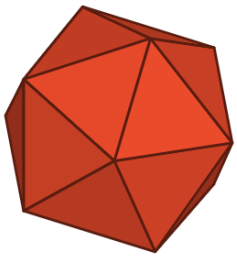
Tutorial

April 27, 2020

[1]:
```
import matplotlib.pyplot as plt
import sys

sys.stderr = sys.__stderr__
plt.rc('font', size=16)
```



PyTorch-Geometric Tutorial

Francesco Landolfi
Università di Pisa

1 Outline

1. Introduction
2. Sparse Data & Indexing in PyTorch
3. Framework Overview
4. Machine Learning with PyG
5. Conclusions

2 The "What"

- Python library for *Geometric Deep Learning*
- Written on top of PyTorch
- Provides utilities for sparse data
- CUDA/C++ routines for max performance

3 The "Why"

- A must-have if you are a (G)DL guy
- Only few more alternatives:
  - Deep Graph Library (DGL, PyTorch)
  - Stellar Graph, Euler (TF)

    - Other wannabes (<1K stars on GitHub)
- Many ready-to-use models and datasets
- Good for any Data-Parallel algorithm on graph

4 The "When"

You have an algorithm on graphs/meshes/point clouds and - you want to execute it on multiple samples in parallel - you want to exploit SIMD/GPU resources - you are able to **remodel** your algorithm as - a composition of simple algebraic operations, or - a *message-passing* model

Spoiler: Some algorithms are not easily remodelable!

5 The "How"

You need (of course) Python, PyTorch 1.4 and a few more libraries:

```
export CUDA=cu101  # or 'cpu', 'cu100', 'cu92'
pip install torch-scatter==latest+${CUDA} \\
    torch-sparse==latest+${CUDA} \\
    torch-cluster==latest+${CUDA} \\
    -f https://pytorch-geometric.com/whl/torch-1.4.0.html
pip install torch-geometric
```

Full docs here: https://pytorch-geometric.readthedocs.io/en/latest/

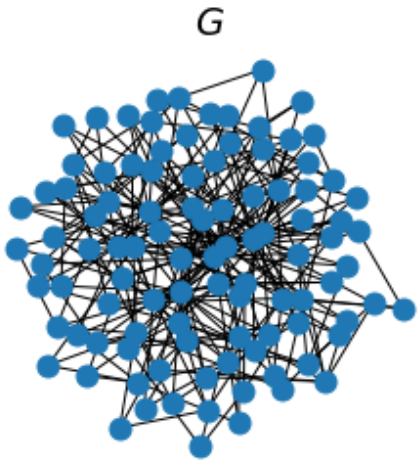
Note: To execute this notebook you will also need `networkx`, `matplotlib`, `trimesh`, `pandas`, `rdkit`, and `skorch`

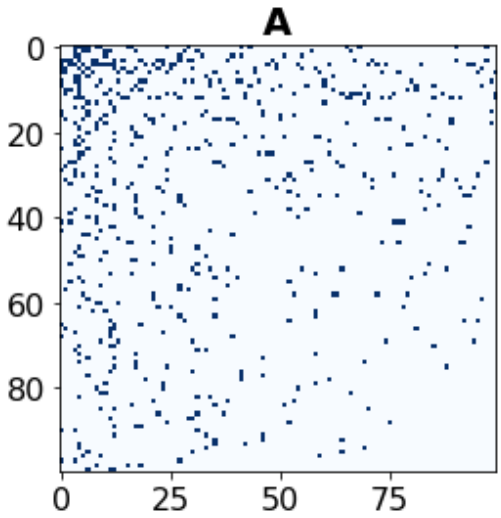
6 Dense v. Sparse

Example: Storing graph edges

- As matrix *(dense)* $\implies O(|V|^2)$
- As indices *(sparse)* $\implies O(|E|) \leq O(|V|^2)$

[2]:
```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.barabasi_albert_graph(100, 3)
_, axes = plt.subplots(1, 2, figsize=(10, 4), gridspec_kw={'wspace': 0.5})
nx.draw_kamada_kawai(G, ax=axes[0], node_size=120)
axes[1].imshow(nx.to_numpy_matrix(G), aspect='auto', cmap='Blues')
axes[0].set_title("$G$")
axes[1].set_title("$\mathbf{A}$")
plt.show()
```





7 Sparse Representations

We store a feature matrix $\mathbf{X} \in \mathbb{R}^{n \times h}$, then

- **Edges:** a matrix of indices $\mathbf{E} \in \mathbb{N}^{2 \times m}$
- **Triangles:** a matrix of indices $\mathbf{T} \in \mathbb{N}^{3 \times t}$
- **Attributes:** feature matrices $\mathbf{W}_E \in \mathbb{R}^{m \times h_e}$ and/or $\mathbf{W}_T \in \mathbb{R}^{t \times h_T}$,

