Introduction to DGL

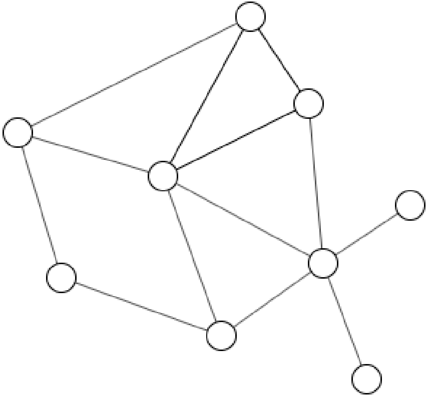
Quan Gan

AWS Shanghai AI Lab

October 14, 2019

Recap: Graph Neural Networks

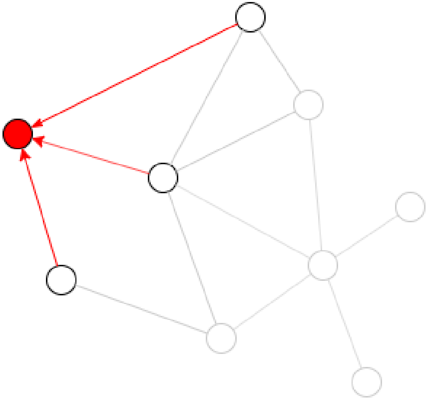
$$h_v^{(k)} = \phi\left(h_v^{(k-1)}, h_{\mathcal{N}(v)}^{(k)}\right) \qquad h_{\mathcal{N}(v)}^{(k)} = f\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)$$ [1]



[1] Xu et al., *How Powerful Are Graph Neural Networks?*, ICLR 2019

Recap: Graph Neural Networks

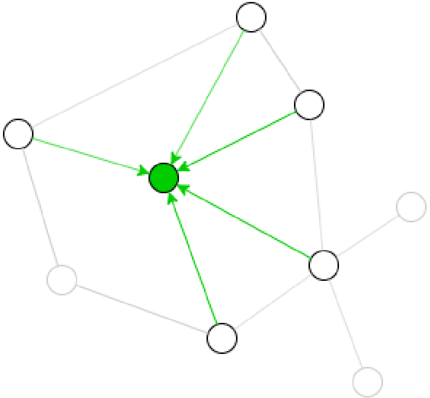
$$h_v^{(k)} = \phi\left(h_v^{(k-1)}, h_{\mathcal{N}(v)}^{(k)}\right) \qquad h_{\mathcal{N}(v)}^{(k)} = f\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)^{1}$$



[1] Xu et al., *How Powerful Are Graph Neural Networks?*, ICLR 2019

Recap: Graph Neural Networks

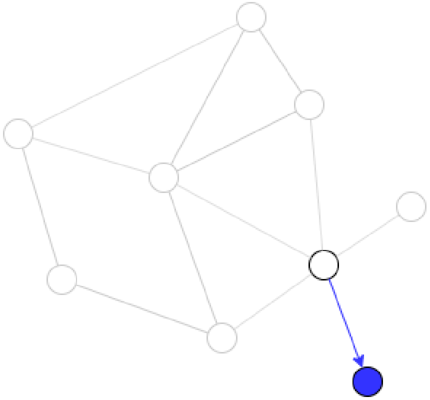
$$h_v^{(k)} = \phi\left(h_v^{(k-1)}, h_{\mathcal{N}(v)}^{(k)}\right) \qquad h_{\mathcal{N}(v)}^{(k)} = f\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)^{1}$$



[1] Xu et al., *How Powerful Are Graph Neural Networks?*, ICLR 2019

Recap: Graph Neural Networks

$$h_v^{(k)} = \phi\left(h_v^{(k-1)}, h_{\mathcal{N}(v)}^{(k)}\right) \qquad h_{\mathcal{N}(v)}^{(k)} = f\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)$$ [1]



[1] Xu et al., *How Powerful Are Graph Neural Networks?*, ICLR 2019

A common case[2]

