



♦ Member-only story

Hands-on Graph Neural Networks with PyTorch & PyTorch Geometric



Kung-Hsiang, Huang (Steeve) · Following

Published in Towards Data Science

9 min read · May 30, 2019

Listen

Share

More



PyTorch geometric

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. It is several times faster than the most well-known GNN framework, DGL.

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

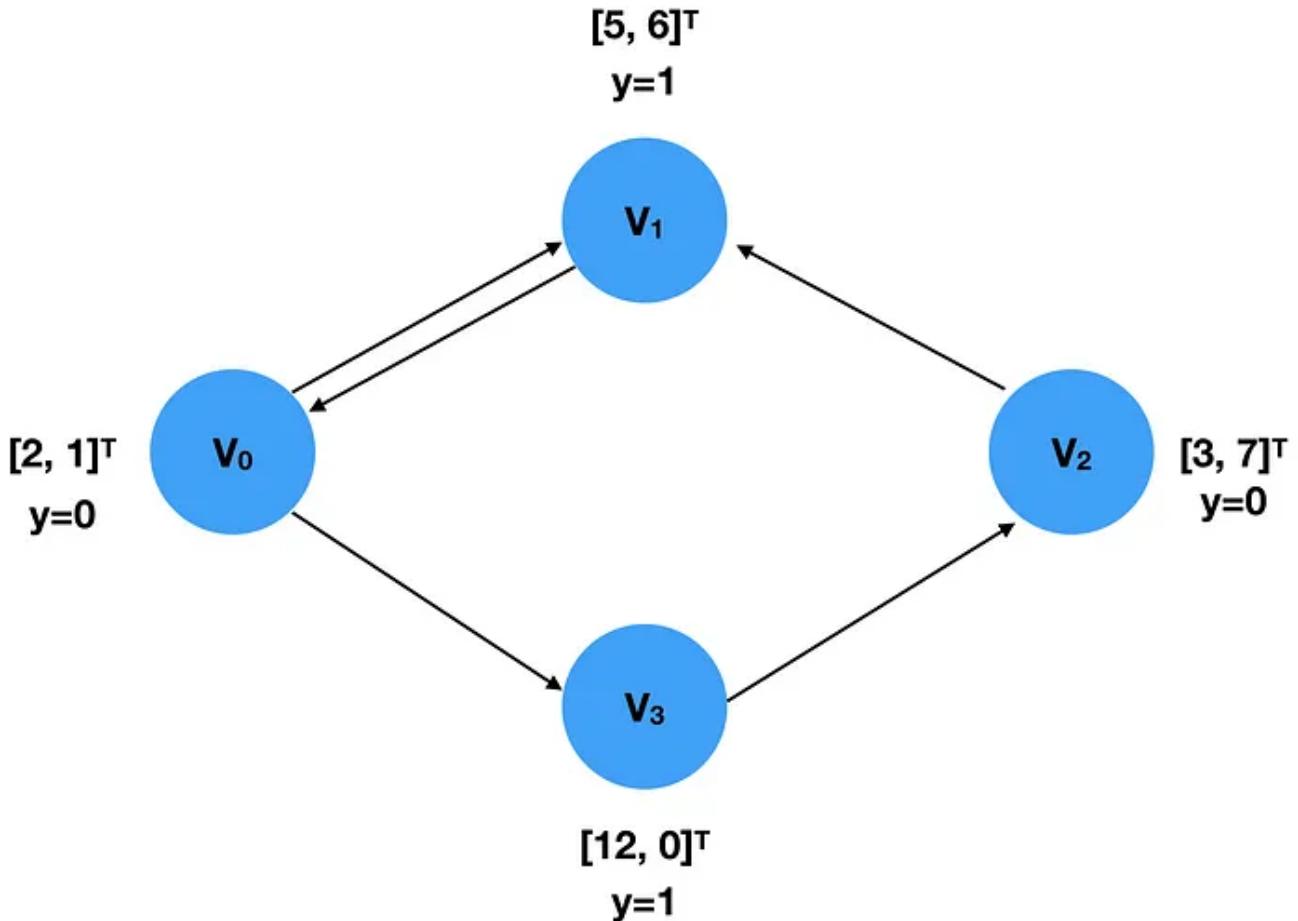
This section will walk you through the basics of PyG. Essentially, it will cover *torch_geometric.data* and *torch_geometric.nn*. You will learn how to pass geometric data into your GNN, and how to design a custom MessagePassing layer, the core of GNN.