8 Indexing/Slicing in PyTorch

Basically `tensor[idx, ...]` and `tensor[start:end:stride, ...]`

[3]:
```
import torch

mat = torch.arange(12).view(3, 4)
mat
```

[3]:
```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```

[4]:
```
mat[0]
```

[4]:
```
tensor([0, 1, 2, 3])
```

[5]:
```
mat[:, -1]
```

[5]:
```
tensor([ 3,  7, 11])
```

[6]:
```
mat[:, 2:]
```

[6]:
```
tensor([[ 2,  3],
        [ 6,  7],
        [10, 11]])
```

[7]:
```
mat[:, ::2]
```

[7]:
```
tensor([[ 0,  2],
        [ 4,  6],
        [ 8, 10]])
```

not only *R-values*...

[8]:
```
mat[:, ::2] = 42
mat
```

[8]:
```
tensor([[42,  1, 42,  3],
        [42,  5, 42,  7],
        [42,  9, 42, 11]])
```

[9]:
```
mat[:, 1::2] = -mat[:, ::2]
mat
```

[9]:
```
tensor([[ 42, -42,  42, -42],
        [ 42, -42,  42, -42],
        [ 42, -42,  42, -42]])
```

[10]:
```
mat[1::2] = -mat[::2]
mat
```

```
---------------------------------------------------------------------------

RuntimeError                              Traceback (most recent call last)

<ipython-input-10-afa2c45091a0> in <module>
----> 1 mat[1::2] = -mat[::2]
      2 mat


RuntimeError: The expanded size of the tensor (1) must match the existing␣
↪size (2) at non-singleton dimension 0.  Target sizes: [1, 4].  Tensor sizes:␣
↪[2, 4]
```

[11]:
```
mat[1, :, :] = 0
mat
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

<ipython-input-11-dd6fd5492b0d> in <module>
----> 1 mat[1, :, :] = 0
      2 mat


IndexError: too many indices for tensor of dimension 2
```

[13]:
```
mat[1, ..., 2] = 5
mat
```

[13]:
```
tensor([[ 42, -42,  42, -42],
        [  0,   0,   5,   0],
        [ 42, -42,  42, -42]])
```

9 Masked Selection

Using a `BoolTensor` to select values inside another `Tensor`

[14]:
```
rnd = torch.rand(3, 9)
rnd
```

[14]:
```
tensor([[0.0671, 0.4826, 0.5229, 0.9172, 0.0080, 0.6228, 0.3292, 0.5323,
0.4379],
        [0.3695, 0.1830, 0.5255, 0.0216, 0.6390, 0.5217, 0.1131, 0.4823,
0.8124],
        [0.9888, 0.4735, 0.1370, 0.2681, 0.6472, 0.4005, 0.3606, 0.9460,
0.6793]])
```

[15]:
```
mask = rnd >= 0.5
mask
```

[15]:
```
tensor([[False, False,  True,  True, False,  True, False,  True, False],
        [False, False,  True, False,  True,  True, False, False,  True],
        [ True, False, False, False,  True, False, False,  True,  True]])
```

[16]:
```
mask.type()
```

[16]:
```
'torch.BoolTensor'
```

[17]:
```
rnd[mask]
```

[17]:
```
tensor([0.5229, 0.9172, 0.6228, 0.5323, 0.5255, 0.6390, 0.5217, 0.8124, 0.9888,
        0.6472, 0.9460, 0.6793])
```

Note: Masking returns *always* a 1-D tensor!

[20]:
```
rnd[:, (~mask).all(0)]
```

[20]:
```
tensor([[0.4826, 0.3292],
        [0.1830, 0.1131],
        [0.4735, 0.3606]])
```

[21]:
```
rnd[mask] = 0
rnd
```

[21]:
```
tensor([[0.0671, 0.4826, 0.0000, 0.0000, 0.0080, 0.0000, 0.3292, 0.0000,
0.4379],
        [0.3695, 0.1830, 0.0000, 0.0216, 0.0000, 0.0000, 0.1131, 0.4823,
0.0000],
        [0.0000, 0.4735, 0.1370, 0.2681, 0.0000, 0.4005, 0.3606, 0.0000,
0.0000]])
```

10 Index selection

Using a `LongTensor` to select values at specific indices

[22]:
```
A = torch.randint(2, (5, 5))
A
```

[22]:
```
tensor([[0, 1, 1, 1, 1],
        [0, 0, 0, 0, 1],
        [1, 0, 0, 1, 0],
        [0, 1, 1, 1, 1],
        [0, 1, 1, 1, 1]])
```

[23]:
```
idx = A.nonzero().T
idx
```

[23]:
```
tensor([[0, 0, 0, 0, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4],
        [1, 2, 3, 4, 4, 0, 3, 1, 2, 3, 4, 1, 2, 3, 4]])
```

[24]:
```
A[idx]
```

[24]:
```
tensor([[[0, 1, 1, 1, 1],
         [0, 1, 1, 1, 1],
         [0, 1, 1, 1, 1],
```

```
        [0, 1, 1, 1, 1],
        [0, 0, 0, 0, 1],
        [1, 0, 0, 1, 0],
        [1, 0, 0, 1, 0],
        [0, 1, 1, 1, 1],
        [0, 1, 1, 1, 1],
        [0, 1, 1, 1, 1],
        [0, 1, 1, 1, 1],
        [0, 1, 1, 1, 1],
        [0, 1, 1, 1, 1],
        [0, 1, 1, 1, 1]],

       [[0, 0, 0, 0, 1],
        [1, 0, 0, 1, 0],
        [0, 1, 1, 1, 1],
        [0, 1, 1, 1, 1],
        [0, 1, 1, 1, 1],
        [0, 1, 1, 1, 1],
        [0, 1, 1, 1, 1],
        [0, 0, 0, 0, 1],
        [1, 0, 0, 1, 0],
        [0, 1, 1, 1, 1],
        [0, 1, 1, 1, 1],
        [0, 0, 0, 0, 1],
        [1, 0, 0, 1, 0],
        [0, 1, 1, 1, 1],
        [0, 1, 1, 1, 1]]])
```

[25]:
```
row, col = idx  # row, col = idx[0], idx[1]
A[row, col]
```

[25]:
```
tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