If $f(\cdot)$ is average:

$$h_{\mathcal{N}(v)}^{(k)} = \frac{1}{|\mathcal{N}(v)|} \sum_{u \in \mathcal{N}(v)} h_u^{(k-1)}$$

$$h_v^{(k)} = \sigma\left(W^{(k)}\left[h_v^{(k-1)} \| h_{\mathcal{N}(v)}^{(k)}\right]\right)$$

Sparse matrix multiplication, easy:

```
# code: PyTorch
# src: edge source node IDs (n_nodes,)
# dst: edge destination node IDs (n_nodes,)
# H: node repr matrix (n_nodes, in_dim)
# W: weights (in_dim * 2, out_dim)
A = torch.sparse_coo_tensor(
    torch.stack([dst, src], 0),
    torch.ones(n_nodes),
    (n_nodes, n_nodes))
in_deg = torch.sparse.sum(A, 1).to_dense()
H_N = A @ H / in_deg.unsqueeze(1)
H = torch.relu(torch.cat([H_N, H], 1) @ W)
```

[2] Hamilton et al., *Inductive Representation Learning on Large Graphs*, NIPS 2017

How about max pooling?

$$h_{\mathcal{N}(v)}^{(k)} = \max_{u \in \mathcal{N}(v)} h_u^{(k-1)}$$

$$h_v^{(k)} = \sigma\left(W^{(k)}\left[h_v^{(k-1)} \| h_{\mathcal{N}(v)}^{(k)}\right]\right)$$

Only Tensorflow supports what we need natively:

```
# code: Tensorflow 2
# src: edge source node IDs (n_nodes,)
# dst: edge destination node IDs (n_nodes,)
# H: node repr matrix (n_nodes, in_dim)
# W: weights (in_dim * 2, out_dim)

# Broadcast source features to edges
H_src = tf.gather(H, src)
H_N = tf.math.unsorted_segment_max(
    H_src, dst, n_nodes)
H = tf.nn.relu(tf.concat([H_N, H], 1) @ W)
```

With attention?[3]

If $f(\cdot)$ is a *weighted* summation:

$$\hat{\alpha}_{v,u}^{(k-1)} = MLP\left(\left[h_v^{(k-1)} \| h_u^{(k-1)}\right]\right)$$

$$\alpha_{v,u}^{(k-1)} = softmax_j\left(\hat{\alpha}_{v,u}^{(k-1)}\right)$$

$$h_{\mathcal{N}(v)}^{(k)} = \sum_{u \in \mathcal{N}(v)} \alpha_{v,u}^{(k-1)} h_u^{(k-1)}$$

$$h_v^{(k)} = \sigma\left(W^{(k)}\left[h_v^{(k-1)}; h_{\mathcal{N}(v)}^{(k)}\right]\right)$$

Can't do it easily with vanilla PyTorch/MXNet.
Possible in Tensorflow

```
# code: Tensorflow 2
# src: edge source node IDs (n_nodes,)
# dst: edge destination node IDs (n_nodes,)
# H: node repr matrix (n_nodes, in_dim)
# W: weights (in_dim * 2, out_dim)
H_src = tf.gather(H, src)
H_dst = tf.gather(H, dst)
alpha_hat = MLP(tf.concat([H_dst, H_src], 1))
alpha_hat_sp = tf.sparse.SparseTensor(
    tf.stack([dst, src], 1),
    alpha_hat,
    (n_nodes, n_nodes))
alpha = tf.sparse.softmax(alpha_hat_sp)
H_N = tf.sparse.sparse_dense_matmul(alpha, H)
H = tf.nn.relu(tf.concat([H_N, H], 1) @ W)
```

[3] Velickovic et al., *Graph Attention Networks*, ICLR 2018

How about LSTM[4][5]?