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, $v_1 \dots v_4$, 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:

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

[gistfile1.txt](#) hosted with ❤ by GitHub

[view raw](#)

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.

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

[gistfile1.txt](#) hosted with ❤ by GitHub

[view raw](#)

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.

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

[gistfile1.txt](#) hosted with ❤ by GitHub

[view raw](#)

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

```
1 import torch  
2 from torch_geometric.data import Data  
3  
4  
5 x = torch.tensor([[2,1], [5,6], [3,7], [12,0]], dtype=torch.float)  
6 y = torch.tensor([0, 1, 0, 1], dtype=torch.float)  
7  
8 edge_index = torch.tensor([[0, 2, 1, 0, 3],  
9                               [3, 1, 0, 1, 2]], dtype=torch.long)  
10  
11  
12 data = Data(x=x, y=y, edge_index=edge_index)  
13 >>> Data(edge_index=[2, 5], x=[4, 2], y=[4])
```

[gistfile1.txt](#) hosted with ❤ by GitHub

[view raw](#)

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](#).

```

1 import torch
2 from torch_geometric.data import InMemoryDataset
3
4
5 class MyOwnDataset(InMemoryDataset):
6     def __init__(self, root, transform=None, pre_transform=None):
7         super(MyOwnDataset, self).__init__(root, transform, pre_transform)
8         self.data, self.slices = torch.load(self.processed_paths[0])
9
10    @property
11    def raw_file_names(self):
12        return ['some_file_1', 'some_file_2', ...]
13
14    @property
15    def processed_file_names(self):
16        return ['data.pt']
17
18    def download(self):
19        # Download to `self.raw_dir`.
20
21    def process(self):
22        # Read data into huge `Data` list.
23        data_list = [...]
24
25        if self.pre_filter is not None:
26            data_list [data for data in data_list if self.pre_filter(data)]
27
28        if self.pre_transform is not None:
29            data_list = [self.pre_transform(data) for data in data_list]
30
31        data, slices = self.collate(data_list)
32        torch.save((data, slices), self.processed_paths[0])

```

gistfile1.txt hosted with ❤ by GitHub

[view raw](#)

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:

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

<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:

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

<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.

```

1 import torch
2 from torch.nn import Sequential as Seq, Linear, ReLU
3 from torch_geometric.nn import MessagePassing
4 from torch_geometric.utils import remove_self_loops, add_self_loops
5 class SAGEConv(MessagePassing):
6     def __init__(self, in_channels, out_channels):
7         super(SAGEConv, self).__init__(aggr='max') # "Max" aggregation.
8         self.lin = torch.nn.Linear(in_channels, out_channels)
9         self.act = torch.nn.ReLU()
10        self.update_lin = torch.nn.Linear(in_channels + out_channels, in_channels, bias=True)
11        self.update_act = torch.nn.ReLU()
12
13    def forward(self, x, edge_index):
14        # x has shape [N, in_channels]
15        # edge_index has shape [2, E]
16
17
18        edge_index, _ = remove_self_loops(edge_index)
19        edge_index, _ = add_self_loops(edge_index, num_nodes=x.size(0))
20
21
22        return self.propagate(edge_index, size=(x.size(0), x.size(0)), x=x)
23
24    def message(self, x_j):
25        # x_j has shape [E, in_channels]
26
27        x_j = self.lin(x_j)
28        x_j = self.act(x_j)
29
30        return x_j
31
32    def update(self, aggr_out, x):
33        # aggr_out has shape [N, out_channels]
34
35
36        new_embedding = torch.cat([aggr_out, x], dim=1)
37
38        new_embedding = self.update_lin(new_embedding)
39        new_embedding = self.update_act(new_embedding)
40
41        return new_embedding