[26]:
```
weight = torch.randint(10, (idx.size(1),))
weight
```

[26]:
```
tensor([2, 2, 8, 3, 0, 9, 2, 2, 4, 1, 6, 3, 5, 5, 5])
```

[27]:
```
A[row, col] = weight
A
```

[27]:
```
tensor([[0, 2, 2, 8, 3],
        [0, 0, 0, 0, 0],
        [9, 0, 0, 2, 0],
        [0, 2, 4, 1, 6],
        [0, 3, 5, 5, 5]])
```

[28]:
```
w, perm = torch.sort(weight)
w, idx[:, perm]
```

[28]:
```
(tensor([0, 1, 2, 2, 2, 2, 3, 3, 4, 5, 5, 5, 6, 8, 9]),
 tensor([[1, 3, 2, 3, 0, 0, 0, 4, 3, 4, 4, 4, 3, 0, 2],
         [4, 3, 3, 1, 2, 1, 4, 1, 2, 2, 3, 4, 4, 3, 0]]))
```

11 Gathering

[29]:
```
rnd = torch.randint(10, (3, 9))
rnd
```

[29]:
```
tensor([[3, 5, 1, 4, 6, 0, 5, 6, 8],
        [0, 3, 1, 9, 5, 9, 8, 6, 8],
        [7, 8, 6, 0, 7, 2, 0, 7, 5]])
```

[30]:
```
sort, perm = torch.sort(rnd, dim=-1)
sort, perm
```

[30]:
```
(tensor([[0, 1, 3, 4, 5, 5, 6, 6, 8],
         [0, 1, 3, 5, 6, 8, 8, 9, 9],
         [0, 0, 2, 5, 6, 7, 7, 7, 8]]),
 tensor([[5, 2, 0, 3, 1, 6, 4, 7, 8],
         [0, 2, 1, 4, 7, 6, 8, 3, 5],
         [3, 6, 5, 8, 2, 0, 4, 7, 1]]))
```

[31]:
```
torch.gather(input=rnd, dim=-1, index=perm)
```

[31]:
```
tensor([[0, 1, 3, 4, 5, 5, 6, 6, 8],
        [0, 1, 3, 5, 6, 8, 8, 9, 9],
        [0, 0, 2, 5, 6, 7, 7, 7, 8]])
```

`input` and `index` *must have the same shape,* except along `dim`!

Example: Top-k elements of each row

[32]:
```
k = 3
torch.gather(input=rnd, dim=-1, index=perm[:, :k])
```

[32]:
```
tensor([[0, 1, 3],
        [0, 1, 3],
        [0, 0, 2]])
```

12 Scattering

[33]:
```
rnd, perm
```

[33]:
```
(tensor([[3, 5, 1, 4, 6, 0, 5, 6, 8],
         [0, 3, 1, 9, 5, 9, 8, 6, 8],
         [7, 8, 6, 0, 7, 2, 0, 7, 5]]),
 tensor([[5, 2, 0, 3, 1, 6, 4, 7, 8],
         [0, 2, 1, 4, 7, 6, 8, 3, 5],
         [3, 6, 5, 8, 2, 0, 4, 7, 1]]))
```

[34]:
```
torch.scatter(input=rnd, dim=-1, index=perm[:, :k], src=-torch.ones_like(rnd))
```

[34]:
```
tensor([[-1,  5, -1,  4,  6, -1,  5,  6,  8],
        [-1, -1, -1,  9,  5,  9,  8,  6,  8],
        [ 7,  8,  6, -1,  7, -1, -1,  7,  5]])
```

What if we assign multiple values to the same index?

[35]:
```
row, col
```

[35]:
```
(tensor([0, 0, 0, 0, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4]),
 tensor([1, 2, 3, 4, 4, 0, 3, 1, 2, 3, 4, 1, 2, 3, 4]))
```

[36]:
```
x = torch.arange(A.size(0))
torch.scatter(input=x, dim=-1, index=col, src=row)
```

[36]:
```
tensor([2, 4, 4, 4, 4])
```

Use `torch_scatter` to perform aggregations

[37]:
```
import torch_scatter

torch_scatter.scatter_min(src=row, index=col, dim=-1)
# torch_scatter.scatter_max(src=row, index=col, dim=-1)
# torch_scatter.scatter_add(src=row, index=col, dim=-1)
# torch_scatter.scatter_mean(src=row, index=col, dim=-1)
# torch_scatter.scatter_mul(src=row, index=col, dim=-1)
```

[37]:
```
(tensor([2, 0, 0, 0, 0]), tensor([5, 0, 1, 2, 3]))
```

13 Framework Overview

PyTorch-Geometric sub-modules:

- `nn`: contains (lots of) GNN models, pooling, normalizations
- `data`: classes for managing sparse and dense data

- `datasets`: basically every standard benchmark dataset for graphs, meshes, and point clouds, with different kinds of tasks
- `transform`: data manipulation functions
- `utils` and `io`: utility functions

14 `Data` Class

An (extensible) container for a single graph/mesh/whatever. Typical attributes:

- `x`: node features of size $n \times h_V$
- `edge_index`: edge indices, of size $2 \times m$
- `edge_attr`: edge attributes, of size $m \times h_E$
- `face`: triangle node indices, of size $3 \times t$
- `pos`: positions in the Euclidean space, of size $n \times d$
- `norm`: normal vectors, of size $n \times d$
- `y`: target of the sample (the size depends on the task)

15 Some Examples

You can define it yourself...