If $f(\cdot)$ is summation:

$$h_{\mathcal{N}(v)}^{(k)} = LSTM(h_{u_1}^{(k-1)}, \ldots, h_{u_n}^{(k-1)})$$

where $u_i \in \mathcal{N}(v)$ are in some order

$$h_v^{(k)} = \sigma\left(W^{(k)}\left[h_v^{(k-1)} \| h_{\mathcal{N}(v)}^{(k)}\right]\right)$$

Very complicated:

```
# code: PyTorch
# src: edge source node IDs (n_nodes,)
# dst: edge destination node IDs (n_nodes,)
# t: timestamp of edges.
#    LSTM will go through messages in the order
#    of timestamps
# H: node repr matrix (n_nodes, in_dim)
# lstm: LSTM module
# W: weights (in_dim * 2, out_dim)
from torch.nn.utils.rnn import pack_sequence
# Build adjacency list
adjlist = []
for v in range(10):
    v_mask = (dst == v)
    t_v = t[v_mask]
    N_v = src[v_mask]
    indices = t_v.argsort()
    adjlist.append(N_v[indices])
# Pack input sequence
seqs = [H[u] for u in adjlist]
packed_seq = pack_sequence(seqs, False)
# Run LSTM and compute the new H
_, (H_N, _) = lstm(packed_seq)
H = torch.relu(torch.cat([H_N, H], 1) @ W)
```

[4] Fan et al., *Metapath-guided Heterogeneous Graph Neural Network for Intent Recommendation*, KDD 2019

[5] Zhang et al., *HetGNN: Heterogeneous Graph Neural Network*, KDD 2019

- So for each aggregation and message computation we have to write different code.
- In contrast, let's see how DGL handles the four cases.

A common case

If $f(\cdot)$ is average:

$$h_{\mathcal{N}(v)}^{(k)} = \frac{1}{|\mathcal{N}(v)|} \sum_{u \in \mathcal{N}(v)} h_u^{(k-1)}$$

$$h_v^{(k)} = \sigma\left(W^{(k)}\left[h_v^{(k-1)} \| h_{\mathcal{N}(v)}^{(k)}\right]\right)$$

```
# code: PyTorch
# src: edge source node IDs (n_nodes,)
# dst: edge destination node IDs (n_nodes,)
# H: node repr matrix (n_nodes, in_dim)
# W: weights (in_dim * 2, out_dim)
A = torch.sparse_coo_tensor(
    torch.stack([dst, src], 0),
    torch.ones(n_nodes),
    (n_nodes, n_nodes))
in_deg = torch.sparse.sum(A, 1).to_dense()
H_N = A @ H / in_deg.unsqueeze(1)
H = torch.relu(torch.cat([H_N, H], 1) @ W)
```

```
# code: PyTorch + DGL
# G: DGL Graph
# H: node repr matrix (n_nodes, in_dim)
# W: weights (in_dim * 2, out_dim)
import dgl.function as fn
G.ndata['h'] = H
G.update_all(fn.copy_u('h', 'm'), fn.mean('m', 'h_n'))
H_N = G.ndata['h_n']
H = torch.relu(torch.cat([H_N, H], 1) @ W)
```

```
# code: PyTorch + DGL
# G: DGL Graph
# H: node repr matrix (n_nodes, in_dim)
# For popular models we also have PyTorch/MXNet NN Modules:
from dgl.nn.pytorch import SAGEConv
conv = SAGEConv(in_dim * 2, out_dim, 'mean')
H = conv(G, H)
```

How about max pooling?

$$h_{\mathcal{N}(v)}^{(k)} = \max_{u \in \mathcal{N}(v)} h_u^{(k-1)}$$

$$h_v^{(k)} = \sigma\left(W^{(k)}\left[h_v^{(k-1)} \| h_{\mathcal{N}(v)}^{(k)}\right]\right)$$

```
# code: Tensorflow 2
# src: edge source node IDs (n_nodes,)
# dst: edge destination node IDs (n_nodes,)
# H: node repr matrix (n_nodes, in_dim)
# W: weights (in_dim * 2, out_dim)

# Broadcast source features to edges
H_src = tf.gather(H, src)
H_N = tf.math.unsorted_segment_max(
    H_src, dst, n_nodes)
H = tf.nn.relu(tf.concat([H_N, H], 1) @ W)
```

```
# code: PyTorch + DGL
# G: DGL Graph
# H: node repr matrix (n_nodes, in_dim)
# W: weights (in_dim * 2, out_dim)
import dgl.function as fn
G.ndata['h'] = H
# NOT broadcasting source features to edges
G.update_all(fn.copy_u('h', 'm'), fn.max('m', 'h_n'))
H_N = G.ndata['h_n']
H = torch.relu(torch.cat([H_N, H], 1) @ W)
```

With attention?