```

aistfile1.txt hosted with ❤ by GitHub

[view raw](#)

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.

```

1 from sklearn.preprocessing import LabelEncoder
2
3 df = pd.read_csv('../input/yoochoose-click.dat', header=None)
4 df.columns=['session_id','timestamp','item_id','category']
5
6 buy_df = pd.read_csv('../input/yoochoose-buys.dat', header=None)
7 buy_df.columns=['session_id','timestamp','item_id','price','quantity']
8
9 item_encoder = LabelEncoder()
10 df['item_id'] = item_encoder.fit_transform(df.item_id)
11 df.head()

```

	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.

```

1 #randomly sample a couple of them
2 sampled_session_id = np.random.choice(df.session_id.unique(), 1000000, replace=False)
3 df = df.loc[df.session_id.isin(sampled_session_id)]
4 df.nunique()

```

gistfile1.txt hosted with ❤ by GitHub

[view raw](#)

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.

```

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

```

gistfile1.txt hosted with ❤ by GitHub

[view raw](#)

	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:

```

1 import torch
2 from torch_geometric.data import InMemoryDataset
3 from tqdm import tqdm
4
5 class YooChooseBinaryDataset(InMemoryDataset):
6     def __init__(self, root, transform=None, pre_transform=None):
7         super(YooChooseBinaryDataset, self).__init__(root, transform, pre_transform)
8         self.data, self.slices = torch.load(self.processed_paths[0])
9
10    @property
11    def raw_file_names(self):
12        return []
13    @property
14    def processed_file_names(self):
15        return ['../input/yoochoose_click_binary_1M_sess.dataset']
16
17    def download(self):
18        pass
19
20    def process(self):
21
22        data_list = []
23
24        # process by session_id
25        grouped = df.groupby('session_id')
26        for session_id, group in tqdm(grouped):
27            sess_item_id = LabelEncoder().fit_transform(group.item_id)
28            group = group.reset_index(drop=True)
29            group['sess_item_id'] = sess_item_id
30            node_features = group.loc[group.session_id==session_id,['sess_item_id','item_id']]
31
32            node_features = torch.LongTensor(node_features).unsqueeze(1)
33            target_nodes = group.sess_item_id.values[1:]
34            source_nodes = group.sess_item_id.values[:-1]
35
36            edge_index = torch.tensor([source_nodes, target_nodes], dtype=torch.long)
37            x = node_features
38
39            y = torch.FloatTensor([group.label.values[0]])
40
41            data = Data(x=x, edge_index=edge_index, y=y)
42            data_list.append(data)
43
44        data, slices = self.collate(data_list)
45        torch.save((data, slices), self.processed_paths[0])
46

```

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.

```
1 dataset = dataset.shuffle()  
2 train_dataset = dataset[:800000]  
3 val_dataset = dataset[800000:900000]  
4 test_dataset = dataset[900000:]  
5 len(train_dataset), len(val_dataset), len(test_dataset)
```

gistfile1.txt hosted with ❤ by GitHub

[view raw](#)

