Coding Graph Neural Networks in Deep Graph Library

By Liyi Zhang, advisor: Adji Bousso Dieng

Table of Contents

- Graph Neural Network (GNN)
    - Quick intro to the message passing framework
- Deep Graph Library (DGL)
    - Background
    - DGLGraph() object
    - Node and edge features
    - Build a high level graph convolutional network (GCN)
    - Build customized graph attention network (GAT) and GCN
    - Logistics
        - Mini-batching, saving, and loading

Graph Neural Network
Message Passing Framework

Overall framework

- Takes several iterations.
- Each node has an embedding vector. The aim is to update node embeddings during each iterations, and use updated node embeddings to perform downstream tasks.
- Each iteration uses two modules: update and aggregate.

Notations

$H^{(0)}$, an N-by-d_o matrix for initial node embeddings, where N is number of nodes, and d_0 is initial embedding

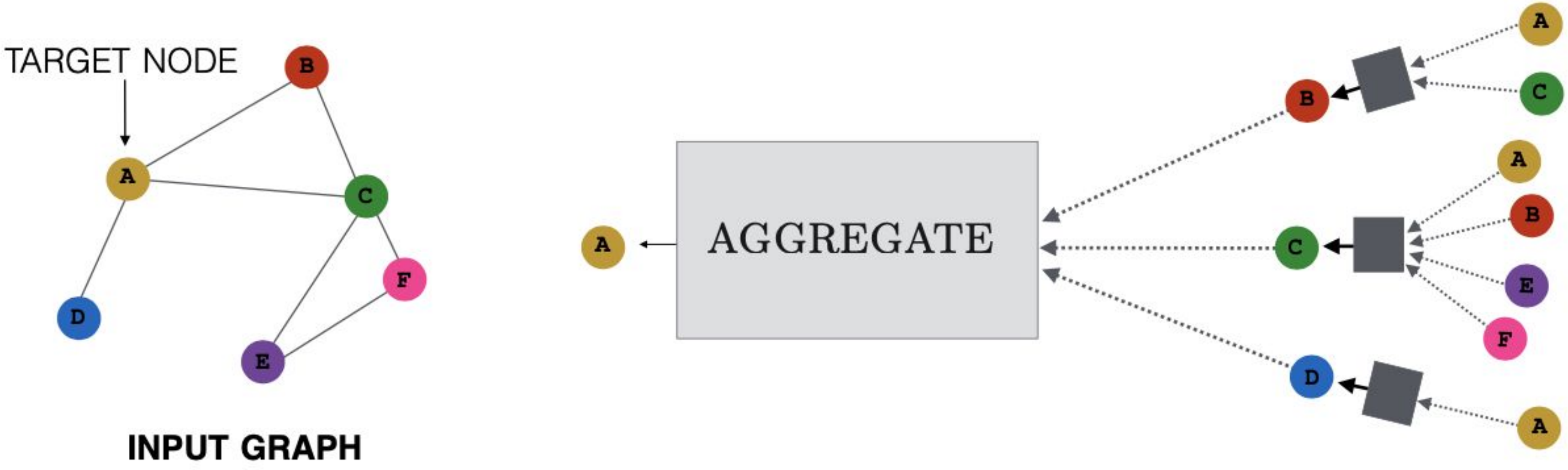
$h_u^{(k)}$: embedding of node u at iteration k

$\mathcal{N}(u)$: set of neighboring nodes of node u

$W^{(k)}, b^{(k)}$: weights and biases at iteration k

Graph Neural Network

Message Passing Framework



Graph Neural Network

Message Passing Framework

$$
\begin{aligned}
\mathbf{h}_u^{(k+1)} &= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right) \\
&= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right),
\end{aligned}
$$

$$
\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)} \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\right)
$$

Book: Hamilton (2020) - Graph Representation Learning
Paper: Gilmer, et al. (2017) - Neural Message Passing for Quantum Chemistry

Deep Graph Library (DGL)

Efficient and scalable

Framework agnostic

- Naturally incorporated into PyTorch, TensorFlow, and MXNet ecosystems.