[38]:
```
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
```

[38]:
```
Data(edge_index=[2, 4], x=[3, 1])
```

16 Some Examples

...or load it from `trimesh`, `networkx`, `scipy`, etc.

[ ]:
```
!wget -nc https://raw.githubusercontent.com/mikedh/trimesh/master/models/bunny.
 ↪ply
```

[39]:
```
import trimesh

m = trimesh.load('bunny.ply')
m.show()
```

[39]:
```
<IPython.core.display.HTML object>
```

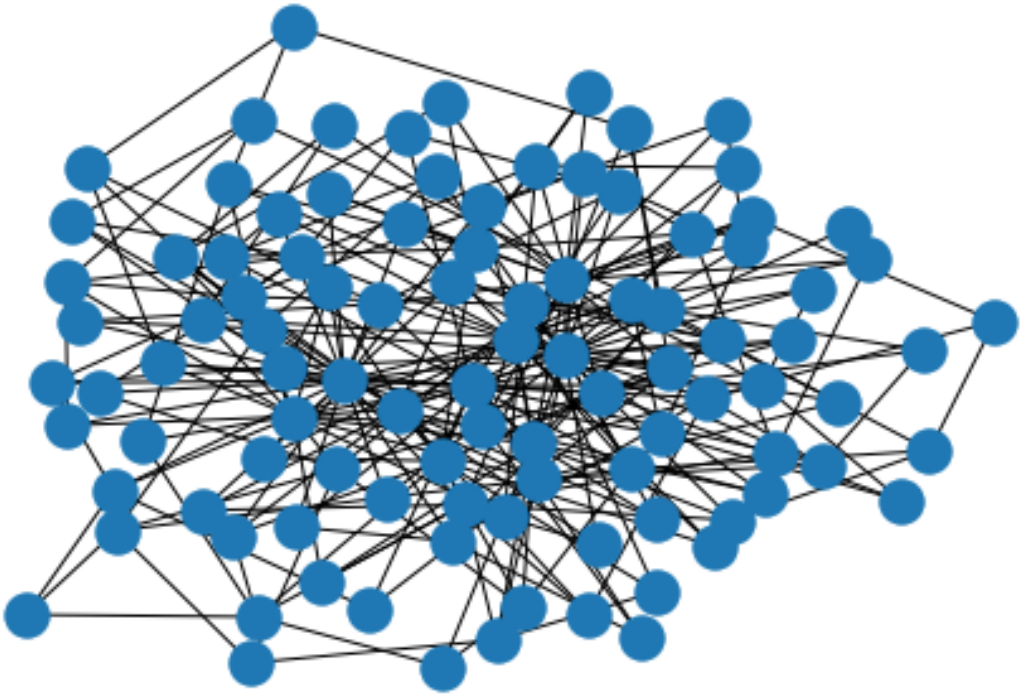
[40]:
```
from torch_geometric import utils

data = utils.from_trimesh(m)
data
```

[40]:
```
Data(face=[3, 16301], pos=[8146, 3])
```

[41]:
```
from torch_geometric import transforms

f2e = transforms.FaceToEdge(remove_faces=False)
f2e(data)
```

[41]:
```
Data(edge_index=[2, 48726], face=[3, 16301], pos=[8146, 3])
```

[42]:
```
import networkx as nx

G = nx.barabasi_albert_graph(n=100, m=3)
nx.draw_kamada_kawai(G)
```



