Hands-on Graph Neural Networks with PyTorch & PyTorch Geometric

黃功詳 Steeve Huang Follow
May 31 · 9 min read ★

In my last article, I introduced the concept of Graph Neural Network (GNN) and some recent advancements of it. Since this topic is getting seriously hyped up, I decided to make this tutorial on how to easily implement your Graph Neural Network in your project. You will learn how to construct your own GNN with PyTorch Geometric, and how to use GNN to solve a real-world problem (Recsys Challenge 2015).

In this blog post, we will be using PyTorch and PyTorch Geometric (PyG), a Graph Neural Network framework built on top of PyTorch that runs blazingly fast. Well … how fast is it? Compared to another popular Graph Neural Network Library, DGL, in terms of training time, it is at most 80% faster!!

Dataset	Epochs	Model	DGL-DB	DGL-GS	PyG
Cora	200	GCN	4.19s	0.32s	**0.25s**
		GAT	6.31s	5.36s	**0.80s**
CiteSeer	200	GCN	3.78s	0.34s	**0.30s**
		GAT	5.61s	4.91s	**0.88s**
PubMed	200	GCN	12.91s	0.36s	**0.32s**
		GAT	18.69s	13.76s	**2.42s**
MUTAG	200	RGCN	18.81s	2.40s	**2.14s**

DGL-DB: DGL 0.2 with *Degree Bucketing*.

DGL-GS: DGL 0.2 with *gather and scatter optimizations* inspired by PyG.

Training runtimes obtained on a NVIDIA GTX 1080Ti.

Benchmark Speed Test (https://github.com/rusty1s/pytorch_geometric)

Aside from its remarkable speed, PyG comes with a collection of well-implemented GNN models illustrated in various papers. Therefore, it would be very handy to reproduce the experiments with PyG.

- **SplineConv** from Fey *et al.*: SplineCNN: Fast Geometric Deep Learning with Continuous B-Spline Kernels (CVPR 2018)
- **GCNConv** from Kipf and Welling: Semi-Supervised Classification with Graph Convolutional Networks (ICLR 2017)
- **ChebConv** from Defferrard *et al.*: Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering (NIPS 2016)
- **NNConv** adapted from Gilmer *et al.*: Neural Message Passing for Quantum Chemistry (ICML 2017)
- **GATConv** from Veličković *et al.*: Graph Attention Networks (ICLR 2018)
- **SAGEConv** from Hamilton *et al.*: Inductive Representation Learning on Large Graphs (NIPS 2017)
- **GraphConv** from, *e.g.*, Morris *et al.*: Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks (AAAI 2019)
- **GatedGraphConv** from Li *et al.*: Gated Graph Sequence Neural Networks (ICLR 2016)
- **GINConv** from Xu *et al.*: How Powerful are Graph Neural Networks? (ICLR 2019)

A Subset of The Implemented Models (https://github.com/rusty1s/pytorch_geometric)

Given its advantage in speed and convenience, without a doubt, PyG is one of the most popular and widely used GNN libraries. Let's dive into the topic and get our hands dirty!