Spring.2018

First Prototype

Designed by Prof. Zheng Zhang and Quan Gan at NYU Shanghai.

06.2018

Serious Development Began

When Minjie, Lingfan, and Prof. Jinyang Li from NYU's system group joined, flanked by a team of student volunteers at NYU Shanghai and Fudan.

09.2018

Development With Industry Team

With AWS MXNet Science team including Da Zheng, Alex Smola, Haibin Lin, Chao Ma and a number of others.

12.07.2018

V0.1 Release

First open release.

More Releases

‘Graph’ object

Conceptually

Graph = G(V, E), where V is the set of vertices; E is the set of edges. An edge is written as (u, v), with u, v ∈ V.

To represent edge connections, we can use an adjacency matrix A. A[u,v] = 1 if (u,v) ∈ E; 0 otherwise. A[u,u] = 1.

In DGL: DGLGraph()

```
g = dgl.graph([])
g
```

```
Graph(num_nodes=0, num_edges=0,
      ndata_schemes={}
      edata_schemes={})
```

DGLGraph()

dgl.graph(*data, ntype=None, etype=None, *, num_nodes=None, idtype=None, device=None, row_sorted=False, col_sorted=False, **deprecated_kwargs*) [source]

Create a graph and return.

Parameters:

- **data** (*graph data*) –
  The data for constructing a graph, which takes the form of (U, V). $(U[i], V[i])$ forms the edge with ID i in the graph. The allowed data formats are:

```
g = dgl.graph(([0, 0, 0, 0, 0, 5], [1, 2, 3, 4, 5, 6]), num_nodes=7)
g
```

✓ 0.1s

```
Graph(num_nodes=7, num_edges=6,
      ndata_schemes={}
      edata_schemes={})
```

DGLGraph()

dgl.graph(*data, ntype=None, etype=None, *, num_nodes=None, idtype=None, device=None, row_sorted=False, col_sorted=False, **deprecated_kwargs*) [source]

Create a graph and return.

Parameters:

- **data** (*graph data*) –
  The data for constructing a graph, which takes the form of (U, V). $(U[i], V[i])$ forms the edge with ID i in the graph. The allowed data formats are:

```
g = dgl.graph(([0, 0, 0, 0, 0, 5], [1, 2, 3, 4, 5, 6]), num_nodes=7)
g
```

✓ 0.1s

```
Graph(num_nodes=7, num_edges=6,
      ndata_schemes={}
      edata_schemes={})
```

Nodes?

DGLGraph()

dgl.graph(*data, ntype=None, etype=None, *, num_nodes=None, idtype=None, device=None, row_sorted=False, col_sorted=False, **deprecated_kwargs*) [source]

Create a graph and return.

Parameters:
- **data** (*graph data*) –
  The data for constructing a graph, which takes the form of (U, V). $(U[i], V[i])$ forms the edge with ID i in the graph. The allowed data formats are:

```
g = dgl.graph(([0, 0, 0, 0, 0, 5], [1, 2, 3, 4, 5, 6]), num_nodes=7)
g
```

✓ 0.1s

```
Graph(num_nodes=7, num_edges=6,
      ndata_schemes={}
      edata_schemes={})
```

Nodes
- 0,1,2,3,4,5,6

(indexing starts from 0)

DGLGraph()

dgl.graph(*data, ntype=None, etype=None, *, num_nodes=None, idtype=None, device=None, row_sorted=False, col_sorted=False, **deprecated_kwargs*) [source]

Create a graph and return.