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.

```

1  embed_dim = 128
2  from torch_geometric.nn import TopKPooling
3  from torch_geometric.nn import global_mean_pool as gmp, global_max_pool as gmp
4  import torch.nn.functional as F
5  class Net(torch.nn.Module):
6      def __init__(self):
7          super(Net, self).__init__()
8
9          self.conv1 = SAGEConv(embed_dim, 128)
10         self.pool1 = TopKPooling(128, ratio=0.8)
11         self.conv2 = SAGEConv(128, 128)
12         self.pool2 = TopKPooling(128, ratio=0.8)
13         self.conv3 = SAGEConv(128, 128)
14         self.pool3 = TopKPooling(128, ratio=0.8)
15         self.item_embedding = torch.nn.Embedding(num_embeddings=df.item_id.max() + 1, en
16         self.lin1 = torch.nn.Linear(256, 128)
17         self.lin2 = torch.nn.Linear(128, 64)
18         self.lin3 = torch.nn.Linear(64, 1)
19         self.bn1 = torch.nn.BatchNorm1d(128)
20         self.bn2 = torch.nn.BatchNorm1d(64)
21         self.act1 = torch.nn.ReLU()
22         self.act2 = torch.nn.ReLU()
23
24     def forward(self, data):
25         x, edge_index, batch = data.x, data.edge_index, data.batch
26         x = self.item_embedding(x)
27         x = x.squeeze(1)
28
29         x = F.relu(self.conv1(x, edge_index))
30
31         x, edge_index, _, batch, _ = self.pool1(x, edge_index, None, batch)
32         x1 = torch.cat([gmp(x, batch), gap(x, batch)], dim=1)
33
34         x = F.relu(self.conv2(x, edge_index))
35
36         x, edge_index, _, batch, _ = self.pool2(x, edge_index, None, batch)
37         x2 = torch.cat([gmp(x, batch), gap(x, batch)], dim=1)
38
39         x = F.relu(self.conv3(x, edge_index))
40
41         x, edge_index, _, batch, _ = self.pool3(x, edge_index, None, batch)
42         x3 = torch.cat([gmp(x, batch), gap(x, batch)], dim=1)
43
44         x = x1 + x2 + x3
45
46         x = self.lin1(x)
47         x = self.act1(x)
48         x = self.lin2(x)

```

```

46         ^ = self.lin2(^)
47         x = self.act2(x)
48         x = F.dropout(x, p=0.5, training=self.training)
49
50         x = torch.sigmoid(self.lin3(x)).squeeze(1)
51
52     return x

```

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.

```

1  def train():
2      model.train()
3
4      loss_all = 0
5      for data in train_loader:
6          data = data.to(device)
7          optimizer.zero_grad()
8          output = model(data)
9          label = data.y.to(device)
10         loss = crit(output, label)
11         loss.backward()
12         loss_all += data.num_graphs * loss.item()
13         optimizer.step()
14     return loss_all / len(train_dataset)
15
16 device = torch.device('cuda')
17 model = Net().to(device)
18 optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
19 crit = torch.nn.BCELoss()
20 train_loader = DataLoader(train_dataset, batch_size=batch_size)
21 for epoch in range(num_epochs):
22     train()
23

```

[gistfile1.txt](#) hosted with ❤ by GitHub

[view raw](#)

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.

```
1 def evaluate(loader):
2     model.eval()
3
4     predictions = []
5     labels = []
6
7     with torch.no_grad():
8         for data in loader:
9
10            data = data.to(device)
11            pred = model(data).detach().cpu().numpy()
12
13            label = data.y.detach().cpu().numpy()
14            predictions.append(pred)
15            labels.append(label)
```

[gistfile1.txt](#) hosted with ❤ by GitHub

[view raw](#)

Result

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

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

[gistfile1.txt](#) hosted with ❤ by GitHub

[view raw](#)

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!

Machine Learning

Data Science

Deep Learning

Artificial Intelligence



Following



Written by Kung-Hsiang, Huang (Steeve)

4K Followers · Writer for Towards Data Science

PhD student at UIUC, Co-Founder at [Rosetta.ai](#) | Prev: MSc at USC, BEng at HKUST | Twitter: https://twitter.com/steeve_huang

More from Kung-Hsiang, Huang (Steeve) and Towards Data Science



 Kung-Hsiang Huang (Steeve) in Towards Data Science

Introduction to Various Reinforcement Learning Algorithms. Part I (Q-Learning, SARSA, DQN, DDPG)

Reinforcement Learning (RL) refers to a kind of Machine Learning method in which the agent receives a delayed reward in the next time step...

★ · 9 min read · Jan 12, 2018

 5.7K  27



...



 Kenneth Leung in Towards Data Science

Running Llama 2 on CPU Inference Locally for Document Q&A

Clearly explained guide for running quantized open-source LLM applications on CPUs using LLama 2, C Transformers, GGML, and LangChain

★ · 11 min read · Jul 18

👏 1.5K

💬 22



...



 Matt Chapman in Towards Data Science

Stop Using PowerPoint for Your ML Presentations and Try This Instead

Gradio is a surefire way to impress both technical and non-technical stakeholders—Why aren't more Data Scientists and MLEs using it?

