Docs » Introduction by example

Introduction by example

We shortly introduce the fundamental concepts of PyTorch Geometric through self-contained examples. At its core, PyTorch Geometric provides the following main features:

- Data handling of graphs
- Common benchmark datasets
- Mini-batches
- Data transforms
- Learning methods on graphs

Data handling of graphs

A graph is used to model pairwise relations (edges) between objects (nodes). A single graph in PyTorch Geometric is described by an instance of `torch_geometric.data.Data`, which holds the following attributes by default:

- `data.x`: Node feature matrix with shape `[num_nodes, num_node_features]`
- `data.edge_index`: Graph connectivity in COO format with shape `[2, num_edges]` and type `torch.long`
- `data.edge_attr`: Edge feature matrix with shape `[num_edges, num_edge_features]`
- `data.y`: Target to train against (may have arbitrary shape)
- `data.pos`: Node position matrix with shape `[num_nodes, num_dimensions]`

None of these attributes is required. In fact, the `Data` object is not even restricted to these attributes. We can, *e.g.*, extend it by `data.face` to save the connectivity of triangles from a 3D mesh in a tensor with shape `[3, num_faces]` and type `torch.long`.

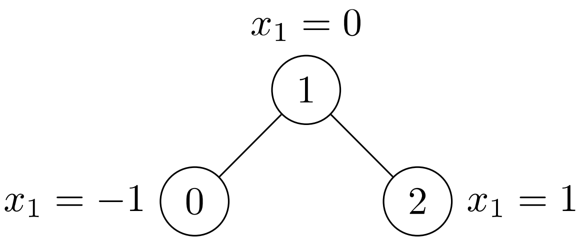
> **Note**
>
> PyTorch and `torchvision` define an example as a tuple of an image and a target. We omit this notation in PyTorch Geometric to allow for various data structures in a clean and understandable way.

We show a simple example of an unweighted and undirected graph with three nodes and four edges. Each node contains exactly one feature:

```
import torch
from torch_geometric.data import Data

edge_index = torch.tensor([[0, 1, 1, 2],
                           [1, 0, 2, 1]], dtype=torch.long)
x = torch.tensor([[-1], [0], [1]], dtype=torch.float)

data = Data(x=x, edge_index=edge_index)
>>> Data(x=[3, 1], edge_index=[2, 4])
```



Note that `edge_index`, *i.e.* the tensor defining the source and target nodes of all edges, is NOT a list of index tuples. If you want to write your indices this way, you should transpose and call `contiguous` on it before passing them to the data constructor:

```
import torch
from torch_geometric.data import Data

edge_index = torch.tensor([[0, 1],
                           [1, 0],
                           [1, 2],
                           [2, 1]], dtype=torch.long)
x = torch.tensor([[-1], [0], [1]], dtype=torch.float)

data = Data(x=x, edge_index=edge_index.t().contiguous())
>>> Data(x=[3, 1], edge_index=[2, 4])
```

Although the graph has only two edges, we need to define four index tuples to account for both directions of a edge.

Note

You can print out your data object anytime and receive a short information about its attributes and their shapes.

Besides of being a plain old python object, `torch_geometric.data.Data` provides a number of utility functions, *e.g.*:

```
print(data.keys)
>>> ['x', 'edge_index']

print(data['x'])
>>> tensor([[0.0],
            [1.0],
            [2.0]])

for key, item in data:
    print('{} found in data)
>>> x found in data
>>> edge_index found in data

'edge_attr' in data
>>> False

data.num_nodes
>>> 3

data.num_edges
>>> 4

data.num_features
>>> 1

data.contains_isolated_nodes()
>>> False

data.contains_self_loops()
>>> False

data.is_directed()
>>> False

# Transfer data object to GPU.
device = torch.device('cuda')
data = data.to(device)
```

You can find a complete list of all methods at `torch_geometric.data.Data`.

Common benchmark datasets

PyTorch Geometric contains a large number of common benchmark datasets, *e.g.* all Planetoid datasets (Cora, Citeseer, Pubmed), all graph classification datasets from http://graphkernels.cs.tu-dortmund.de/, the QM7 and QM9 dataset, and a handful of 3D mesh/point cloud datasets like FAUST, ModelNet10/40 and ShapeNet.