Parameters:

- **data** (*graph data*) –
  The data for constructing a graph, which takes the form of (U, V). $(U[i], V[i])$ forms the edge with ID i in the graph. The allowed data formats are:

```
g = dgl.graph(([0, 0, 0, 0, 0, 5], [1, 2, 3, 4, 5, 6]), num_nodes=7)
g
```

✓ 0.1s

```
Graph(num_nodes=7, num_edges=6,
      ndata_schemes={}
      edata_schemes={})
```

Nodes

- 0,1,2,3,4,5,6

(indexing starts from 0)

Edges

DGLGraph()

dgl.graph(*data, ntype=None, etype=None, *, num_nodes=None, idtype=None, device=None, row_sorted=False, col_sorted=False, **deprecated_kwargs*) [source]

Create a graph and return.

Parameters:
- **data** (*graph data*) –
  The data for constructing a graph, which takes the form of (U, V). $(U[i], V[i])$ forms the edge with ID i in the graph. The allowed data formats are:

```
g = dgl.graph(([0, 0, 0, 0, 0, 5], [1, 2, 3, 4, 5, 6]), num_nodes=7)
g
✓ 0.1s

Graph(num_nodes=7, num_edges=6,
      ndata_schemes={}
      edata_schemes={})
```

Nodes
- 0,1,2,3,4,5,6

(indexing starts from 0)

Edges
- (0,1)
- (0,2)
- (0,3)
- (0,4)
- (0,5)
- (5,6)

DGLGraph()

Visualize with networkx

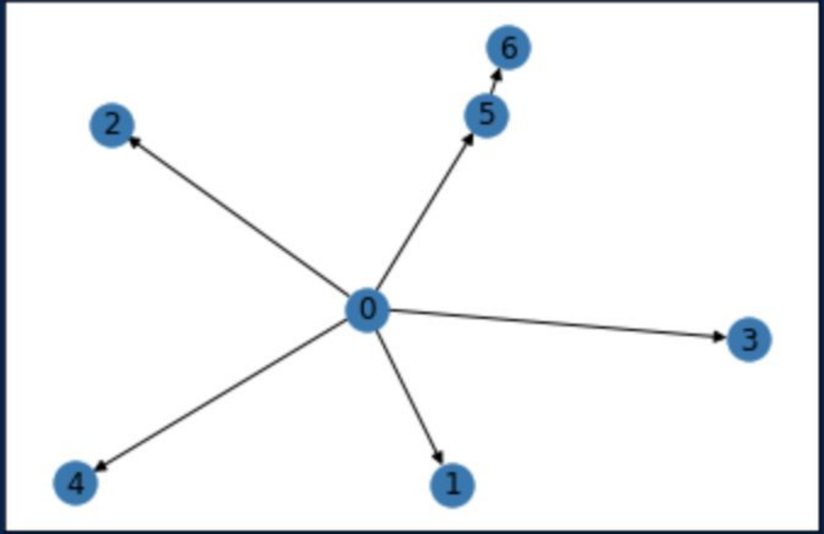
```
g = dgl.graph(([0, 0, 0, 0, 0, 5], [1, 2, 3, 4, 5, 6]), num_nodes=7)
g
```

✓ 0.1s

```
Graph(num_nodes=7, num_edges=6,
      ndata_schemes={}
      edata_schemes={})
```

```
G = dgl.to_networkx(g)
nx.draw_networkx(G)
```

✓ 0.5s



DGLGraph()

Visualize with networkx

We see that edges are directed. Often it's convenient to ignore direction. In code, it means giving each edge double direction.