★ · 6 min read · Jul 3

👏 709

💬 10



...



 Kung-Hsiang, Huang (Steeve) in Towards Data Science

Word2Vec and FastText Word Embedding with Gensim

In Natural Language Processing (NLP), we often map words into vectors that contains numeric values so that machine can understand it. Word...

♦ · 7 min read · Feb 4, 2018

 3.3K

 20

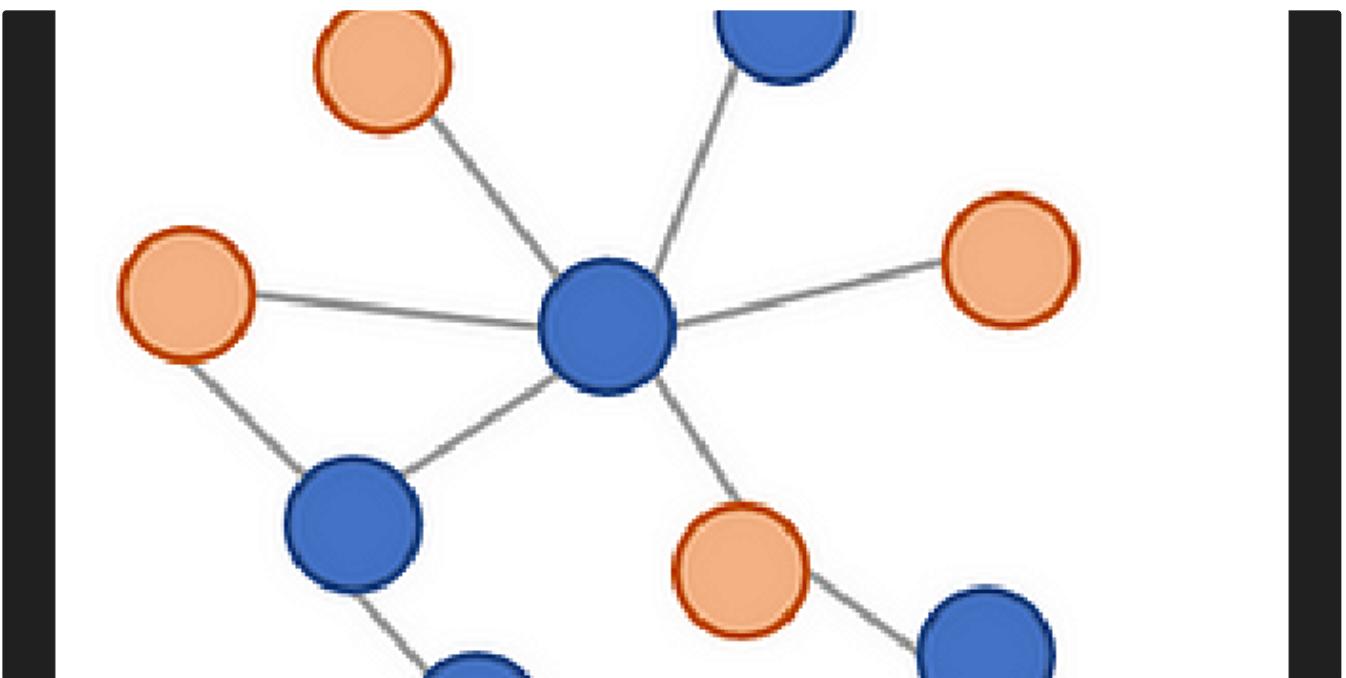


...

See all from Kung-Hsiang, Huang (Steeve)

See all from Towards Data Science

Recommended from Medium



Tejpal Kumawat

Application of Graph Neural Networks for Node Classification (on Cora Dataset)

We will classify the node class with the help of Graph Neural Network

5 min read · Mar 14

33

+

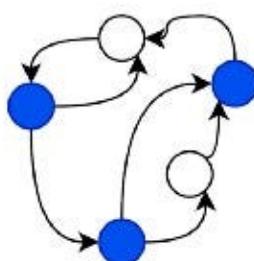
...

