Note

Click here to download the full example code

Training a GNN for Graph Classification

By the end of this tutorial, you will be able to

- Load a DGL-provided graph classification dataset.
- Understand what *readout* function does.
- Understand how to create and use a minibatch of graphs.
- Build a GNN-based graph classification model.
- Train and evaluate the model on a DGL-provided dataset.

(Time estimate: 18 minutes)

```
import os

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

Overview of Graph Classification with GNN

Graph classification or regression requires a model to predict certain graph-level properties of a single graph given its node and edge features. Molecular property prediction is one particular application.

This tutorial shows how to train a graph classification model for a small dataset from the paper How Powerful Are Graph Neural Networks.

Loading Data

```
# Generate a synthetic dataset with 10000 graphs, ranging from 10 to 500 nodes.
dataset = dgl.data.GINDataset("PROTEINS", self_loop=True)
```

The dataset is a set of graphs, each with node features and a single label. One can see the node feature dimensionality and the number of possible graph categories of `GINDataset` objects in `dim_nfeats` and `gclasses` attributes.

```
print("Node feature dimensionality:", dataset.dim_nfeats)
print("Number of graph categories:", dataset.gclasses)


from dgl.dataloading import GraphDataLoader
```

Out:

```
Node feature dimensionality: 3
Number of graph categories: 2
```

Defining Data Loader

A graph classification dataset usually contains two types of elements: a set of graphs, and their graph-level labels. Similar to an image classification task, when the dataset is large enough, we need to train with mini-batches. When you train a model for image classification or language modeling, you will use a `DataLoader` to iterate over the dataset. In DGL, you can use the `GraphDataLoader`.

You can also use various dataset samplers provided in torch.utils.data.sampler. For example, this tutorial creates a training `GraphDataLoader` and test `GraphDataLoader`, using `SubsetRandomSampler` to tell PyTorch to sample from only a subset of the dataset.

```
from torch.utils.data.sampler import SubsetRandomSampler

num_examples = len(dataset)
num_train = int(num_examples * 0.8)

train_sampler = SubsetRandomSampler(torch.arange(num_train))
test_sampler = SubsetRandomSampler(torch.arange(num_train, num_examples))

train_dataloader = GraphDataLoader(
    dataset, sampler=train_sampler, batch_size=5, drop_last=False
)
test_dataloader = GraphDataLoader(
    dataset, sampler=test_sampler, batch_size=5, drop_last=False
)
```

You can try to iterate over the created `GraphDataLoader` and see what it gives:

```
it = iter(train_dataloader)
batch = next(it)
print(batch)
```

Out:

```
[Graph(num_nodes=257, num_edges=1217,
      ndata_schemes={'attr': Scheme(shape=(3,), dtype=torch.float32), 'label': Scheme(shape=(),
dtype=torch.int64)}
      edata_schemes={}), tensor([1, 1, 0, 1, 0])]
```

As each element in `dataset` has a graph and a label, the `GraphDataLoader` will return two objects for each iteration. The first element is the batched graph, and the second element is simply a label vector representing the category of each graph in the mini-batch. Next, we'll talked about the batched graph.

A Batched Graph in DGL

In each mini-batch, the sampled graphs are combined into a single bigger batched graph via `dgl.batch`. The single bigger batched graph merges all original graphs as separately connected components, with the node and edge features concatenated. This bigger graph is also a `DGLGraph` instance (so you can still treat it as a normal `DGLGraph` object as in here). It however contains the information necessary for recovering the original graphs, such as the number of nodes and edges of each graph element.

```
batched_graph, labels = batch
print(
    "Number of nodes for each graph element in the batch:",
    batched_graph.batch_num_nodes(),
)
print(
    "Number of edges for each graph element in the batch:",
    batched_graph.batch_num_edges(),
)

# Recover the original graph elements from the minibatch
graphs = dgl.unbatch(batched_graph)
print("The original graphs in the minibatch:")
print(graphs)
```

Out:

```
Number of nodes for each graph element in the batch: tensor([65, 27, 66, 10, 89])
Number of edges for each graph element in the batch: tensor([291, 127, 378,  46, 375])
The original graphs in the minibatch:
[Graph(num_nodes=65, num_edges=291,
      ndata_schemes={'attr': Scheme(shape=(3,), dtype=torch.float32), 'label': Scheme(shape=(),
dtype=torch.int64)}
      edata_schemes={}), Graph(num_nodes=27, num_edges=127,
      ndata_schemes={'attr': Scheme(shape=(3,), dtype=torch.float32), 'label': Scheme(shape=(),
dtype=torch.int64)}
      edata_schemes={}), Graph(num_nodes=66, num_edges=378,
      ndata_schemes={'attr': Scheme(shape=(3,), dtype=torch.float32), 'label': Scheme(shape=(),
dtype=torch.int64)}
      edata_schemes={}), Graph(num_nodes=10, num_edges=46,
      ndata_schemes={'attr': Scheme(shape=(3,), dtype=torch.float32), 'label': Scheme(shape=(),
dtype=torch.int64)}
      edata_schemes={}), Graph(num_nodes=89, num_edges=375,
      ndata_schemes={'attr': Scheme(shape=(3,), dtype=torch.float32), 'label': Scheme(shape=(),
dtype=torch.int64)}
      edata_schemes={})]
```

Define Model

This tutorial will build a two-layer Graph Convolutional Network (GCN). Each of its layer computes new node representations by aggregating neighbor information. If you have gone through the introduction, you will notice two differences:

- Since the task is to predict a single category for the *entire graph* instead of for every node, you will need to aggregate the representations of all the nodes and potentially the edges to form a graph-level representation. Such process is more commonly referred as a *readout*. A simple choice is to average the node features of a graph with `dgl.mean_nodes()`.
- The input graph to the model will be a batched graph yielded by the `GraphDataLoader`. The readout functions provided by DGL can handle batched graphs so that they will return one representation for each minibatch element.

```
from dgl.nn import GraphConv


class GCN(nn.Module):
    def __init__(self, in_feats, h_feats, num_classes):
        super(GCN, self).__init__()
        self.conv1 = GraphConv(in_feats, h_feats)
        self.conv2 = GraphConv(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)
        g.ndata["h"] = h
        return dgl.mean_nodes(g, "h")
```

Training Loop

The training loop iterates over the training set with the `GraphDataLoader` object and computes the gradients, just like image classification or language modeling.

```
# Create the model with given dimensions
model = GCN(dataset.dim_nfeats, 16, dataset.gclasses)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

for epoch in range(20):
    for batched_graph, labels in train_dataloader:
        pred = model(batched_graph, batched_graph.ndata["attr"].float())
        loss = F.cross_entropy(pred, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

num_correct = 0
num_tests = 0
for batched_graph, labels in test_dataloader:
    pred = model(batched_graph, batched_graph.ndata["attr"].float())
    num_correct += (pred.argmax(1) == labels).sum().item()
    num_tests += len(labels)

print("Test accuracy:", num_correct / num_tests)
```

Out:

```
Test accuracy: 0.2600896860986547
```

What's next

- See GIN example for an end-to-end graph classification model.

```
# Thumbnail credits: DGL
# sphinx_gallery_thumbnail_path = '_static/blitz_5_graph_classification.png'
```

Total running time of the script: (0 minutes 24.297 seconds)

Download Python source code: 5_graph_classification.py

Download Jupyter notebook: 5_graph_classification.ipynb