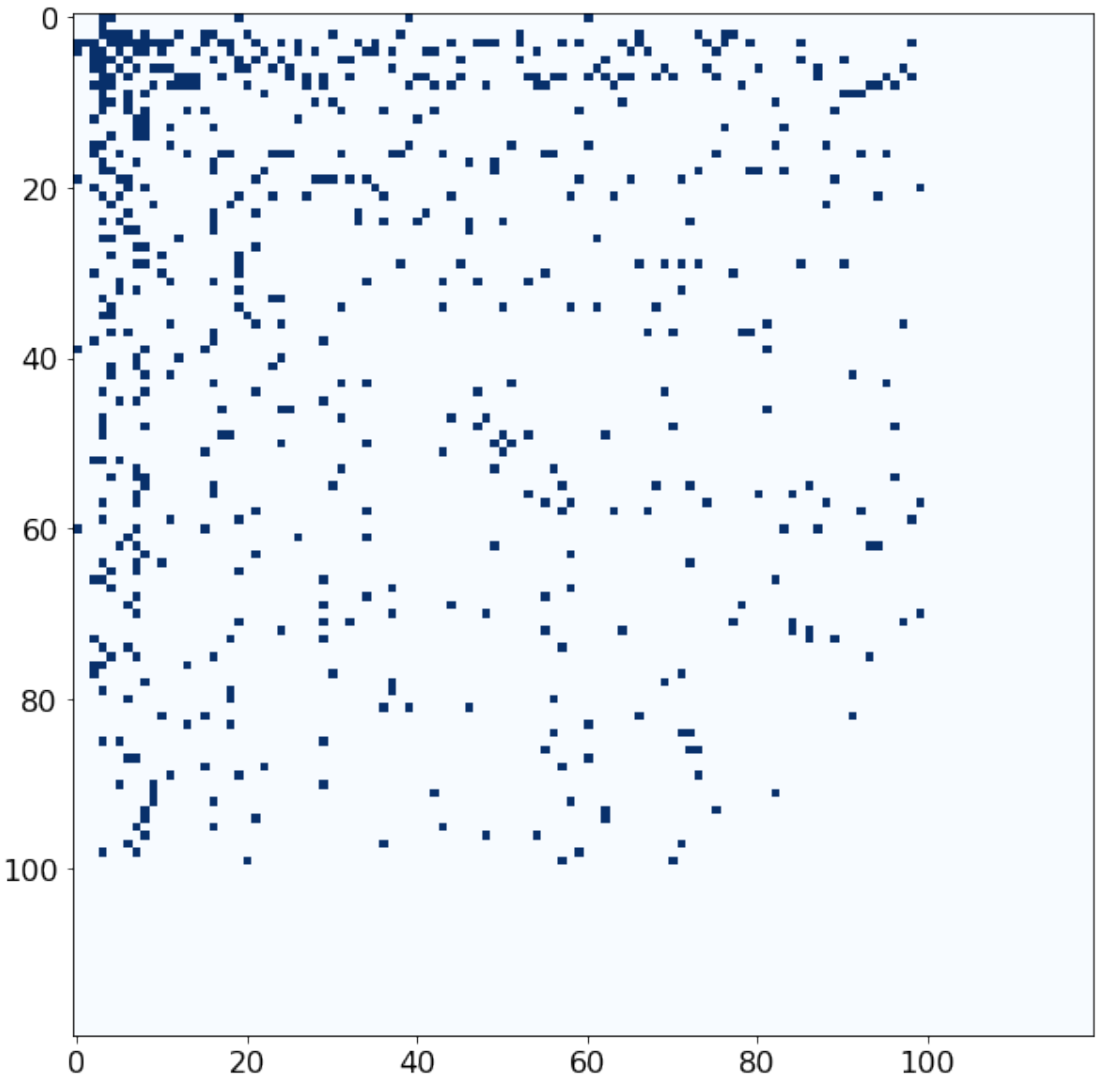
[43]:
```
data = utils.from_networkx(G)
data
```

[43]:
```
Data(edge_index=[2, 582])
```

[44]:
```
s2d = transforms.ToDense(num_nodes=120)
s2d(data)
```

[44]:
```
Data(adj=[120, 120], mask=[120])
```

[45]:
```
adj = data.adj.numpy()

_, ax = plt.subplots(figsize=(10, 10))
ax.imshow(adj, cmap='Blues')
plt.show()
```



17 Ready-to-use Datasets

Lots of them:

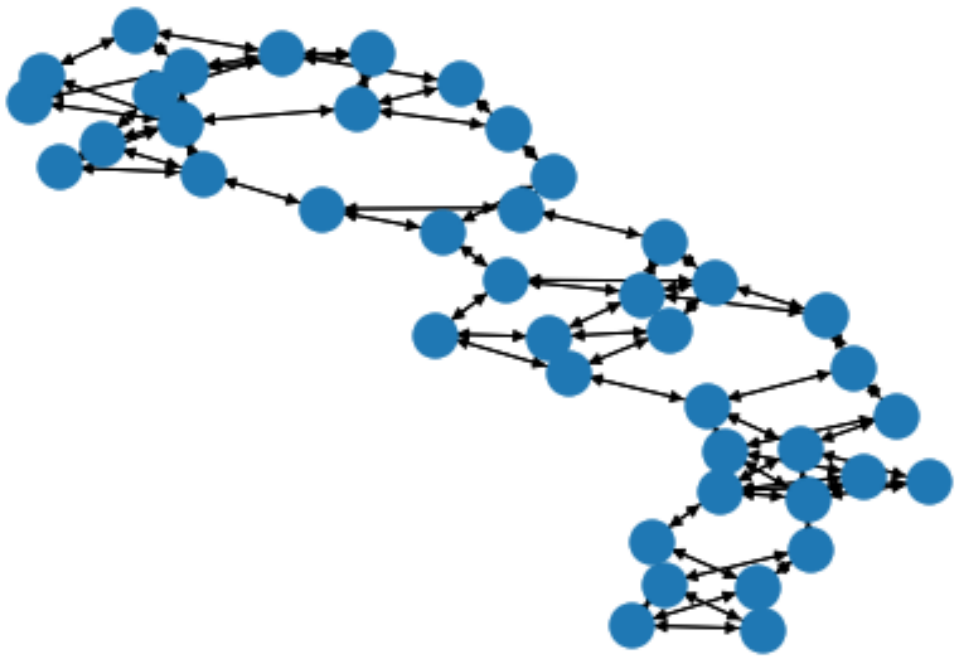
- Graph tasks: `TUDataset` collection, `Amazon`, `QM9`, ...
- Mesh tasks: `FAUST` and `DynamicFaust` human poses, `ModelNet`, `SHRECs`, ...
- Point Clouds tasks: `ShapeNet`, `S3DIS` in-door scenes, `PCPNetDataset`, ...