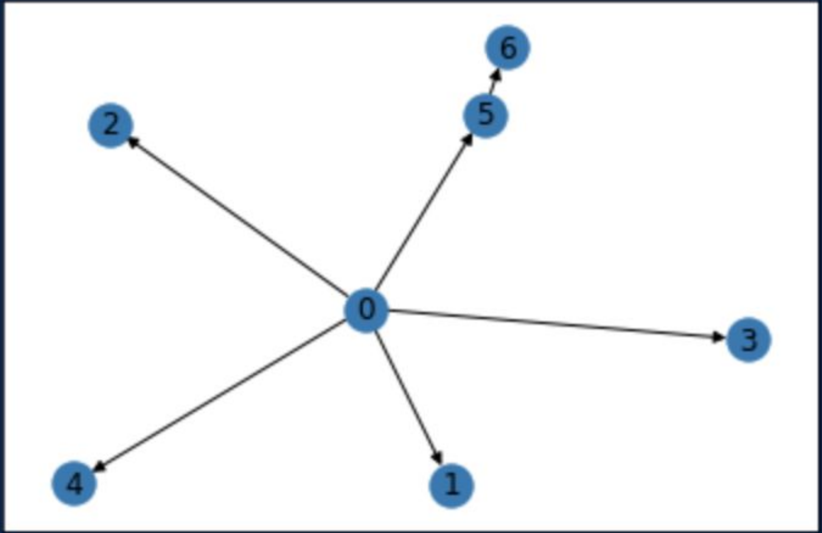
```
g = dgl.graph(([0, 0, 0, 0, 0, 5], [1, 2, 3, 4, 5, 6]), num_nodes=7)
g
```

✓ 0.1s

```
Graph(num_nodes=7, num_edges=6,
      ndata_schemes={}
      edata_schemes={})
```

```
G = dgl.to_networkx(g)
nx.draw_networkx(G)
```

✓ 0.5s

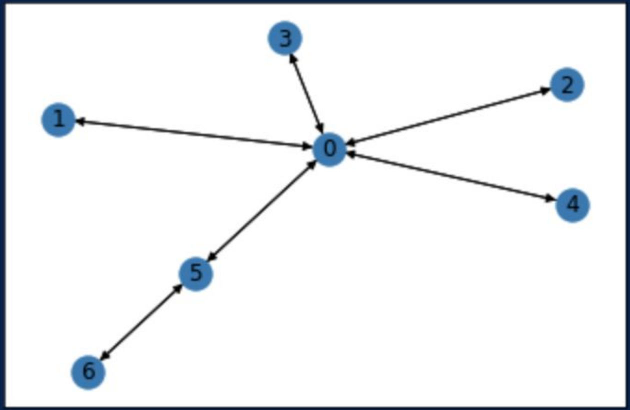


DGLGraph()

```
src_nodes = [0, 0, 0, 0, 0, 5]
dst_nodes = [1, 2, 3, 4, 5, 6]
src_nodes.extend(dst_nodes)
dst_nodes.extend(src_nodes[:len(dst_nodes)])
print(src_nodes)
print(dst_nodes)

g = dgl.graph((src_nodes, dst_nodes), num_nodes=7)
g

G = dgl.to_networkx(g)
nx.draw_networkx(G)
```

✓ 0.5s

```
[0, 0, 0, 0, 0, 5, 1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 0, 0, 0, 0, 0, 5]
```



Node and edge-related basic functions

```
# Access nodes and edges
print(g.nodes())
print(g.edges())
# Access number of nodes and edges
print(g.num_nodes())
print(g.num_edges())
```

✓ 0.1s

```
tensor([0, 1, 2, 3, 4, 5, 6])
(tensor([0, 0, 0, 0, 0, 5]), tensor([1, 2, 3, 4, 5, 6]))
7
6
```

Node and edge features

- Access through `g.ndata`, `g.edata`.
- Both are dictionaries.
  - There can be multiple types of features
  - Each feature's name is arbitrary
- The first dimension of each tensor, i.e. g.ndata[key].shape[0], should equal the number of nodes (or number of edges for g.edata).

```
g.ndata['h'] = torch.ones(g.num_nodes(), 3)
g.edata['e'] = torch.ones(g.num_edges(), 1)

print(g.ndata)
print(g.edata)
```

✓ 0.1s

```
{'h': tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])}
{'e': tensor([[1.],
        [1.],
        [1.],
        [1.],
        [1.],
        [1.]])}
```

High-Level Functions for GNN

A whole message-passing layer (iteration)