Requirements

- PyTorch—1.1.0
- PyTorch Geometric—1.2.0

PyTorch Geometric Basics

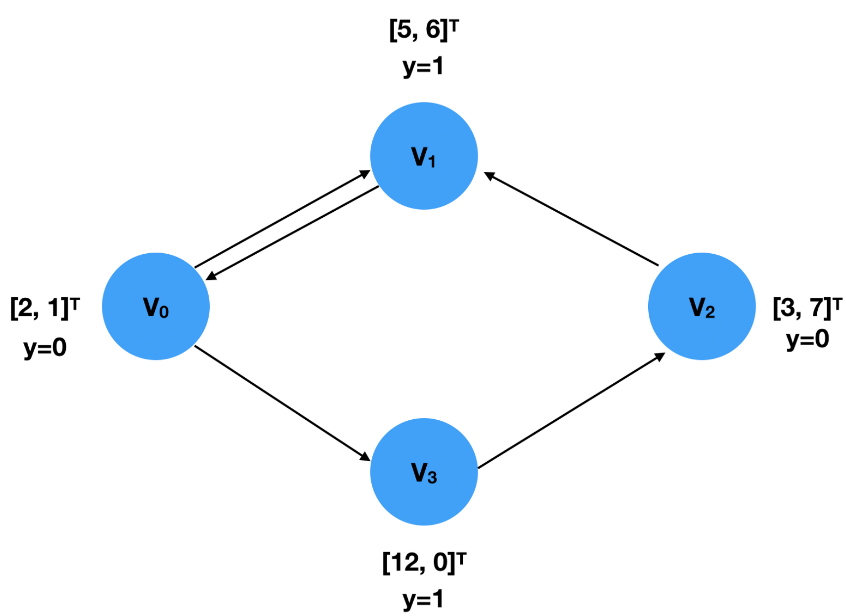
This section will walk you through the basics of PyG. Essentially, it will cover *torch_geometric.data* and *torch_geometric.nn*. You will learn how to pass geometric data into your GNN, and how to design a custom MessagePassing layer, the core of GNN.

Data

The *torch_geometric.data* module contains a Data class that allows you to create graphs from your data very easily. You only need to specify:

1. the attributes/ features associated with each node
2. the connectivity/adjacency of each node (edge index)

Let's use the following graph to demonstrate how to create a Data object



Example Graph

So there are 4 nodes in the graph, v1 … v4, each of which is associated with a 2-dimensional feature vector, and a label y indicating its class. These two can be represented as FloatTensors:

```
x = torch.tensor([[2,1], [5,6], [3,7], [12,0]], dtype=t
y = torch.tensor([0, 1, 0, 1], dtype=torch.float)
```

The graph connectivity (edge index) should be confined with the COO format, i.e. the first list contains the index of the source nodes, while the index of target nodes is specified in the second list.

```
edge_index = torch.tensor([[0, 1, 2, 0, 3],
                           [1, 0, 1, 3, 2]], dtype=torc
```

Note that the order of the edge index is irrelevant to the Data object you create since such information is only for computing the adjacency matrix. Therefore, the above edge_index express the same information as the following one.

```
edge_index = torch.tensor([[0, 2, 1, 0, 3],
                           [3, 1, 0, 1, 2]], dtype=torc
```

Putting them together, we can create a Data object as shown below:

```
import torch
from torch_geometric.data import Data


x = torch.tensor([[2,1], [5,6], [3,7], [12,0]], dtype=
y = torch.tensor([0, 1, 0, 1], dtype=torch.float)

edge_index = torch.tensor([[0, 2, 1, 0, 3],
                           [3, 1, 0, 1, 2]], dtype=tor
```

Dataset

The dataset creation procedure is not very straightforward, but it may seem familiar to those who've used *torchvision*, as PyG is following its convention. PyG provides two different types of dataset classes, InMemoryDataset and Dataset. As they indicate literally, the former one is for data that fit in your RAM, while the second one is for much larger data. Since their implementations are quite similar, I will only cover InMemoryDataset.

To create an InMemoryDataset object, there are 4 functions you need to implement:

- *raw_file_names()*

It returns a list that shows a list of raw, unprocessed file names. If you only have a file then the returned list should only contain 1 element. In fact, you can simply return an empty list and specify your file later in *process()*.

- *processed_file_names()*

Similar to the last function, it also returns a list containing the file names of all the processed data. After process() is called, Usually, the returned list should only have one element, storing the only processed data file name.

- *download()*

This function should download the data you are working on to the directory as specified in self.raw_dir. If you don't need to download data, simply drop in

```
pass
```

in the function.

- *process()*

This is the most important method of Dataset. You need to gather your data into a list of Data objects. Then, call *self.collate()* to compute the slices that will be used by the DataLoader object. The following shows an example of the custom dataset from PyG official website.

```
import torch
from torch_geometric.data import InMemoryDataset


class MyOwnDataset(InMemoryDataset):
    def __init__(self, root, transform=None, pre_trans
        super(MyOwnDataset, self).__init__(root, trans
        self.data, self.slices = torch.load(self.proce

    @property
    def raw_file_names(self):
        return ['some_file_1', 'some_file_2', ...]

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

    def download(self):
        # Download to `self.raw_dir`.

    def process(self):
```

I will show you how I create a custom dataset from the data provided in RecSys Challenge 2015 later in this article.