If $f(\cdot)$ is a *weighted* summation:

$$\hat{\alpha}_{v,u}^{(k-1)} = MLP\left(\left[h_v^{(k-1)} \| h_u^{(k-1)}\right]\right)$$

$$\alpha_{v,u}^{(k-1)} = softmax_j\left(\hat{\alpha}_{v,u}^{(k-1)}\right)$$

$$h_{\mathcal{N}(v)}^{(k)} = \sum_{u \in \mathcal{N}(v)} \alpha_{v,u}^{(k-1)} h_u^{(k-1)}$$

$$h_v^{(k)} = \sigma\left(W^{(k)}\left[h_v^{(k-1)}; h_{\mathcal{N}(v)}^{(k)}\right]\right)$$

One can write his/her own message and aggregation functions:

```
# code: PyTorch + DGL
# G: DGL Graph
# H: node repr matrix (n_nodes, in_dim)
# W: weights (in_dim * 2, out_dim)
def msg_func(edges):
    h_src = edges.src['h']
    h_dst = edges.dst['h']
    alpha_hat = MLP(torch.cat([h_dst, h_src], 1))
    return {'m': h_src, 'alpha_hat': alpha}

def reduce_func(nodes):
    # Incoming messages are batched along 2nd axis.
    # m has a shape of
    # (n_nodes_in_batch, in_degrees, msg_dims)
    m = nodes.mailbox['m']
    # alpha_hat has a shape of
    # (n_nodes_in_batch, in_degrees)
    alpha_hat = nodes.mailbox['alpha_hat']

    alpha = torch.softmax(alpha_hat, 1)
    return {'h_n': (m * alpha[:, None]).sum(1)}

import dgl.function as fn
G.ndata['h'] = H
G.update_all(msg_func, reduce_func)
H_N = G.ndata['h_n']
H = torch.relu(torch.cat([H_N, H], 1) @ W)
```

With attention?

If $f(\cdot)$ is a *weighted* summation:

$$\hat{\alpha}_{v,u}^{(k-1)} = MLP\left(\left[h_v^{(k-1)} \| h_u^{(k-1)}\right]\right)$$

$$\alpha_{v,u}^{(k-1)} = softmax_j\left(\hat{\alpha}_{v,u}^{(k-1)}\right)$$

$$h_{\mathcal{N}(v)}^{(k)} = \sum_{u \in \mathcal{N}(v)} \alpha_{v,u}^{(k-1)} h_u^{(k-1)}$$

$$h_v^{(k)} = \sigma\left(W^{(k)}\left[h_v^{(k-1)}; h_{\mathcal{N}(v)}^{(k)}\right]\right)$$

Built-in message/reduce functions are more time-/memory-efficient.

```
# code: PyTorch + DGL
# G: DGL Graph
# H: node repr matrix (n_nodes, in_dim)
# W: weights (in_dim * 2, out_dim)
# edge_softmax uses built-ins in computation
from dgl.nn.pytorch import edge_softmax
import dgl.function as fn
def msg(edges):
    h_src = edges.src['h']
    h_dst = edges.dst['h']
    return {'alpha': MLP(torch.cat([h_dst, h_src], 1))}

G.ndata['h'] = H
G.apply_edges(msg)  # Edges now have a feature called 'alpha'
G.edata['alpha'] = edge_softmax(G, G.edata['alpha'])
G.update_all(
    fn.u_mul_e('h', 'alpha', 'm'), fn.sum('m', 'h_n'))
H_N = G.ndata['h_n']
H = torch.relu(torch.cat([H_N, H], 1) @ W)
```

How about LSTM?