A graph, \mathcal{G} , dealing with a timeseries task
is a function of :

\mathcal{V} vertices

\mathcal{E} edges

$\mathcal{X}_v(t)$ } dynamic features



Barak Or, PhD in Towards AI

Traffic Forecasting: The Power of Graph Convolutional Networks on Time Series

The Graph Convolutional Network (GCN) has revolutionized the field of deep learning by showcasing its versatility in solving real-world...

5 min read · Feb 12

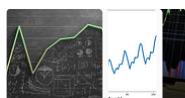
👏 76

💬 2



...

Lists



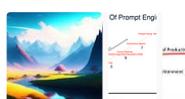
Predictive Modeling w/ Python

18 stories · 183 saves



Practical Guides to Machine Learning

10 stories · 198 saves



Natural Language Processing

459 stories · 87 saves



ChatGPT prompts

22 stories · 180 saves



Matthew Mo

Graph Isomorphism Neural Networks—Drug Discovery Example

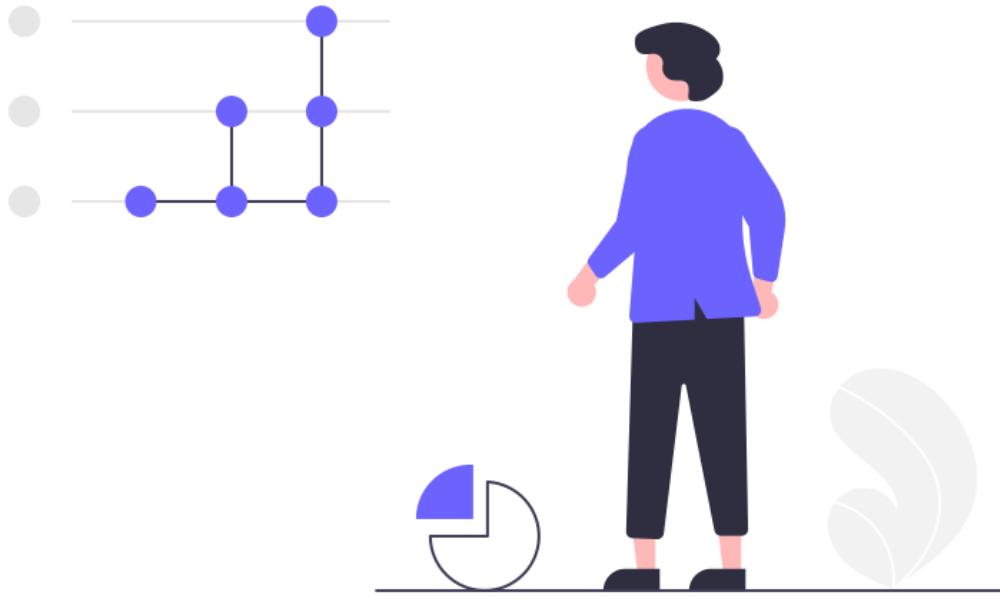
Molecular properties regressive modelling using Graph Isomorphism Networks

6 min read · Feb 27

14



...