DataLoader

The DataLoader class allows you to feed data by batch into the model effortlessly. To create a DataLoader object, you simply specify the Dataset and the batch size you want.

```
loader = DataLoader(dataset, batch_size=512,
shuffle=True)
```

Every iteration of a DataLoader object yields a Batch object, which is very much like a Data object but with an attribute, "batch". It indicates which graph each node is associated with. Since a DataLoader aggregates *x*, *y*, and *edge_index* from different samples/graphs into Batches, the GNN model needs this "batch" information to know which nodes belong to the same graph within a batch to perform computation.

```
for batch in loader:
    batch
    >>> Batch(x=[1024, 21], edge_index=[2, 1568], y=
[512], batch=[1024])
```

MessagePassing

Message passing is the essence of GNN which describes how node embeddings are learned. I have talked about in my last post, so I will just briefly run through this with terms that conform to the PyG documentation.

$$\mathbf{x}_i^{(k)} = \gamma^{(k)} \left(\mathbf{x}_i^{(k-1)}, \square_{j \in \mathcal{N}(i)} \, \phi^{(k)} \left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{i,j} \right) \right)$$

Message Passing

x denotes the node embeddings, e denotes the edge features, ϕ denotes the **message** function, $\square$ denotes the **aggregation** function, γ denotes the **update** function. If the edges in the graph have no feature other than connectivity, e is essentially the edge index of the graph. The superscript represents the index of the layer. When k=1, x represents the input feature of each node. Below I will illustrate how each function works:

- *propagate(edge_index, size=None, **kwargs):*

It takes in edge index and other optional information, such as node features (embedding). Calling this function will consequently call *message* and *update*.

- *message(**kwargs):*

You specify how you construct "message" for each of the node pair (x_i, x_j). Since it follows the calls of *propagate*, it can take any argument passing to *propagate*. One thing to note is that you can define the mapping from arguments to the specific nodes with "_i" and "_j". Therefore, you must be very careful when naming the argument of this function.

- *update(aggr_out, **kwargs)*

It takes in the aggregated message and other arguments passed into *propagate*, assigning a new embedding value for each node.

Example

Let's see how we can implement a **SageConv** layer from the paper *"Inductive Representation Learning on Large Graphs"*. The message passing formula of SageConv is defined as:

$$\mathbf{h}^k_{\mathcal{N}(v)} \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}^{k-1}_u, \forall u \in \mathcal{N}(v)\});$$
$$\mathbf{h}^k_v \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}^{k-1}_v, \mathbf{h}^k_{\mathcal{N}(v)})\right)$$

https://arxiv.org/abs/1706.02216

Here, we use max pooling as the aggregation method. Therefore, the right-hand side of the first line can be written as:

$$\max(\{\sigma\left(\mathbf{W}_{\text{pool}}\mathbf{h}^k_{u_i} + \mathbf{b}\right), \forall u_i \in \mathcal{N}(v)\})$$

https://arxiv.org/abs/1706.02216

which illustrates how the "message" is constructed. Each neighboring node embedding is multiplied by a weight matrix, added a bias and passed through an activation function. This can be easily done with torch.nn.Linear.

```
class SAGEConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(SAGEConv, self).__init__(aggr='max')
```

```
        self.lin = torch.nn.Linear(in_channels,
out_channels)
        self.act = torch.nn.ReLU()

    def message(self, x_j):
        # x_j has shape [E, in_channels]

        x_j = self.lin(x_j)
        x_j = self.act(x_j)

        return x_j
```

As for the update part, the aggregated message and the current node embedding is aggregated. Then, it is multiplied by another weight matrix and applied another activation function.

```
class SAGEConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(SAGEConv, self).__init__(aggr='max')
        self.update_lin = torch.nn.Linear(in_channels
+ out_channels, in_channels, bias=False)
        self.update_act = torch.nn.ReLU()

    def update(self, aggr_out, x):
        # aggr_out has shape [N, out_channels]

        new_embedding = torch.cat([aggr_out, x],
dim=1)
        new_embedding = self.update_lin(new_embedding)
        new_embedding =
torch.update_act(new_embedding)

        return new_embedding
```