If $f(\cdot)$ is summation:

$$h_{\mathcal{N}(v)}^{(k)} = LSTM(h_{u_1}^{(k-1)}, \dots, h_{u_n}^{(k-1)})$$

where $u_i \in \mathcal{N}(v)$ are in some order

$$h_v^{(k)} = \sigma\left(W^{(k)}\left[h_v^{(k-1)} \| h_{\mathcal{N}(v)}^{(k)}\right]\right)$$

```
# code: PyTorch + DGL
# G: DGL Graph
# t: timestamp of edges.
#    LSTM will go through messages in the order
#    of timestamps
# H: node repr matrix (n_nodes, in_dim)
# lstm: LSTM module
# W: weights (in_dim * 2, out_dim)
def reduce_func(nodes):
    indices = nodes.mailbox['t'].argsort(1)
    m = nodes.mailbox['m']
    m_ordered = m.gather(1, t[:, :, None].expand_as(m))
    return {'h_n': lstm(m)}

import dgl.function as fn
G.ndata['h'] = H
G.update_all(fn.copy_u('h', 'm'), reduce_func)
H_N = G.ndata['h_n']
H = torch.relu(torch.cat([H_N, H], 1) @ W)
```

How about updating partially[6,7]?

DGL does not confine itself in full-graph updates; one can send messages on, and receive message along, *some of* the edges at a time.

```
# code: PyTorch + DGL
# An extremely simplified version of Know-Evolve, where
# messages are sent/received in the order of edge timestamps.
# H: node repr matrix (n_nodes, in_dim)
# T: numpy array of edge timestamps
def msg_func(edges):
    return {'m': MLP_msg(edges.src['h'])}
def reduce_func(nodes):
    h_old = nodes.data['h']
    h_n = nodes.mailbox['m'].sum(1)
    return {'h': MLP_reduce(torch.cat([h_old, h_n], 1))}

G.ndata['h'] = H
distinct_T = np.sort(np.unique(T))
for t in distinct_T:
    eid = np.where(T == t)
    G.send_and_recv(eid, msg_func, reduce_func)
H_output = G.ndata['h']
```

[6] Trivedi et al., *Know-Evolve: Deep Temporal Reasoning for Dynamic Knowledge Graphs*, ICML 2017

[7] Tai et al., *Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks* (TreeLSTM), ACL 2015

How about heterogeneous graphs[8]?

- DGL supports heterogeneous graphs whose nodes and edges are typed and may have type-specific features.
- One can perform message passing on one edge type at a time.

```python
# code: PyTorch + DGL
# xs: node features for each node type
# ws: weights for each edge type
# g: DGL heterogeneous graph
for i, ntype in enumerate(g.ntypes):
    g.nodes[ntype].data['x'] = xs[i]

# intra-type aggregation
for i, (srctype, etype, dsttype) in enumerate(g.canonical_etypes):
    g.nodes[srctype].data['h'] = g.nodes[srctype].data['x'] @ ws[etype]
    g[srctype, etype, dsttype].update_all(
        fn.copy_u('h', 'm'), fn.mean('m', 'h_%d'))

# inter-type aggregation
for ntype in g.ntypes:
    g.nodes[ntype].data['h'] = sum(
        g.nodes[ntype].data[h_name]
        for h_name in g.nodes[ntype].data.keys()
        if h_name.startswith('h_'))
```

[8] Schlichtkrull et al., *Modeling Relational Data with Graph Convolutional Networks*

How about heterogeneous graphs?

- DGL supports heterogeneous graphs whose nodes and edges are typed and may have type-specific features.
- One can also perform message passing on multiple edge types, further aggregating the outcome of per-edge-type aggregation with an *cross-type reducer*.

```
# code: PyTorch + DGL
# xs: node features for each node type
# ws: weights for each edge type
# g: DGL heterogeneous graph
for i, ntype in enumerate(g.ntypes):
    g.nodes[ntype].data['x'] = xs[i]

funcs = {}
for i, (srctype, etype, dsttype) in enumerate(g.canonical_etypes):
    g.nodes[srctype].data['h%d' % i] = g.nodes[srctype].data['x'] @ ws[etype]
    funcs[(srctype, etype, dsttype)] = (
        fn.copy_u('h%d' % i, 'm'), fn.mean('m', 'h'))

# message passing
g.multi_update_all(funcs, cross_reducer='sum')
```

How about scaling to larger graphs?