18 Some Examples

[46]:
```
from torch_geometric.datasets import TUDataset, ModelNet, ShapeNet

ds = TUDataset(root='./data/', name='PROTEINS')
ds
```

[46]:
```
PROTEINS(1113)
```

[47]:
```
G = utils.to_networkx(ds[0])
nx.draw_kamada_kawai(G)
```



[48]:
```
ds = ModelNet(root="./data/ModelNet/")
ds
```

[48]:
```
ModelNet10(3991)
```

[49]:
```
m = utils.to_trimesh(ds[0])
m.show()
```

[49]:
```
<IPython.core.display.HTML object>
```

19 DIY Dataset

You can create your own dataset by extending `InMemoryDataset`:

- Specify the data you *need* in `raw_file_names()`
- Specify the data you *generate* in `processed_file_names()`
- Implement `download()` and `process()`
- Load your (processed) data in `__init__()`

20 Example: COVID dataset

Fragments screened for 3CL protease binding, see https://www.aicures.mit.edu/

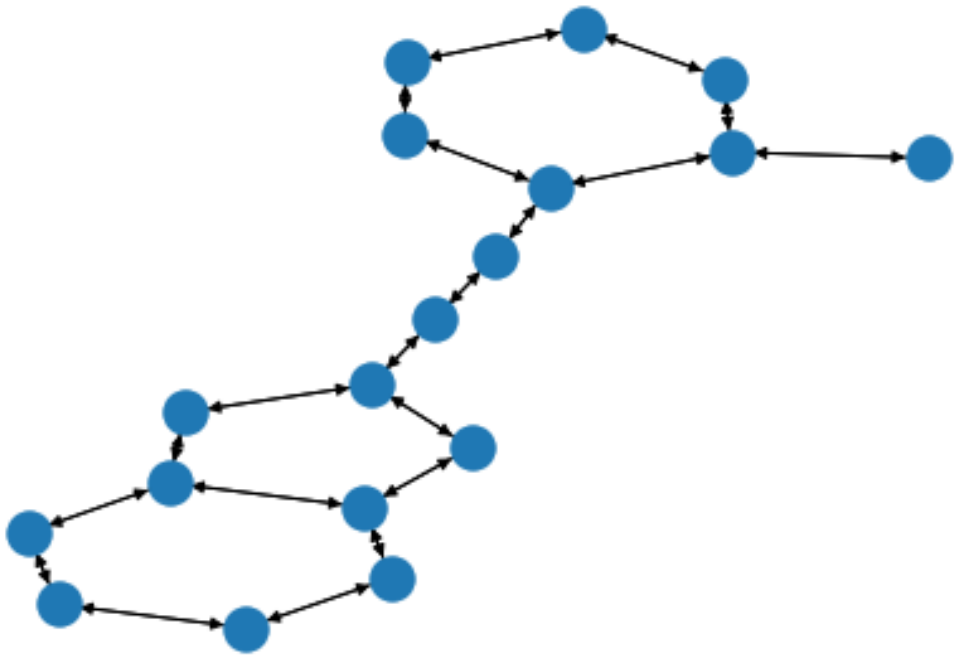
[50]:
```
from torch_geometric.data import InMemoryDataset, download_url
from rdkit import Chem
import pandas as pd


class COVID(InMemoryDataset):
    url = 'https://github.com/yangkevin2/coronavirus_data/raw/master/data/
 ↪mpro_xchem.csv'

    def __init__(self, root, transform=None, pre_transform=None,␣
 ↪pre_filter=None):
        super(COVID, self).__init__(root, transform, pre_transform, pre_filter)

        # Load processed data
        self.data, self.slices = torch.load(self.processed_paths[0])

    @property
    def raw_file_names(self):
        return ['mpro_xchem.csv']

    @property
    def processed_file_names(self):
        return ['data.pt']

    def download(self):
        download_url(self.url, self.raw_dir)
```

```
    def process(self):
        df = pd.read_csv(self.raw_paths[0])
        data_list = []

        for smiles, label in df.itertuples(False, None):
            mol = Chem.MolFromSmiles(smiles)  # Read the molecule info
            adj = Chem.GetAdjacencyMatrix(mol)  # Get molecule structure

            # You should extract other features here!
            data = Data(num_nodes=adj.shape[0],
                        edge_index=torch.Tensor(adj).nonzero().T,
                        y=label)

            data_list.append(data)

        self.data, self.slices = self.collate(data_list)
        torch.save((self.data, self.slices), self.processed_paths[0])
```

[51]:
```
covid = COVID(root='./data/COVID/')
covid
```

[51]:
```
COVID(880)
```

[52]:
```
G = utils.to_networkx(covid[0])
nx.draw_kamada_kawai(G)
```



21 Mini-Batches

Processing multiple graphs as a single, disconnected graph

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

$$\mathbf{A} = \begin{bmatrix} \widehat{\mathbf{A}}_1 \\ \vdots \\ \widehat{\mathbf{A}}_n \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_n \end{bmatrix}, \quad \mathbf{Y} = \begin{bmatrix} \mathbf{Y}_1 \\ \vdots \\ \mathbf{Y}_n \end{bmatrix}$$

22 `Batch` Class

A `Data` sub-class, representing a *sparse* batch of graphs

[53]:
```
from torch_geometric.data import Batch

b = Batch.from_data_list(covid[:10])
b
```

[53]:
```
Batch(batch=[166], edge_index=[2, 352], y=[10])
```

[54]:
```
b.batch
```

[54]:
```
tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
        3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
        4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
        5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7,
        7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8,
        8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9])
```