GraphConv

class **dgl.nn.pytorch.conv.GraphConv**(*in_feats, out_feats, norm='both', weight=True, bias=True, activation=None, allow_zero_in_degree=False*) [source]

Bases: `torch.nn.modules.module.Module`

Graph convolutional layer from Semi-Supervised Classification with Graph Convolutional Networks

Mathematically it is defined as follows:

$$h_i^{(l+1)} = \sigma(b^{(l)} + \sum_{j\in\mathcal{N}(i)} \frac{1}{c_{ji}} h_j^{(l)} W^{(l)})$$

where $\mathcal{N}(i)$ is the set of neighbors of node i, c_{ji} is the product of the square root of node degrees (i.e., $c_{ji} = \sqrt{|\mathcal{N}(j)|}\sqrt{|\mathcal{N}(i)|}$), and σ is an activation function.

Paper: Kipf and Welling (2016) - Semi-Supervised Classification with Graph Convolutional Networks

```python
class GNN(torch.nn.Module):
    '''
    GNN model. Wraps together several GNN layers.

    Arguments:
    - in_dim: input node dimension
    - list_h_dim: list of node embedding dimension across layers

    Forward:
    - takes a graph (or batched graph using dgl.batch())
    - returns node embedding H, a num-nodes by h_dim[-1] matrix
    '''

    def __init__(self, in_dim, list_h_dim):

        super(GNN, self).__init__()

        self.num_layers = len(list_h_dim)
        self.gnn_layers = torch.nn.ModuleList()

        for i in range(self.num_layers):
            if i == 0:
                start_dim = in_dim
            else:
                start_dim = list_h_dim[i-1]
            self.gnn_layers.append(GraphConv(start_dim, list_h_dim[i]))

    def forward(self, g):

        with g.local_scope():

            for gnn_layer in self.gnn_layers:
                g.ndata['h'] = gnn_layer(g, g.ndata['h'])

                print(g.ndata['h'].shape)

            h_out = g.ndata['h']

            return h_out
```

Initializations

For each layer in all layers, append one GraphConv layer

Forward function

```python
class GNN(torch.nn.Module):
    '''
    GNN model. Wraps together several GNN layers.

    Arguments:
    - in_dim: input node dimension
    - list_h_dim: list of node embedding dimension across layers

    Forward:
    - takes a graph (or batched graph using dgl.batch())
    - returns node embedding H, a num-nodes by h_dim[-1] matrix
    '''

    def __init__(self, in_dim, list_h_dim):

        super(GNN, self).__init__()

        self.num_layers = len(list_h_dim)
        self.gnn_layers = torch.nn.ModuleList()

        for i in range(self.num_layers):
            if i == 0:
                start_dim = in_dim
            else:
                start_dim = list_h_dim[i-1]
            self.gnn_layers.append(GraphConv(start_dim, list_h_dim[i]))

    def forward(self, g):

        with g.local_scope():

            for gnn_layer in self.gnn_layers:
                g.ndata['h'] = gnn_layer(g, g.ndata['h'])

                print(g.ndata['h'].shape)

            h_out = g.ndata['h']

            return h_out
```