E Ravenspike in Dev Genius

Analyzing graph networks: Utilizing advanced methods

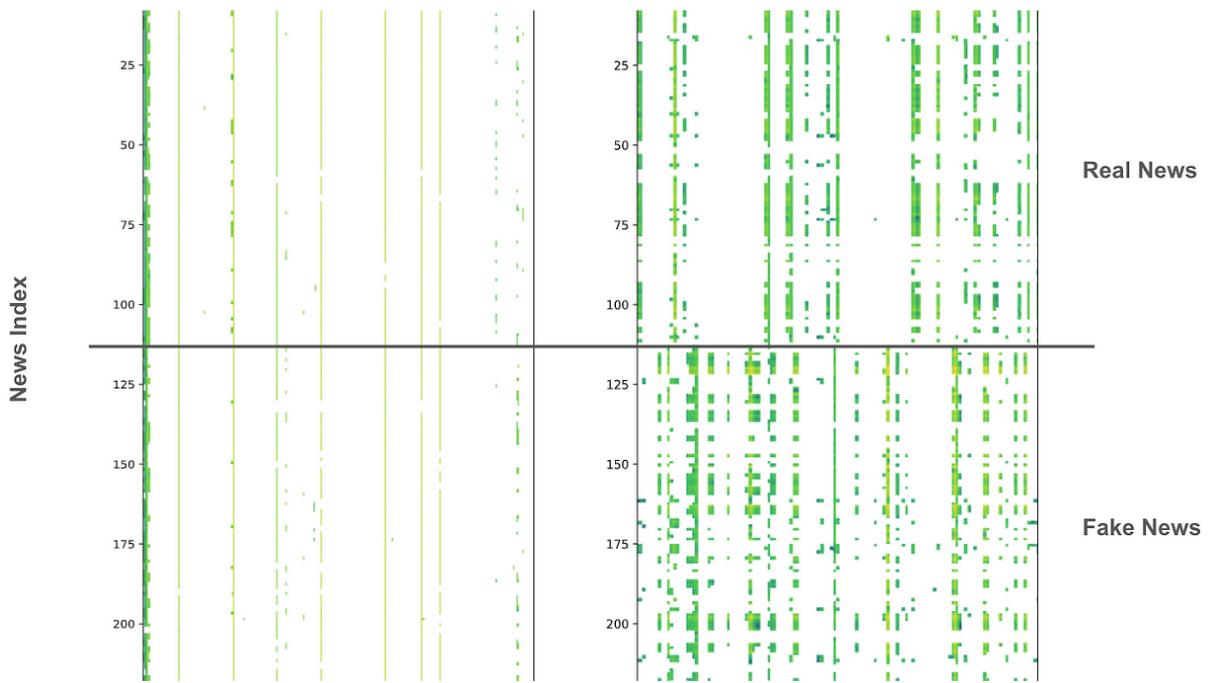
Story overview

16 min read · Jun 5

29



...



Sherry Wu in Stanford CS224W GraphML Tutorials

Spread No More: Twitter Fake News Detection with GNN

By Li Tian, Sherry Wu, Yifei Zheng as part of the Stanford CS224W course project.

11 min read · May 15

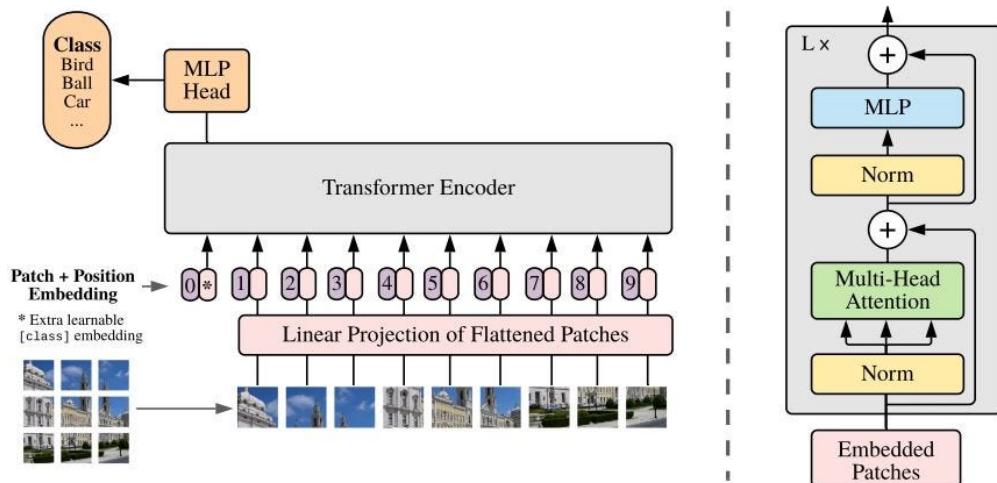


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable “classification token” to the sequence. The illustration of the Transformer encoder was inspired by

Fahim Rustamy, PhD

Vision Transformers vs. Convolutional Neural Networks

This blog post is inspired by the paper titled AN IMAGE IS WORTH 16×16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE from google's...

7 min read · Jun 4

👏 178

💬 3



...

[See more recommendations](#)