Putting it together, we have the following SageConv layer.

```
import torch
from torch.nn import Sequential as Seq, Linear, ReLU
from torch_geometric.nn import MessagePassing
from torch_geometric.utils import remove_self_loops, a
class SAGEConv(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super(SAGEConv, self).__init__(aggr='max') #
        self.lin = torch.nn.Linear(in_channels, out_ch
        self.act = torch.nn.ReLU()
        self.update_lin = torch.nn.Linear(in_channels
        self.update_act = torch.nn.ReLU()

    def forward(self, x, edge_index):
        # x has shape [N, in_channels]
        # edge_index has shape [2, E]


        edge_index, _ = remove_self_loops(edge_index)
        edge_index, _ = add_self_loops(edge_index, num


        return self.propagate(edge_index, size=(x.size

    def message(self, x_j):
        # x_j has shape [E, in_channels]

        x_j = self.lin(x_j)
```

A Real-World Example — RecSys Challenge 2015

The RecSys Challenge 2015 is challenging data scientists to build a session-based recommender system. Participants in this challenge are asked to solve two tasks:

1. Predict whether there will be a buy event followed by a sequence of clicks
2. Predict which item will be bought

First, we download the data from the official website of RecSys Challenge 2015 and construct a Dataset. We'll start with the first task as that one is easier.

The challenge provides two main sets of data, *yoochoose-clicks.dat*, and *yoochoose-buys.dat*, containing click events and buy events, respectively. Let's quickly glance through the data:

	session_id	timestamp	item_id	category
0	1	2014-04-07T10:51:09.277Z	214536502	0
1	1	2014-04-07T10:54:09.868Z	214536500	0
2	1	2014-04-07T10:54:46.998Z	214536506	0
3	1	2014-04-07T10:57:00.306Z	214577561	0
4	2	2014-04-07T13:56:37.614Z	214662742	0
5	2	2014-04-07T13:57:19.373Z	214662742	0
6	2	2014-04-07T13:58:37.446Z	214825110	0
7	2	2014-04-07T13:59:50.710Z	214757390	0
8	2	2014-04-07T14:00:38.247Z	214757407	0
9	2	2014-04-07T14:02:36.889Z	214551617	0
10	3	2014-04-02T13:17:46.940Z	214716935	0
11	3	2014-04-02T13:26:02.515Z	214774687	0
12	3	2014-04-02T13:30:12.318Z	214832672	0
13	4	2014-04-07T12:09:10.948Z	214836765	0
14	4	2014-04-07T12:26:25.416Z	214706482	0
15	6	2014-04-06T16:58:20.848Z	214701242	0
16	6	2014-04-06T17:02:26.976Z	214826623	0
17	7	2014-04-02T06:38:53.104Z	214826835	0
18	7	2014-04-02T06:39:05.854Z	214826715	0
19	8	2014-04-06T08:49:58.728Z	214838855	0

yoochoose-click.dat

	session_id	timestamp	item_id	price	quantity
0	420374	2014-04-06T18:44:58.314Z	214537888	12462	1
1	420374	2014-04-06T18:44:58.325Z	214537850	10471	1
2	281626	2014-04-06T09:40:13.032Z	214535653	1883	1
3	420368	2014-04-04T06:13:28.848Z	214530572	6073	1
4	420368	2014-04-04T06:13:28.858Z	214835025	2617	1
5	140806	2014-04-07T09:22:28.132Z	214668193	523	1
6	140806	2014-04-07T09:22:28.176Z	214587399	1046	1
7	140806	2014-04-07T09:22:28.219Z	214586690	837	1
8	140806	2014-04-07T09:22:28.268Z	214774667	1151	1
9	140806	2014-04-07T09:22:28.280Z	214578823	1046	1
10	11	2014-04-03T11:04:11.417Z	214821371	1046	1
11	11	2014-04-03T11:04:18.097Z	214821371	1046	1
12	12	2014-04-02T10:42:17.227Z	214717867	1778	4
13	489758	2014-04-06T09:59:52.422Z	214826955	1360	2
14	489758	2014-04-06T09:59:52.476Z	214826715	732	2
15	489758	2014-04-06T09:59:52.578Z	214827026	1046	1
16	140802	2014-04-02T16:52:21.678Z	214716984	1883	1
17	489756	2014-04-05T16:51:59.947Z	214716932	6177	1
18	420378	2014-04-03T07:41:02.566Z	214821017	1046	1
19	420378	2014-04-03T07:41:02.616Z	214821020	732	1

