Note

Click here to download the full example code

Node Classification with DGL

GNNs are powerful tools for many machine learning tasks on graphs. In this introductory tutorial, you will learn the basic workflow of using GNNs for node classification, i.e. predicting the category of a node in a graph.

By completing this tutorial, you will be able to

- Load a DGL-provided dataset.
- Build a GNN model with DGL-provided neural network modules.
- Train and evaluate a GNN model for node classification on either CPU or GPU.

This tutorial assumes that you have experience in building neural networks with PyTorch.

(Time estimate: 13 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 Node Classification with GNN

One of the most popular and widely adopted tasks on graph data is node classification, where a model needs to predict the ground truth category of each node. Before graph neural networks, many proposed methods are using either connectivity alone (such as DeepWalk or node2vec), or simple combinations of connectivity and the node's own features. GNNs, by contrast, offers an opportunity to obtain node representations by combining the connectivity and features of a *local neighborhood*.

Kipf et al., is an example that formulates the node classification problem as a semi-supervised node classification task. With the help of only a small portion of labeled nodes, a graph neural network (GNN) can accurately predict the node category of the others.

This tutorial will show how to build such a GNN for semi-supervised node classification with only a small number of labels on the Cora dataset, a citation network with papers as nodes and citations as edges. The task is to predict the category of a given paper. Each paper node contains a word count vector as its features, normalized so that they sum up to one, as described in Section 5.2 of the paper.

Loading Cora Dataset

```
dataset = dgl.data.CoraGraphDataset()
print(f"Number of categories: {dataset.num_classes}")
```

Out:

```
  NumNodes: 2708
  NumEdges: 10556
  NumFeats: 1433
  NumClasses: 7
  NumTrainingSamples: 140
  NumValidationSamples: 500
  NumTestSamples: 1000
Done loading data from cached files.
Number of categories: 7
```

A DGL Dataset object may contain one or multiple graphs. The Cora dataset used in this tutorial only consists of one single graph.

```
g = dataset[0]
```

A DGL graph can store node features and edge features in two dictionary-like attributes called `ndata` and `edata`. In the DGL Cora dataset, the graph contains the following node features:

- `train_mask`: A boolean tensor indicating whether the node is in the training set.
- `val_mask`: A boolean tensor indicating whether the node is in the validation set.
- `test_mask`: A boolean tensor indicating whether the node is in the test set.
- `label`: The ground truth node category.
- `feat`: The node features.

```
print("Node features")
print(g.ndata)
print("Edge features")
print(g.edata)
```

Out:

```
Node features
{'feat': tensor([[0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.],
        [0., 0., 0.,  ..., 0., 0., 0.]]), 'label': tensor([3, 4, 4,  ..., 3, 3, 3]),
'test_mask': tensor([False, False, False,  ...,  True,  True,  True]), 'train_mask': tensor([
True,  True,  True,  ..., False, False, False]), 'val_mask': tensor([False, False, False,  ...,
False, False, False])}
Edge features
{}
```

Defining a Graph Convolutional Network (GCN)

This tutorial will build a two-layer Graph Convolutional Network (GCN). Each layer computes new node representations by aggregating neighbor information.

To build a multi-layer GCN you can simply stack `dgl.nn.GraphConv` modules, which inherit `torch.nn.Module`.

```
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)
        return h


# Create the model with given dimensions
model = GCN(g.ndata["feat"].shape[1], 16, dataset.num_classes)
```

DGL provides implementation of many popular neighbor aggregation modules. You can easily invoke them with one line of code.

Training the GCN

Training this GCN is similar to training other PyTorch neural networks.

```
def train(g, model):
    optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
    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(100):
        # Forward
        logits = model(g, features)

        # Compute prediction
        pred = logits.argmax(1)

        # Compute loss
        # Note that you should only compute the losses of the nodes in the training set.
        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()

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


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

Out:

```
In epoch 0, loss: 1.947, val acc: 0.064 (best 0.064), test acc: 0.086 (best 0.086)
In epoch 5, loss: 1.902, val acc: 0.584 (best 0.596), test acc: 0.574 (best 0.585)
In epoch 10, loss: 1.825, val acc: 0.680 (best 0.680), test acc: 0.673 (best 0.673)
In epoch 15, loss: 1.723, val acc: 0.670 (best 0.680), test acc: 0.665 (best 0.673)
In epoch 20, loss: 1.596, val acc: 0.684 (best 0.684), test acc: 0.685 (best 0.685)
In epoch 25, loss: 1.448, val acc: 0.692 (best 0.694), test acc: 0.695 (best 0.694)
In epoch 30, loss: 1.286, val acc: 0.672 (best 0.694), test acc: 0.689 (best 0.694)
In epoch 35, loss: 1.117, val acc: 0.688 (best 0.694), test acc: 0.695 (best 0.694)
In epoch 40, loss: 0.949, val acc: 0.698 (best 0.698), test acc: 0.699 (best 0.699)
In epoch 45, loss: 0.793, val acc: 0.698 (best 0.700), test acc: 0.700 (best 0.700)
In epoch 50, loss: 0.654, val acc: 0.720 (best 0.720), test acc: 0.716 (best 0.716)
In epoch 55, loss: 0.535, val acc: 0.728 (best 0.730), test acc: 0.724 (best 0.726)
In epoch 60, loss: 0.436, val acc: 0.738 (best 0.738), test acc: 0.731 (best 0.729)
In epoch 65, loss: 0.355, val acc: 0.756 (best 0.756), test acc: 0.743 (best 0.743)
In epoch 70, loss: 0.291, val acc: 0.762 (best 0.762), test acc: 0.745 (best 0.745)
In epoch 75, loss: 0.240, val acc: 0.766 (best 0.766), test acc: 0.748 (best 0.748)
In epoch 80, loss: 0.199, val acc: 0.768 (best 0.770), test acc: 0.753 (best 0.751)
In epoch 85, loss: 0.167, val acc: 0.766 (best 0.770), test acc: 0.758 (best 0.751)
In epoch 90, loss: 0.141, val acc: 0.770 (best 0.770), test acc: 0.759 (best 0.751)
In epoch 95, loss: 0.120, val acc: 0.772 (best 0.772), test acc: 0.758 (best 0.758)
```

Training on GPU

Training on GPU requires to put both the model and the graph onto GPU with the `to` method, similar to what you will do in PyTorch.

```
g = g.to('cuda')
model = GCN(g.ndata['feat'].shape[1], 16, dataset.num_classes).to('cuda')
train(g, model)
```

What's next?

- How does DGL represent a graph?
- Write your own GNN module.
- Link prediction (predicting existence of edges) on full graph.
- Graph classification.
- Make your own dataset.
- The list of supported graph convolution modules.
- The list of datasets provided by DGL.

```
# Thumbnail credits: Stanford CS224W Notes
# sphinx_gallery_thumbnail_path = '_static/blitz_1_introduction.png'
```

Total running time of the script: (0 minutes 1.967 seconds)

Download Python source code: 1_introduction.py

Download Jupyter notebook: 1_introduction.ipynb