Initializing a dataset is straightforward. An initialization of a dataset will automatically download its raw files and process them to the previously described `Data` format. *E.g.*, to load the ENZYMES dataset (consisting of 600 graphs within 6 classes), type:

```
from torch_geometric.datasets import TUDataset

dataset = TUDataset(root='/tmp/ENZYMES', name='ENZYMES')
>>> ENZYMES(600)

len(dataset)
>>> 600

dataset.num_classes
>>> 6

dataset.num_features
>>> 21
```

We now have access to all 600 graphs in the dataset:

```
data = dataset[0]
>>> Data(x=[37, 21], edge_index=[2, 168], y=[1])

data.is_undirected()
>>> True
```

We can see that the first graph in the dataset contains 37 nodes, each one having 21 features. There are 168/2 = 84 undirected edges and the graph is assigned to exactly one class.

We can even use slices, long or byte tensors to split the dataset. *E.g.*, to create a 90/10 train/test split, type:

```
train_dataset = dataset[:540]
>>> ENZYMES(540)

test_dataset = dataset[540:]
>>> ENZYMES(60)
```

If you are unsure whether the dataset is already shuffled before you split, you can randomly permutate it by running:

```
dataset = dataset.shuffle()
>>> ENZYMES(600)
```

This is equivalent of doing:

```
perm = torch.randperm(len(dataset))
dataset = dataset[perm]
>> ENZYMES(600)
```

Let’s try another one! Let’s download Cora, the standard benchmark dataset for semi-supervised graph node classification:

```
from torch_geometric.datasets import Planetoid

dataset = Planetoid(root='/tmp/Cora', name='Cora')
>>> Cora()

len(dataset)
>>> 1

dataset.num_classes
>>> 7

dataset.num_features
>>> 1433
```

Here, the dataset contains only a single, undirected citation graph:

```
data = dataset[0]
>>> Data(x=[2708, 1433], edge_index=[2, 10556], y=[2708],
         train_mask=[2708], val_mask=[2708], test_mask=[2708])

data.is_undirected()
>>> True

data.train_mask.sum()
>>> 140

data.val_mask.sum()
>>> 500

data.test_mask.sum()
>>> 1000
```

This time, the `Data` objects holds additional attributes: `train_mask`, `val_mask` and `test_mask`:

- `train_mask` denotes against which nodes to train (140 nodes)
- `val_mask` denotes which nodes to use for validation, *e.g.*, to perform early stopping (500 nodes)
- `test_mask` denotes against which nodes to test (1000 nodes)

Mini-batches

Neural networks are usually trained in a batch-wise fashion. PyTorch Geometric achieves parallelization over a mini-batch by creating sparse block diagonal adjacency matrices (defined by `edge_index` and `edge_attr`) and concatenating feature and target matrices in the node dimension. This composition allows differing number of nodes and edges over examples in one batch:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & & \\ & \ddots & \\ & & \mathbf{A}_n \end{bmatrix}, \qquad \mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_n \end{bmatrix}, \qquad \mathbf{Y} = \begin{bmatrix} \mathbf{Y}_1 \\ \vdots \\ \mathbf{Y}_n \end{bmatrix}$$

PyTorch Geometric contains its own `torch_geometric.data.DataLoader`, which already takes care of this concatenation process. Let's learn about it in an example:

```
from torch_geometric.datasets import TUDataset
from torch_geometric.data import DataLoader

dataset = TUDataset(root='/tmp/ENZYMES', name='ENZYMES')
loader = DataLoader(dataset, batch_size=32, shuffle=True)

for batch in loader:
    batch
    >>> Batch(x=[1082, 21], edge_index=[2, 4066], y=[32], batch=[1082])

    batch.num_graphs
    >>> 32
```

`torch_geometric.data.Batch` inherits from `torch_geometric.data.Data` and contains an additional attribute: `batch`.

