of nodes
        h = self.conv1(graph, inputs)
        h = F.relu(h)
        h = self.conv2(graph, h)
        return h
```

Note that you can use the model above for not only node classification, but also obtaining hidden node representations for other downstream tasks such as 5.2 Edge Classification/Regression, 5.3 Link Prediction, or 5.4 Graph Classification.

For a complete list of built-in graph convolution modules, please refer to apinn.

For more details in how DGL neural network modules work and how to write a custom neural network module with message passing please refer to the example in Chapter 3: Building GNN Modules.

Training loop

Training on the full graph simply involves a forward propagation of the model defined above, and computing the loss by comparing the prediction against ground truth labels on the training nodes.

This section uses a DGL built-in dataset `dgl.data.CiteseerGraphDataset` to show a training loop. The node features and labels are stored on its graph instance, and the training-validation-test split are also stored on the graph as boolean masks. This is similar to what you have seen in Chapter 4: Graph Data Pipeline.

```
node_features = graph.ndata['feat']
node_labels = graph.ndata['label']
train_mask = graph.ndata['train_mask']
valid_mask = graph.ndata['val_mask']
test_mask = graph.ndata['test_mask']
n_features = node_features.shape[1]
n_labels = int(node_labels.max().item() + 1)
```

The following is an example of evaluating your model by accuracy.

```
def evaluate(model, graph, features, labels, mask):
    model.eval()
    with torch.no_grad():
        logits = model(graph, features)
        logits = logits[mask]
        labels = labels[mask]
        _, indices = torch.max(logits, dim=1)
        correct = torch.sum(indices == labels)
        return correct.item() * 1.0 / len(labels)
```

You can then write our training loop as follows.

```
model = SAGE(in_feats=n_features, hid_feats=100, out_feats=n_labels)
opt = torch.optim.Adam(model.parameters())

for epoch in range(10):
    model.train()
    # forward propagation by using all nodes
    logits = model(graph, node_features)
    # compute loss
    loss = F.cross_entropy(logits[train_mask], node_labels[train_mask])
    # compute validation accuracy
    acc = evaluate(model, graph, node_features, node_labels, valid_mask)
    # backward propagation
    opt.zero_grad()
    loss.backward()
    opt.step()
    print(loss.item())

    # Save model if necessary.  Omitted in this example.
```

GraphSAGE provides an end-to-end homogeneous graph node classification example. You could see the corresponding model implementation is in the `GraphSAGE` class in the example with adjustable number of layers, dropout probabilities, and customizable aggregation functions and nonlinearities.

Heterogeneous graph

If your graph is heterogeneous, you may want to gather message from neighbors along all edge types. You can use the module `dgl.nn.pytorch.HeteroGraphConv` (also available in MXNet and Tensorflow) to perform message passing on all edge types, then combining different graph convolution modules for each edge type.

The following code will define a heterogeneous graph convolution module that first performs a separate graph convolution on each edge type, then sums the message aggregations on each edge type as the final result for all node types.

```
# Define a Heterograph Conv model

class RGCN(nn.Module):
    def __init__(self, in_feats, hid_feats, out_feats, rel_names):
        super().__init__()

        self.conv1 = dglnn.HeteroGraphConv({
            rel: dglnn.GraphConv(in_feats, hid_feats)
            for rel in rel_names}, aggregate='sum')
        self.conv2 = dglnn.HeteroGraphConv({
            rel: dglnn.GraphConv(hid_feats, out_feats)
            for rel in rel_names}, aggregate='sum')

    def forward(self, graph, inputs):
        # inputs are features of nodes
        h = self.conv1(graph, inputs)
        h = {k: F.relu(v) for k, v in h.items()}
        h = self.conv2(graph, h)
        return h
```

`dgl.nn.HeteroGraphConv` takes in a dictionary of node types and node feature tensors as input, and returns another dictionary of node types and node features.

So given that we have the user and item features in the heterogeneous graph example.

```
model = RGCN(n_hetero_features, 20, n_user_classes, hetero_graph.etypes)
user_feats = hetero_graph.nodes['user'].data['feature']
item_feats = hetero_graph.nodes['item'].data['feature']
labels = hetero_graph.nodes['user'].data['label']
train_mask = hetero_graph.nodes['user'].data['train_mask']
```

One can simply perform a forward propagation as follows:

```
node_features = {'user': user_feats, 'item': item_feats}
h_dict = model(hetero_graph, {'user': user_feats, 'item': item_feats})
h_user = h_dict['user']
h_item = h_dict['item']
```

Training loop is the same as the one for homogeneous graph, except that now you have a dictionary of node representations from which you compute the predictions. For instance, if you are only predicting the `user` nodes, you can just extract the `user` node embeddings from the returned dictionary:

```
opt = torch.optim.Adam(model.parameters())

for epoch in range(5):
    model.train()
    # forward propagation by using all nodes and extracting the user embeddings
    logits = model(hetero_graph, node_features)['user']
    # compute loss
    loss = F.cross_entropy(logits[train_mask], labels[train_mask])
    # Compute validation accuracy.  Omitted in this example.
    # backward propagation
    opt.zero_grad()
    loss.backward()
    opt.step()
    print(loss.item())

    # Save model if necessary.  Omitted in the example.
```

DGL provides an end-to-end example of RGCN for node classification. You can see the definition of heterogeneous graph convolution in `RelGraphConvLayer` in the model implementation file.