Run the model

```python
model = GNN(3, [5,10,15])
out = model(g)
```

✓ 0.7s

```
torch.Size([7, 5])
torch.Size([7, 10])
torch.Size([7, 15])
```

Attempt to reproduce output value

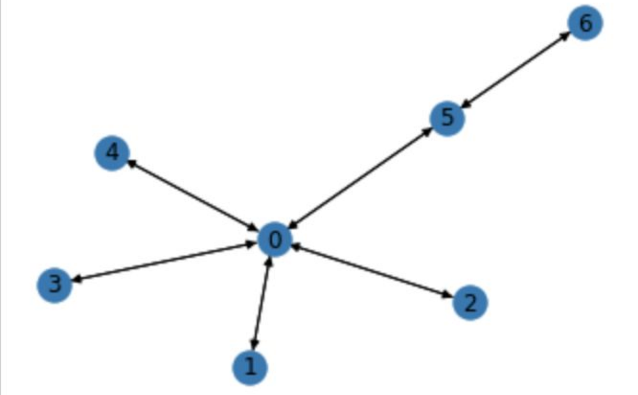
1. Instantiate a one-layer GNN
2. Fix weights to be all 1's, and no bias
3. Run the model and print output

```python
# Instantiate model
model = GNN(3, [5])

# Fix weights
with torch.no_grad():
    for i, param in enumerate(model.parameters()):
        if i == 0:
            param.copy_(torch.ones(3, 5))
        elif i == 1:
            param.copy_(torch.zeros(5))
        else:
            break

# Run the model (printing the output shape by the way)
print(model(g))
```

✓ 0.1s

```
torch.Size([7, 5])
tensor([[6.3152, 6.3152, 6.3152, 6.3152, 6.3152],
        [1.3416, 1.3416, 1.3416, 1.3416, 1.3416],
        [1.3416, 1.3416, 1.3416, 1.3416, 1.3416],
        [1.3416, 1.3416, 1.3416, 1.3416, 1.3416],
        [1.3416, 1.3416, 1.3416, 1.3416, 1.3416],
        [3.0700, 3.0700, 3.0700, 3.0700, 3.0700],
        [2.1213, 2.1213, 2.1213, 2.1213, 2.1213]], grad_fn=<AddBackward0>)
```

Attempt to reproduce output value



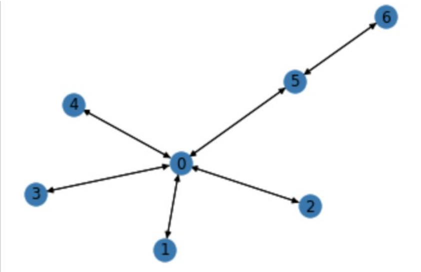

```
[[6.3152, 6.3152, 6.3152, 6.3152, 6.3152],
 [1.3416, 1.3416, 1.3416, 1.3416, 1.3416],
 [1.3416, 1.3416, 1.3416, 1.3416, 1.3416],
 [1.3416, 1.3416, 1.3416, 1.3416, 1.3416],
 [1.3416, 1.3416, 1.3416, 1.3416, 1.3416],
 [3.0700, 3.0700, 3.0700, 3.0700, 3.0700],
 [2.1213, 2.1213, 2.1213, 2.1213, 2.1213]]
```

```python
# Node 1
print(torch.matmul(g.ndata['h'][[0],], torch.ones(3, 5)) \
    / (1**0.5 * 5**0.5))

# Node 0
print(torch.matmul(g.ndata['h'][[1],], torch.ones(3, 5)) \
    / (1**0.5 * 5**0.5) * 4 + \
      torch.matmul(g.ndata['h'][[5],], torch.ones(3, 5)) \
    / (2**0.5 * 5**0.5))
```

✓ 0.1s

```
tensor([[1.3416, 1.3416, 1.3416, 1.3416, 1.3416]])
tensor([[6.3152, 6.3152, 6.3152, 6.3152, 6.3152]])
```

Attempt to reproduce output value




```
[[6.3152, 6.3152, 6.3152, 6.3152, 6.3152],
 [1.3416, 1.3416, 1.3416, 1.3416, 1.3416],
 [1.3416, 1.3416, 1.3416, 1.3416, 1.3416],
 [1.3416, 1.3416, 1.3416, 1.3416, 1.3416],
 [1.3416, 1.3416, 1.3416, 1.3416, 1.3416],
 [3.0700, 3.0700, 3.0700, 3.0700, 3.0700],
 [2.1213, 2.1213, 2.1213, 2.1213, 2.1213]]
```

```python
# Node 1
print(torch.matmul(g.ndata['h'][[0],], torch.ones(3, 5)) \
    / (1**0.5 * 5**0.5))

# Node 0
print(torch.matmul(g.ndata['h'][[1],], torch.ones(3, 5)) \
    / (1**0.5 * 5**0.5) * 4 + \
      torch.matmul(g.ndata['h'][[5],], torch.ones(3, 5)) \
    / (2**0.5 * 5**0.5))
```

✓ 0.1s

```
tensor([[1.3416, 1.3416, 1.3416, 1.3416, 1.3416]])
tensor([[6.3152, 6.3152, 6.3152, 6.3152, 6.3152]])
```

Note:

- GraphConv normalizes via GCN by default (unless norm is otherwise specified).
- If no self-to-self edge is included, the update function for node i does not include node i itself.

