Graph Representation Learning

Kaige Yang
University College London
`Kaige.yang.11@ucl.ac.uk`

1 Graph Representation Learning

Graph (network) consists of nodes to represent entities and edges to denote the relationships or interactions between them. The graph can be enrich by adding features to nodes of edges. The networks with multiple node or edge type are called heterogeneous networks, whereas networks with a single node and edge type are named as homogeneous networks.

Graph representation learning is basically any method that takes a graph (node features, edge features) and output vectors which correspond to the embedding of nodes/edge/the whole graph. The resulting representation can be used as input of neural network, principle component analysis, support vector machine or other machine learning algorithms.

Tasks:

- Node Classification
- Link Prediction
- Node Clustering
- Graph Classification

2 Node2Vec

Node2Vec is an algorithm for learning representation for nodes in graph given the graph as input. The node/edge features are no used in node2vec.

Idea:
In node2vec, we learn a mapping of nodes to a low-dimensional space of features that maximizes the likelihood of preserving network neighborhoods of nodes.

The notation of neighborhood
It is essential to allow for a flexible algorithm that can learn node representations obeying two principles:

- Ability to learning representation that embed nodes from the same network community closely together
- To learn representation where nodes that share similar structural roles have similar embedding.

How to sampling neighbors?
Given a source node, we aim to generate its neighborhood. There are two ways to find neighbors

- Breath-first sampling (BFS): samples k immediate neighbors. This is based on homophily hypothesis: nodes that are highly connected and belongs to similar clusters or communities should be embedded closely together.

- Depth-first sampling (DFS): samples k nodes sequentially at increasing distance. This is based on structural equivalence: nodes that have similar structural roles should be embedded closely together.

Random walks: we simulate a random walk of fixed length l. Let c_i denote the i-th node in the walk, starting with $c_0 = u$. Node c_i are generated by the following distribution

$$Pr(c_i = x | c_{i-1} = v) = \pi_{uv}/Z,\ if\ (v, u) \in E, 0\ otherwise \tag{1}$$

We define a 2nd order random walk with two parameters p and q. Consider a random walk that just traversed edge (t, v) an now resides at node v. The walk now needs to decide on the next step so it evaluates the transition probability $\pi(v, x)$ leading from v. We set the unnormalized transition probability to

$$\tilde{\pi}_{vx} = \alpha_{pq}(t, x) w_{vx} \tag{2}$$

where $w_{vx} = \pi_{vx}$ and

$$\alpha_{pq}(t, x) = \begin{cases} \frac{1}{p} & if\ d_{tx} = 0 \\ 1 & if\ d_{tx} = 1 \\ \frac{1}{q} & if\ d_{tx} = 2 \end{cases} \tag{3}$$

where d_{tx} denotes the shortest path distance between nodes t and x.

Intuitively, parameters p and q control how fast the walk explores and leaves the neighborhood of starting node u.

2.1 Node2Vec Theory

we design an objective that seeks to preserve local neighborhood of nodes. The objective can be efficiently optimized using stochastic gradient descent akin to back-propagation on just single hidden-layer feed-forward neural network.

$$\max_f \sum_{u \in V} \log Pr(N_s(u) | f(u)) \tag{4}$$

where

$$Pr(N_s(u) | f(u)) = \Pi_{n_i \in N_s(U)} Pr(n_i | f(u)) \tag{5}$$

$$Pr(n_i | f(u) = \frac{\exp(f(n_i)^T f(u))}{\sum_{v \in V} \exp(f(v)^T f(u))} \tag{6}$$

The objective function is

$$\max_f \sum_{u \in V} \left[-\log \sum_{v \in V} \exp(f(v)^T f(u)) + \sum_{n_i \in N_s(u)} f(n_i)^T f(u) \right] \tag{7}$$

2.2 Node2Vec in Practice

At every step of the walk, sampling is one based on the transition probability π_{vx}. The three phases of Node2Vec:

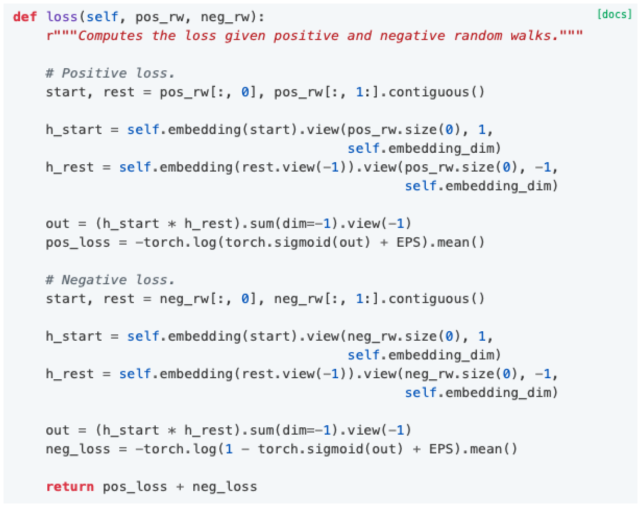
- Pre-processing to compute transition probabilities.
- Random walk simulations.
- Optimization using SGD.