yoochoose-buys.dat

Preprocessing

After downloading the data, we preprocess it so that it can be fed to our model. item_ids are categorically encoded to ensure the encoded item_ids, which will later be mapped to an embedding matrix, starts at 0.

```
from sklearn.preprocessing import LabelEncoder

df = pd.read_csv('../input/yoochoose-click.dat', heade
df.columns=['session_id','timestamp','item_id','catego

buy_df = pd.read_csv('../input/yoochoose-buys.dat', he
buy_df.columns=['session_id','timestamp','item_id','pr

```

	session_id	timestamp	item_id	category
10	3	2014-04-02T13:17:46.940Z	21302	0
11	3	2014-04-02T13:26:02.515Z	25022	0
12	3	2014-04-02T13:30:12.318Z	29039	0
49	19	2014-04-01T20:52:12.357Z	5749	0
50	19	2014-04-01T20:52:13.758Z	5749	0

Since the data is quite large, we subsample it for easier demonstration.

```
#randomly sample a couple of them
sampled_session_id = np.random.choice(df.session_id.uni
df = df.loc[df.session_id.isin(sampled_session_id)]
df.nunique()
```

```
session_id     1000000
timestamp      5546275
item_id          37353
category           260
dtype: int64
```

Number of unique elements in the subsampled data

To determine the ground truth, i.e. whether there is any buy event for a given session, we simply check if a session_id in *yoochoose-clicks.dat* presents in *yoochoose-buys.dat* as well.

```
df['label'] = df.session_id.isin(buy_df.session_id)
df.head()
```

	session_id	timestamp	item_id	category	label
10	3	2014-04-02T13:17:46.940Z	21302	0	False
11	3	2014-04-02T13:26:02.515Z	25022	0	False
12	3	2014-04-02T13:30:12.318Z	29039	0	False
49	19	2014-04-01T20:52:12.357Z	5749	0	False
50	19	2014-04-01T20:52:13.758Z	5749	0	False

Dataset Construction

The data is ready to be transformed into a Dataset object after the preprocessing step. Here, we treat each item in a session as a node, and therefore all items in the same session form a graph. To build the dataset, we group the preprocessed data by *session_id* and iterate over these groups. In each iteration, the item_id in each group are categorically encoded again since for each graph, the node index should count from 0. Thus, we have the following:

```
import torch
from torch_geometric.data import InMemoryDataset
from tqdm import tqdm

class YooChooseBinaryDataset(InMemoryDataset):
    def __init__(self, root, transform=None, pre_trans
        super(YooChooseBinaryDataset, self).__init__(r
        self.data, self.slices = torch.load(self.proce

    @property
    def raw_file_names(self):
        return []
    @property
    def processed_file_names(self):
        return ['../input/yoochoose_click_binary_1M_se

    def download(self):
        pass

    def process(self):

        data_list = []

        # process by session_id
        grouped = df.groupby('session_id')
        for session_id, group in tqdm(grouped):
            sess_item_id = LabelEncoder().fit_transfor
            group = group.reset_index(drop=True)
            group['sess_item_id'] = sess_item_id
            node_features = group.loc[group.session_id
```

After building the dataset, we call *shuffle()* to make sure it has been randomly shuffled and then split it into three sets for training, validation, and testing.

```
dataset = dataset.shuffle()
train_dataset = dataset[:800000]
val_dataset = dataset[800000:900000]
test_dataset = dataset[900000:]
```