`batch` is a column vector of graph identifiers for all nodes of all graphs in the batch:

$$\mathrm{batch} = \begin{bmatrix} 0 & \cdots & 0 & 1 & \cdots & n-2 & n-1 & \cdots & n-1 \end{bmatrix}^\top$$

You can use it to, *e.g.*, average node features in the node dimension for each graph individually:

```
from torch_scatter import scatter_mean
from torch_geometric.datasets import TUDataset
from torch_geometric.data import DataLoader

dataset = TUDataset(root='/tmp/ENZYMES', name='ENZYMES')
loader = DataLoader(dataset, batch_size=32, shuffle=True)

for data in loader:
    data
    >>> Batch(x=[1082, 21], edge_index=[2, 4066], y=[32], batch=[1082])

    data.num_graphs
    >>> 32

    x = scatter_mean(data.x, data.batch, dim=0)
    x.size()
    >>> torch.Size([32, 21])
```

You can learn more about scatter operations in the documentation of `torch_scatter`.

Data transforms

Transforms are a common way in `torchvision` to transform images and perform augmentation. PyTorch Geometric comes with its own transforms, which expect a `Data` object as input and return a new transformed `Data` object. Transforms can be chained together using `torch_geometric.transforms.Compose` and are applied before saving a processed dataset on disk (`pre_transform`) or before accessing a graph in a dataset (`transform`).

Let's look at an example, where we apply transforms on the ShapeNet dataset (containing 17,000 3D shape point clouds and per point labels from 16 shape categories).

```
from torch_geometric.datasets import ShapeNet

dataset = ShapeNet(root='/tmp/ShapeNet', category='Airplane')

data[0]
>>> Data(pos=[2518, 3], y=[2518])
```

We can convert the point cloud dataset into a graph dataset by generating nearest neighbor graphs from the point clouds via transforms:

```
import torch_geometric.transforms as T
from torch_geometric.datasets import ShapeNet

dataset = ShapeNet(root='/tmp/ShapeNet', category='Airplane',
                    pre_transform=T.KNNGraph(k=6))

data[0]
>>> Data(edge_index=[2, 17768], pos=[2518, 3], y=[2518])
```

Note

We use the `pre_transform` to convert the data before saving it to disk (leading to faster loading times). Note that the next time the dataset is initialized it will already contain graph edges, even if you do not pass any transform.

In addition, we can use the `transform` argument to randomly augment a `Data` object, *e.g.* translating each node position by a small number:

```
import torch_geometric.transforms as T
from torch_geometric.datasets import ShapeNet

dataset = ShapeNet(root='/tmp/ShapeNet', category='Airplane',
                   pre_transform=T.NNGraph(k=6),
                   transform=T.RandomTranslate(0.01))

data[0]
>>> Data(edge_index=[2, 17768], pos=[2518, 3], y=[2518])
```

You can find a complete list of all implemented transforms at `torch_geometric.transforms`.

Learning methods on graphs

After learning about data handling, datasets, loader and transforms in PyTorch Geometric, it's time to implement our first graph neural network!

We will use a simple GCN layer and replicate the experiments on the Cora citation dataset. For a high-level explanation on GCN, have a look at its blog post.

We first need to load the Cora dataset:

```
from torch_geometric.datasets import Planetoid

dataset = Planetoid(root='/tmp/Cora', name='Cora')
>>> Cora()
```

Note that we do not need to use transforms or a dataloader. Now let's implement a two-layer GCN:

```
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv

class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = GCNConv(dataset.num_features, 16)
        self.conv2 = GCNConv(16, dataset.num_classes)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index

        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        x = self.conv2(x, edge_index)

        return F.log_softmax(x, dim=1)
```

We use ReLU as our non-linear activation function and output a softmax distribution over the number of classes. Let's train this model on the train nodes for 200 epochs:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = Net().to(device)
data = dataset[0].to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-4)

model.train()
for epoch in range(200):
    optimizer.zero_grad()
    out = model(data)
    loss = F.nll_loss(out[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()
```

Finally we can evaluate our model on the test nodes:

```
model.eval()
_, pred = model(data).max(dim=1)
correct = pred[data.test_mask].eq(data.y[data.test_mask]).sum().item()
acc = correct / data.test_mask.sum().item()
print('Accuracy: {:.4f}'.format(acc))
>>> Accuracy: 0.8150
```

That is all it takes to implement your first graph neural network. The easiest way to learn more about graph convolution and pooling is to study the examples in the `examples/` directory and to browse `torch_geometric.nn`. Happy hacking!