[55]:
```
b.edge_index[0]
```

[55]:
```
tensor([  0,   1,   1,   1,   2,   2,   3,   3,   4,   4,   5,   5,   6,   6,
          6,   7,   7,   8,   8,   9,   9,   9,  10,  10,  11,  11,  11,  12,
         12,  13,  13,  14,  14,  15,  15,  16,  16,  16,  17,  17,  18,  19,
         19,  19,  20,  21,  21,  22,  22,  23,  23,  24,  24,  24,  25,  25,
         26,  26,  27,  27,  27,  28,  28,  29,  29,  30,  30,  30,  31,  32,
         32,  33,  33,  33,  34,  35,  35,  35,  36,  36,  36,  37,  37,  37,
         38,  38,  38,  39,  39,  40,  40,  41,  41,  42,  42,  43,  43,  44,
         44,  45,  45,  46,  47,  48,  48,  48,  49,  49,  50,  50,  51,  51,
         52,  52,  52,  53,  53,  54,  54,  55,  55,  55,  56,  56,  57,  57,
         57,  58,  58,  58,  58,  59,  60,  61,  62,  63,  63,  63,  64,  65,
         65,  66,  66,  66,  67,  67,  68,  68,  69,  69,  69,  70,  70,  71,
         71,  71,  72,  72,  73,  73,  74,  74,  75,  75,  76,  76,  77,  77,
         78,  78,  79,  80,  80,  80,  81,  82,  82,  82,  83,  83,  84,  84,
         85,  85,  85,  86,  86,  87,  87,  88,  88,  88,  89,  90,  90,  91,
         91,  91,  92,  92,  93,  93,  94,  94,  95,  95,  96,  96,  97,  98,
         98,  99,  99,  99, 100, 101, 101, 101, 102, 102, 103, 103, 104, 104,
        104, 105, 105, 106, 106, 107, 107, 107, 107, 108, 109, 110, 110, 110,
        111, 111, 112, 112, 113, 113, 113, 114, 115, 115, 116, 116, 117, 118,
        118, 119, 119, 119, 120, 121, 121, 121, 122, 122, 122, 123, 123, 124,
        124, 124, 124, 125, 126, 127, 127, 128, 128, 129, 129, 129, 130, 130,
        131, 131, 132, 132, 133, 133, 134, 134, 135, 136, 136, 137, 137, 137,
        138, 139, 139, 139, 140, 140, 141, 141, 142, 142, 142, 143, 143, 144,
        144, 145, 145, 145, 145, 146, 147, 148, 149, 150, 150, 151, 151, 151,
        152, 153, 153, 153, 154, 154, 155, 155, 156, 156, 156, 157, 157, 158,
        158, 158, 159, 159, 160, 160, 161, 161, 161, 162, 163, 163, 164, 164,
        165, 165])
```

[56]:
```
covid[9].edge_index[0]
```

[56]:
```
tensor([ 0,  1,  1,  2,  2,  2,  3,  4,  4,  4,  5,  5,  6,  6,  7,  7,  7,  8,
         8,  9,  9,  9, 10, 10, 11, 11, 12, 12, 12, 13, 14, 14, 15, 15, 16, 16])
```

[57]:
```
Data.__inc__?
```

23 `MessagePassing` Class

Model a Neural Network (or any algorithm) by defining

- `message()`: the information "delivered" (by `propagate()`) from *source* to *target* (or viceversa) endnodes of an edge
  - can use data from both nodes (`x`, `pos`, or user-defined)
  - can use data from the edge itself (`edge_attr`)
- `aggregate()`: how the messages from the neighbors are aggregated
  - must be *permutation invariant*
  - no need to override if it's just `"add"`, `"mean"`, or `"max"`
- `update()`: a function applied to the aggregated messages

24 Example: Connected Components

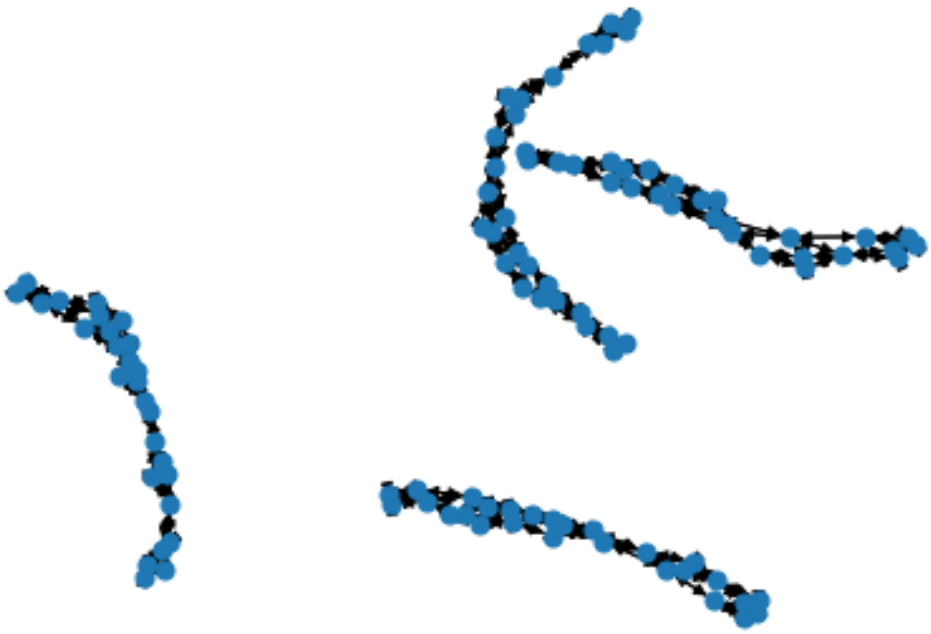
Typically done sequentially using Disjoint-Sets, we remodel as MP to exploit parallelism