The embedding is learned by torch.nn.embedding layer. The objective loss is calculated as shown in Figure 2.

```
**Algorithm 1** The *node2vec* algorithm.
**LearnFeatures** (Graph $G = (V, E, W)$, Dimensions $d$, Walks per
    node $r$, Walk length $l$, Context size $k$, Return $p$, In-out $q$)
    $\pi$ = PreprocessModifiedWeights($G, p, q$)
    $G' = (V, E, \pi)$
    Initialize $walks$ to Empty
    for $iter = 1$ to $r$ do
        for all nodes $u \in V$ do
            $walk$ = node2vecWalk($G', u, l$)
            Append $walk$ to $walks$
    $f$ = StochasticGradientDescent($k$, $d$, $walks$)
    return $f$

**node2vecWalk** (Graph $G' = (V, E, \pi)$, Start node $u$, Length $l$)
    Inititalize $walk$ to $[u]$
    for $walk\_iter = 1$ to $l$ do
        $curr = walk[-1]$
        $V_{curr}$ = GetNeighbors($curr, G'$)
        $s$ = AliasSample($V_{curr}, \pi$)
        Append $s$ to $walk$
    return $walk$
```

Figure 1: Node2Vec Algorithm



```
def loss(self, pos_rw, neg_rw):                                    [docs]
    r"""Computes the loss given positive and negative random walks."""

    # Positive loss.
    start, rest = pos_rw[:, 0], pos_rw[:, 1:].contiguous()

    h_start = self.embedding(start).view(pos_rw.size(0), 1,
                                         self.embedding_dim)
    h_rest = self.embedding(rest.view(-1)).view(pos_rw.size(0), -1,
                                                self.embedding_dim)

    out = (h_start * h_rest).sum(dim=-1).view(-1)
    pos_loss = -torch.log(torch.sigmoid(out) + EPS).mean()

    # Negative loss.
    start, rest = neg_rw[:, 0], neg_rw[:, 1:].contiguous()

    h_start = self.embedding(start).view(neg_rw.size(0), 1,
                                         self.embedding_dim)
    h_rest = self.embedding(rest.view(-1)).view(neg_rw.size(0), -1,
                                                self.embedding_dim)

    out = (h_start * h_rest).sum(dim=-1).view(-1)
    neg_loss = -torch.log(1 - torch.sigmoid(out) + EPS).mean()

    return pos_loss + neg_loss
```

Figure 2: Node2Vec Loss Function

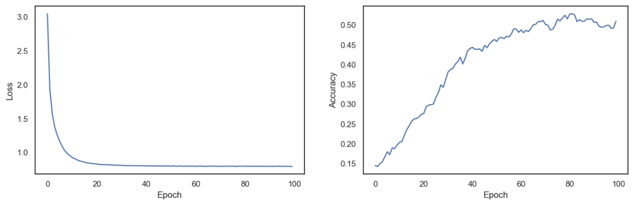


Figure 3: Node2Vec: Training Loss and Accuracy

The learned node embedding are tested on node classification problem via Logistic regression. Figure 3 shows the training loss and accuracy.

3 Stru2Vec

Structural identity is a concept of symmetry in which network nodes are identified according to the network structure and their relationship to other nodes.

This work presents struc2vec a novel framework for learning latent representation for the structural identity of nodes.

Idea
struc2vec uses a hierarchy to measure node similarity at different scales and constructs a multi-layer graph to encode structural similarities and generate structural context for nodes.

Main steps

- Determine the structural similarity between nodes for different neighborhood size. This induces a hierarchy in the measure for structural similarity.
- Construct a weighted multi-layer graph where all nodes in the network are present in every layer and each layer corresponds to a level of hierarchy in measuring structural similarity.
- Use the multi-layer graph to generate context for each node. In particular, a biased random walk on the multi-layer graph is used to generate node sequence.
- Apply Skip-Gram to learn node embedding.