Low-Level Functions for GNN

An addition to message-passing: add attention weights to each node, and attention weights depend on edge information (Graph Attention Network (GAT)).

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_j]\right)\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_k]\right)\right)}$$

$$\vec{h}'_i = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}_i}\alpha_{ij}^k\mathbf{W}^k\vec{h}_j\right)$$

Paper: Velickovic, et al. (2017) - Graph Attention Networks

apply_edges()

dgl.DGLGraph.apply_edges

`DGLGraph.apply_edges`(*func='default', edges='__ALL__', inplace=False*) [source]

Apply the function on the edges to update their features.

If None is provided for `func`, nothing will happen.

Parameters:

- **func** (*callable, optional*) – Apply function on the edge. The function should be an `Edge UDF`.
- **edges** (*valid edges type, optional*) – Edges on which to apply `func`. See `send()` for valid edges type. Default is all the edges.
- **inplace** (*bool, optional*) – If True, update will be done in place, but autograd will break.

apply_edges()

```
def double_value(edges):
    return {'e': edges.data['e'] * 2}

g.apply_edges(func=double_value, edges=[0,1])

g.edata['e'][:5]
```

✓ 0.1s

```
tensor([[2.],
        [2.],
        [1.],
        [1.],
        [1.]])
```

A user-defined function always take `edges` object.

Here double_value simply doubles the value of the input.

update_all()

dgl.DGLGraph.update_all

`DGLGraph.update_all`*(message_func, reduce_func, apply_node_func=None, etype=None)*

Send messages along all the edges of the specified type and update all the nodes of the corresponding destination type.

Parameters:

- **message_func** (*dgl.function.BuiltinFunction or callable*) – The message function to generate messages along the edges. It must be either a DGL Built-in Function or a User-defined Functions.
- **reduce_func** (*dgl.function.BuiltinFunction or callable*) – The reduce function to aggregate the messages. It must be either a DGL Built-in Function or a User-defined Functions.
- **apply_node_func** (*callable, optional*) – An optional apply function to further update the node features after the message reduction. It must be a User-defined Functions.
- **etype** (*str or (str, str, str), optional*) –
  The type name of the edges. The allowed type name formats are:
  - `(str, str, str)` for source node type, edge type and destination node type.
  - or one `str` edge type name if the name can uniquely identify a triplet format in the graph.

  Can be omitted if the graph has only one type of edges.

update_all()

```
g.update_all(dgl.function.src_mul_edge('h', 'e', 'u'), dgl.function.sum('u', 'a'))
g.ndata['a']
```

✓ 0.9s

```
tensor([[5., 5., 5.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [2., 2., 2.],
        [1., 1., 1.]])
```

Putting It Together

```python
class GAT(torch.nn.Module):

    def __init__(self, h_dim, e_dim, h_out_dim):

        super(GAT, self).__init__()

        self.project_edge = torch.nn.Sequential(
            torch.nn.Linear(h_dim*2 + e_dim, 1),
            torch.nn.LeakyReLU()
        )
        self.transform_node = torch.nn.Linear(h_dim, h_out_dim)
        self.gru = torch.nn.GRUCell(h_out_dim, h_out_dim)

    def forward(self, g):

        with g.local_scope():

            g.apply_edges(lambda edges: {'e_t': torch.cat([edges.src['h'], edges.dst['h'], edges.data['e']], dim=1)})

            logits = self.project_edge(g.edata['e_t'])
            g.edata['alpha'] = dgl.nn.functional.edge_softmax(g, logits)

            g.ndata['h_t'] = self.transform_node(g.ndata['h'])
            g.update_all(dgl.function.src_mul_edge('h_t', 'alpha', 'u'), dgl.function.sum('u', 'a'))

            message = torch.nn.functional.elu(g.ndata['a'])
            h_out = torch.nn.functional.relu(self.gru(message, g.ndata['h_t']))

            return h_out
```