Build a Graph Neural Network

The following custom GNN takes reference from one of the examples in PyG's official Github repository. I changed the GraphConv layer with our self-implemented SAGEConv layer illustrated above. In addition, the output layer was also modified to match with a binary classification setup.

```
embed_dim = 128
from torch_geometric.nn import TopKPooling
from torch_geometric.nn import global_mean_pool as gap
import torch.nn.functional as F
class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()

        self.conv1 = SAGEConv(embed_dim, 128)
        self.pool1 = TopKPooling(128, ratio=0.8)
        self.conv2 = SAGEConv(128, 128)
        self.pool2 = TopKPooling(128, ratio=0.8)
        self.conv3 = SAGEConv(128, 128)
        self.pool3 = TopKPooling(128, ratio=0.8)
        self.item_embedding = torch.nn.Embedding(num_e
        self.lin1 = torch.nn.Linear(256, 128)
        self.lin2 = torch.nn.Linear(128, 64)
        self.lin3 = torch.nn.Linear(64, 1)
        self.bn1 = torch.nn.BatchNorm1d(128)
        self.bn2 = torch.nn.BatchNorm1d(64)
        self.act1 = torch.nn.ReLU()
        self.act2 = torch.nn.ReLU()

    def forward(self, data):
        x, edge_index, batch = data.x, data.edge_index
        x = self.item_embedding(x)
        x = x.squeeze(1)

        x = F.relu(self.conv1(x, edge_index))

        x, edge_index, _, batch, _ = self.pool1(x, edg
        x1 = torch.cat([gmp(x, batch), gap(x, batch)],

        x = F.relu(self.conv2(x, edge_index))

```

Training

Training our custom GNN is very easy, we simply iterate the DataLoader constructed from the training set and back-propagate the loss function. Here, we use Adam as the optimizer with the learning rate set to 0.005 and Binary Cross Entropy as the loss function.

```
def train():
    model.train()

    loss_all = 0
    for data in train_loader:
        data = data.to(device)
        optimizer.zero_grad()
        output = model(data)
        label = data.y.to(device)
        loss = crit(output, label)
        loss.backward()
        loss_all += data.num_graphs * loss.item()
        optimizer.step()
    return loss_all / len(train_dataset)

device = torch.device('cuda')
```

Validation

This label is highly unbalanced with an overwhelming amount of negative labels since most of the sessions are not followed by any buy event. In other words, a dumb model guessing all negatives would give you above 90% accuracy. Therefore, instead of accuracy, Area Under Curve (AUC) is a better metric for this task as it only cares if the positive examples are scored higher than the negative examples. We use the off-the-shelf AUC calculation function from Sklearn.

```
def evaluate(loader):
    model.eval()

    predictions = []
    labels = []

    with torch.no_grad():
        for data in loader:

            data = data.to(device)
            pred = model(data).detach().cpu().numpy()
```

Result

I trained the model for 1 epoch, and measure the training, validation, and testing AUC scores:

```
for epoch in range(1):
    loss = train()
    train_acc = evaluate(train_loader)
    val_acc = evaluate(val_loader)
    test_acc = evaluate(test_loader)
    print('Epoch: {:03d}, Loss: {:.5f}, Train Auc: {:.5
```

```
Epoch: 000, Loss: 0.28378, Train Auc: 0.76702, Val Auc: 0.73039, Test Auc: 0.72918
```

With only 1 Million rows of training data (around 10% of all data) and 1 epoch of training, we can obtain an AUC score of around 0.73 for validation and test set. The score is very likely to improve if more data is used to train the model with larger training steps.

Conclusion

You have learned the basic usage of PyTorch Geometric, including dataset construction, custom graph layer, and training GNNs with real-world data. All the code in this post can also be found in my Github repo, where you can find another Jupyter notebook file in which I solve the second task of the RecSys Challenge 2015. I hope you have enjoyed this article. Should you have any questions or comments, please leave it below! Make sure to follow me on twitter where I share my blog post or interesting Machine Learning/ Deep Learning news! Have fun playing GNN with PyG!