[58]:
```python
from torch_geometric.nn import MessagePassing


class ConnectedComponents(MessagePassing):
    def __init__(self):
        super(ConnectedComponents, self).__init__(aggr="max")

    def forward(self, data):
        x = torch.arange(data.num_nodes).view(-1, 1)
        last_x = torch.zeros_like(x)

        while not x.equal(last_x):
            last_x = x.clone()
            x = self.propagate(data.edge_index, x=x)
            x = torch.max(x, last_x)

        unique, perm = torch.unique(x, return_inverse=True)
        perm = perm.view(-1)

        if "batch" not in data:
            return unique.size(0), perm

        cc_batch = unique.scatter(dim=-1, index=perm, src=data.batch)

        return cc_batch.bincount(minlength=data.num_graphs), perm

    def message(self, x_j):
        return x_j

    def update(self, aggr_out):
        return aggr_out
```

[59]:
```
ds = TUDataset(root='./data/', name='PROTEINS')
data = Batch.from_data_list(ds[:10])

cc = ConnectedComponents()
count, perm = cc(data)
count
```

[59]:
```
tensor([1, 1, 1, 1, 1, 8, 4, 1, 1, 1])
```

[60]:
```
G = utils.to_networkx(ds[6])
nx.draw(G, node_size=50)
```



25 Learning with PyG

Many, many standard GNNs & PointNets in the `nn` submodule

[61]:
```
from torch_geometric.nn import GCNConv, JumpingKnowledge, global_add_pool
from torch.nn import functional as F


class SimpleGNN(torch.nn.Module):
    def __init__(self, dataset, hidden=64, layers=6):
```

```
        super(SimpleGNN, self).__init__()

        self.dataset = dataset
        self.convs = torch.nn.ModuleList()

        self.convs.append(GCNConv(in_channels=dataset.num_node_features,
                                  out_channels=hidden))

        for _ in range(1, layers):
            self.convs.append(GCNConv(in_channels=hidden,
                                      out_channels=hidden))

        self.jk = JumpingKnowledge(mode="cat")
        self.jk_lin = torch.nn.Linear(in_features=hidden*layers,
                                      out_features=hidden)

        self.lin_1 = torch.nn.Linear(in_features=hidden,
                                     out_features=hidden)
        self.lin_2 = torch.nn.Linear(in_features=hidden,
                                     out_features=dataset.num_classes)

    def forward(self, index):
        data = Batch.from_data_list(self.dataset[index])

        x = data.x
        xs = []

        for conv in self.convs:
            x = F.relu(conv(x=x, edge_index=data.edge_index))
            xs.append(x)

        x = self.jk(xs)
        x = F.relu(self.jk_lin(x))
        x = global_add_pool(x, batch=data.batch)
        x = F.relu(self.lin_1(x))
        x = F.softmax(self.lin_2(x), dim=-1)

        return x
```

[62]:
```
ohd = transforms.OneHotDegree(max_degree=4)
covid = COVID(root='./data/COVID/', transform=ohd)
covid[0]
```

[62]:
```
Data(edge_index=[2, 40], x=[18, 5], y=[1])
```

[63]:
```
from skorch import NeuralNetClassifier
```

```
X, y = torch.arange(len(covid)).long(), covid.data.y

net = NeuralNetClassifier(
    module=SimpleGNN,
    module__dataset=covid,
    max_epochs=20,
    batch_size=-1,
    lr=0.001
)
```

[64]:
```
fit = net.fit(X, y)
```

```
  epoch    train_loss    valid_acc    valid_loss      dur
-------  ------------  -----------  ------------  ------
      1        0.6521       0.9091        0.6547
0.9067
      2        0.6463       0.9091        0.6504  0.9636
      3        0.6408       0.9091        0.6463  0.9287
      4        0.6355       0.9091        0.6422  0.7626
      5        0.6304       0.9091        0.6383  0.8893
      6        0.6254       0.9091        0.6345  0.9494
      7        0.6206       0.9091        0.6307  0.7778
      8        0.6158       0.9091        0.6270  0.8367
      9        0.6111       0.9091        0.6234  0.8407
     10        0.6064       0.9091        0.6199  0.7808
     11        0.6019       0.9091        0.6165  0.7656
     12        0.5975       0.9091        0.6131  0.7988
     13        0.5931       0.9091        0.6098  0.7455
     14        0.5889       0.9091        0.6065  0.8112
     15        0.5847       0.9091        0.6034  0.7360
     16        0.5807       0.9091        0.6003  0.8642
     17        0.5767       0.9091        0.5973  0.8519
     18        0.5728       0.9091        0.5944  0.8211
     19        0.5689       0.9091        0.5916  0.7989
     20        0.5652       0.9091        0.5889  0.8513
```

[65]:
```
1 - y.float().mean().item()
```

[65]:
```
0.9113636389374733
```

26 Conclusion

Pros: - Versatile framework - 0h spent recreating s.o.t.a. models - No need to look for data

Cons: - Steep learning curve - You need to get used to sparse data - Sometimes you need to create two models for one task (sparse/dense)

Questions?

Francesco Landolfi francesco.landolfi@phd.unipi.it