✓ 0.1s

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_j]\right)\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_k]\right)\right)}$$

$$\vec{h}'_i = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}_i}\alpha_{ij}^k \mathbf{W}^k \vec{h}_j\right)$$

No Attention - Vanilla Message-Passing

```
class GAT(torch.nn.Module):

    def __init__(self, h_dim, e_dim, h_out_dim):

        super(GAT, self).__init__()

        self.project_edge = torch.nn.Sequential(
            torch.nn.Linear(h_dim*2 + e_dim, 1),
            torch.nn.LeakyReLU()
        )
        self.transform_node = torch.nn.Linear(h_dim, h_out_dim)
        self.gru = torch.nn.GRUCell(h_out_dim, h_out_dim)

    def forward(self, g):

        with g.local_scope():

            g.apply_edges(lambda edges: {'e_t': torch.cat([edges.src['h'], edges.dst['h'], edges.data['e']], dim=1)})

            logits = self.project_edge(g.edata['e_t'])
            g.edata['alpha'] = dgl.nn.functional.edge_softmax(g, logits)

            g.ndata['h_t'] = self.transform_node(g.ndata['h'])
            g.update_all(dgl.function.copy_src('h_t', 'u'), dgl.function.sum('u', 'a'))
            # g.update_all(dgl.function.src_mul_edge('h_t', 'alpha', 'u'), dgl.function.sum('u', 'a'))

            message = torch.nn.functional.elu(g.ndata['a'])
            h_out = torch.nn.functional.relu(self.gru(message, g.ndata['h_t']))

            return h_out
```

✓ 0.1s

Logistics - Minibatches

dgl.batch

`dgl.batch`*(graph_list, node_attrs='__ALL__', edge_attrs='__ALL__')* [source]

Batch a collection of `DGLGraph` and return a `BatchedDGLGraph` object that is independent of the `graph_list`.

Parameters:

- **graph_list** (*iterable*) – A collection of `DGLGraph` to be batched.
- **node_attrs** (*None*, *str or iterable*) – The node attributes to be batched. If `None`, the `BatchedDGLGraph` object will not have any node attributes. By default, all node attributes will be batched. If `str` or iterable, this should specify exactly what node attributes to be batched.
- **edge_attrs** (*None*, *str or iterable, optional*) – Same as for the case of `node_attrs`

Returns: one single batched graph

```
g1 = dgl.graph(([0, 0, 0, 0, 0, 5], [1, 2, 3, 4, 5, 6]), num_nodes=7)
g2 = dgl.graph(([0, 0, 0, 0, 0], [1, 2, 3, 4, 5]), num_nodes=6)

dgl.batch([g1, g2])
```

✓ 0.1s

```
Graph(num_nodes=13, num_edges=11,
      ndata_schemes={}
      edata_schemes={})
```

Logistics - Save and Load

```
from dgl.data.utils import save_graphs, load_graphs

graph_labels = torch.tensor([0,1])
label_dict = {'glabel':torch.tensor(graph_labels)}

save_graphs('graphs.bin', [g1, g2], label_dict)
g_list, label = load_graphs('graphs.bin')
```

✓ 0.1s

Note:

- Save and load a *list* of graphs.
- Label is a dictionary. The value length must be same as number of graphs.
- If you save a batched graph, it cannot be unbatched after loading.
    - Might want to save number of nodes of each individual graph as well.

Thank you!

Q & A