- Full-graph updates become infeasible on large graphs with millions of nodes and billions of edges.
- We usually sample a batch of nodes at a time to compute their representations for loss computation.
- Furthermore, for each node, we can choose to receive messages from only a few nodes.

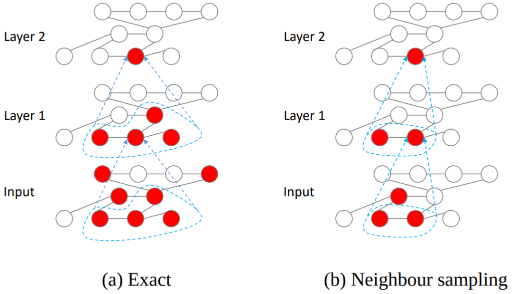
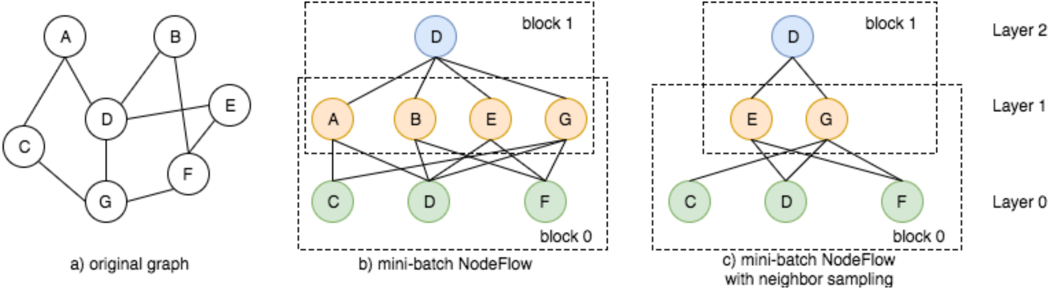


Figure taken from Chen et al., *Stochastic Training of Graph Convolutional Networks with Variance Reduction*, ICML 2018

Scaling to larger graphs (with NodeFlow)

For each minibatch of nodes, we explicitly construct another graph containing the dependency between nodes on each message passing layer.



Scaling to larger graphs (with NodeFlow)

```
# code: MXNet + DGL
from dgl.contrib.sampling import NeighborSampler

# initialize the model and cross entropy loss
model = GCNSampling(in_feats, n_hidden, n_classes, L,
                    mx.nd.relu, dropout, prefix='GCN')
model.initialize()
loss_fcn = gluon.loss.SoftmaxCELoss()

for nf in NeighborSampler(
        g, batch_size, num_neighbors,
        neighbor_type='in', shuffle=True,
        num_hops=L, seed_nodes=train_nid):
    nf.copy_from_parent()
    with mx.autograd.record():
        # forward
        pred = model(nf)
        batch_nids = (
             nf.layer_parent_nid(-1)
             .astype('int64'))
        batch_labels = labels[batch_nids]
        # cross entropy loss
        loss = loss_fcn(pred, batch_labels)
        loss = loss.sum() / len(batch_nids)
    loss.backward()
```

```
# code: MXNet + DGL
class GCNSampling(gluon.Block):
    # __init__ is skipped...

    def forward(self, nf):
        nf.layers[0].data['activation'] = \
            nf.layers[0].data['features']
        for i, MLP in enumerate(self.MLPs):
            h = nf.layers[i].data.pop('activation')
            nf.layers[i].data['h'] = h
            nf.block_compute(
                i, fn.copy_src(src='h', out='m'),
                fn.mean('m', 'h'))
            nf.layers[i + 1].data['h'] = MLP(h)
        h = nf.layers[-1].data.pop('activation')
        return h
```

What's more?

- Check out our repository: `https://github.com/dmlc/dgl`
  - We have lots of PyTorch and MXNet examples!
  - In 0.4 we also released DGL-KE, a subpackage for training knowledge graph embeddings.
- Check out our documentation: `https://docs.dgl.ai`
- Discussion forum: `https://discuss.dgl.ai`