Introduction to DGL

Quan Gan

AWS Shanghai AI Lab

October 17, 2019

What does DGL provide?

- From bottom-level to top-level:
  - *Plug-and-play model zoo*, to run an existing model on your data directly
  - *Easy-to-use graph neural network layer modules*, to plugin popular GNN layers into your model.
  - *Flexible and efficient message passing APIs*, to design your own message passing (not necessarily full-graph!) from scratch.

Model Zoo?

- Get a (pretrained) model that works on molecules immediately:

```
from dgl.model_zoo.chem import load_pretrained
model = load_pretrained('MPNN_Alchemy')
result = model(molecule, atom_features, bond_features)
```

- We just released a subpackage *DGL-KE* for training embeddings on large scale knowledge graphs such as FreeBase.
- We are shooting for a model zoo for recommender systems in 0.5.

Graph Neural Network Layer Modules?

- Use DGL GNN Modules to build a bigger network:

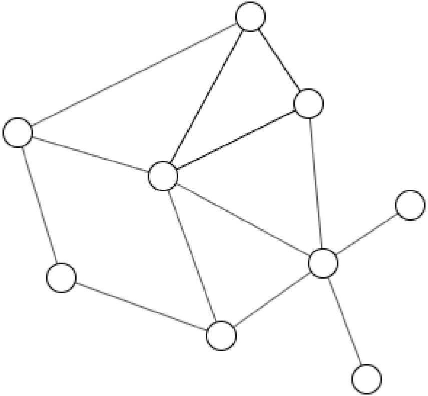
```
from dgl.nn.pytorch import SAGEConv

# One layer GraphSAGE
class NodeClassifier(nn.Module):
    def __init__(self, in_dim, n_classes):
        self.gnn = SAGEConv(in_dim, in_dim, 'mean')
        self.cls = nn.Linear(in_dim, n_classes)
    def forward(self, g, x):
        h = self.gnn(g, x)
        return self.cls(h)
```

- We have lots of popular GNN modules implemented.
  - The list is growing!

Custom Graph Neural Network Layer?

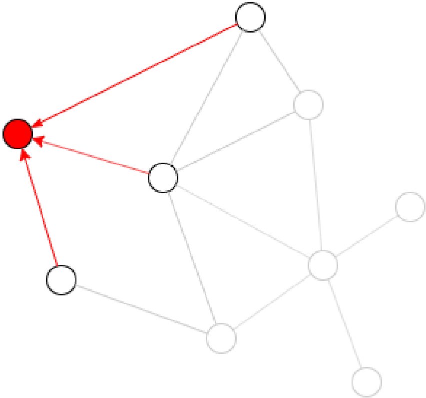
$$h_v^{(k)} = \phi\left(h_v^{(k-1)}, h_{\mathcal{N}(v)}^{(k)}\right) \qquad h_{\mathcal{N}(v)}^{(k)} = f\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)$$ [1]



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

Custom Graph Neural Network Layer?

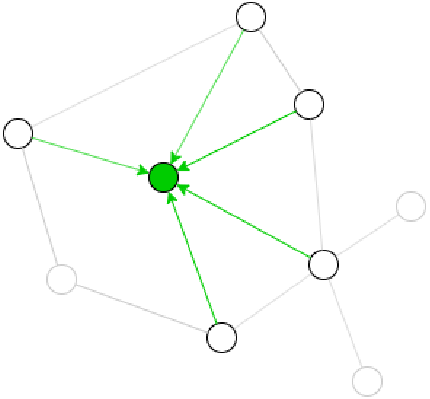
$$h_v^{(k)} = \phi\left(h_v^{(k-1)}, h_{\mathcal{N}(v)}^{(k)}\right) \qquad h_{\mathcal{N}(v)}^{(k)} = f\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)$$ [1]



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

Custom Graph Neural Network Layer?

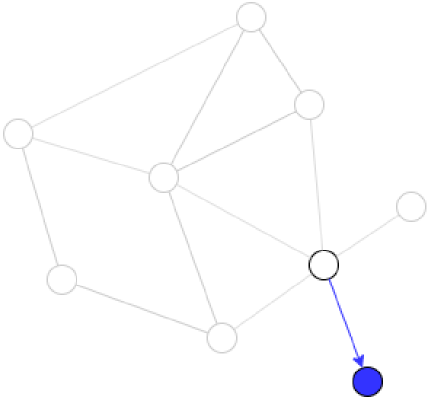
$$h_v^{(k)} = \phi\left(h_v^{(k-1)}, h_{\mathcal{N}(v)}^{(k)}\right) \qquad h_{\mathcal{N}(v)}^{(k)} = f\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)$$ [1]



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

Custom Graph Neural Network Layer?

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



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

Aggregation: Average Pooling[2]

Sparse matrix multiplication, very well-known:

```
# 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)
```

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

How about max pooling?

Not possible in Vanilla PyTorch & MXNet. Not memory-efficient 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)

# 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
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?[3]

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)
# SIMPLIFIED - only one attention head is
     considered
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)
```

```
# 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(msg_func, reduce_func)
H_N = G.ndata['h_n']
H = torch.relu(torch.cat([H_N, H], 1) @ W)
```

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

With attention?

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)
# SIMPLIFIED - only one attention head is
    considered
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)
```

```
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 2
    nd axis.
    m = nodes.mailbox['m']
    alpha_hat = nodes.mailbox['alpha_hat']
    alpha = torch.softmax(alpha_hat, 1)
    return {'h_n':
        (m * alpha[:, None]).sum(1)}
```

How about LSTM[4][5]?

```
# 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(n_nodes):
    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)
```

```
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)
```

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

How about LSTM?

```
# 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(n_nodes):
    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)
```

```
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)}
```

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
# messages are sent/received in the order of
      edge timestamps.
# H: node repr matrix (n_nodes, in_dim)
# T: numpy array of edge timestamps
G.ndata['h'] = H
distinct_T = np.sort(np.unique(T))
for t in distinct_T:
    eid = np.where(T == t)
    G.reduce_func(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.

```
# 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'))
```

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

How about heterogeneous graphs?

- 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')
```

Comparison of flexibility

Computation	Tensorflow	PyTorch/MXNet	DGL
Average pooling	Sparse matmul	Sparse matmul	Message Passing API
Max pooling	Segment-max	N/A	
Attention pooling	Sparse softmax	N/A	
LSTM pooling	Sequence padding	Sequence padding	
Partial graph computation	Manual labor	Manual labor	
Heterogeneous graph	Manual labor	Manual labor	

Is it efficient?

Model	Train time/epoch (Original)	Train time/epoch (DGL)	Speedup
Graph Convolutional Networks	0.0051s (TF)	0.0031s	1.64x
Graph Attention Networks	0.0982s (TF)	0.0113s	8.69x
Relational GCN (classification)	0.2853s (Theano)	0.0075s	38.2x
Relational GCN (link prediction)	2.204s (TF)	0.453s	4.86x
Graph Convolutional Matrix Completion (MovieLens-100k)	0.1008s (TF)	0.0246s (MXNet)	4.09x
TreeLSTM	14.02s (DyNet)	3.18s	4.3x
Junction Tree Variational Autoencoder	1826s (PyTorch)	743s	2.5x

And much more examples....

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`
- Stay tuned for 0.5, which will include better support on large-scale & distributed GNN training!