How to find structural similarity?
Let $f_k(u,v)$ denote the structural distance between u and v consider their $k-hop$ neighborhoods.

$$f_k(u,v) = f_{k-1}(u,v) + g(s(R_k(u)), s(T_k(v))),\ k \geq 0\ and\ |R_k(u)|,\ |R_k(v)| > 0 \tag{8}$$

Construct the mutil-layer graph
Let M denote the multi-layer graph where layer k is defined using $k-hop$ neighborhoods of the nodes

Each layer $k \in 0,..,k^*$ is formed by a weighted undirected complete graph with node set V. The edge weight between nodes in a layer is

$$w_k(u,v) = e^{-f_k(u,v)},\ k = 0,...,k^* \tag{9}$$

Note that weights are inversely proportional to structural distance. nodes that are structurally similar to u will have larger weights across various layer of M.

We connect layers using directed edges. Every node in layer k is connected to the corresponding node in layer $k-1$ and $k+1$. The edge weight is

$$w_{u_k,u_{k+1}} = \log(\Gamma_k(u) + e),\ k = 0,...,k^* - 1 \tag{10}$$

$$w_{u_k,u_{k-1}} = 1,\ k = 1,...,k^* \tag{11}$$

where

$$\Gamma_k(u) = \sum_{v \in V} \mathbb{I}(w_k(u,v)|\bar{w}_k) \tag{12}$$

where $\bar{w}_k = \sum_{(u,v)} w_k(u,v)/edge_num$.
Note that $\Gamma_k(u)$ measures the similarity of node u to other nodes in layer k.

Generate context for nodes
M captures the structural similarities of nodes in G. struc2vec uses bias random walk to determine the context of a given node. The bias random walk move around M according to weights.

$$p_k(u,v) = \frac{e^{-f_k(u,v)}}{Z_k(u,v)} \tag{13}$$

$$Z_k(u) = \sum_{v \in V} e^{-f_k(u,v)} \tag{14}$$

The random walk will prefer to step onto nodes that are structurally more similar to the current node, avoiding nodes that have very little structural similarity with it.

Learn embedding
SkipGram algorithm is used.

4 DeepWalk

We present DeepWalk, a novel approach for learning representation of nodes in a network. This approach used local information obtained from truncated random walks.

Idea: nodes which have similar neighborhoods should be embedded closely together.

$$\min_{\Phi} -\log Pr(\{v_{i-w}, ..., v_{i+w}\} \setminus v_i | \phi(v_i)\}) \tag{15}$$

Solving the optimization problem builds representation structure between nodes.

DeepWalk
This algorithm consists of two main components

- A random walk generator:
  sample uniformly a random node as the root of random walk. A walk samples uniformly from the neighbors of the last node visited until the maximum length t is reached. A number of random walks γ of length t is generated stating at each node.
- SkipGram algorithm to update representation.

$$Pr(\{v_{i-w}, ..., v_{i+w} \setminus v_i | \phi(v_i)\} = \Pi_{j=i-w}^{i+w} Pr(v_j | \phi(v_i)) \tag{16}$$

5 GraphWave

GraphWave learning a representation for each node based on the diffusion of a spectral graph wavelet centered at the node. The diffusion process contains the neighboring topology of a node.

Nodes residing in different parts of a graph can have similar structural riles within their local network topology. This alternative definition of node similarity is very different than more traditional notion, which assumes some measure of smoothness over the graph and thus consider nodes residing in close network proximity to be similar.

Spectral graph wavelet
Let U be the eigenvector decomposition of the Lapalcain $L = D - A = U\Lambda U^T$ and let $\lambda_1 < \lambda_2, ... \leq \lambda_N$ denote the eigenvalues of L. Let $g_s(\lambda) = e^{-\lambda s}$ be the heat kernel. The spectral wavelet Ψ_a round node a is defined as

$$\Psi_a = UDiag(g_s(\lambda_1), ..., g_s(\lambda_N))U^T \delta_a \in \mathbb{R}^{N \times 1} \tag{17}$$

where $\delta_a = \mathbb{I}_a$ is a one-hot vector for node a, which is a Dirac signal centered around node a.

GraphWave
For every node a, GraphWave returns a 2d-dimensional vector $\mathcal{X}_a$ representation its structural embedding, where nodes with structurally similar local network neighborhoods will have similar embeddings.

- We first apply spectral graph wavelets to obtain diffusion pattern for every node, which we gather in a matrix $\Psi \in \mathbb{R}^{N \times N}$. where a-th column vectors is the spectral graph wavelet centered at node a.
- We model the coefficient matrix as a probability distribution and characterizes the distribution via empirical characteristic function.

$$\phi(\Psi) = \mathbb{E}[e^{it\Psi}] \tag{18}$$

  The empirical characteristic function of Ψ_a is defined as

$$\phi_a(t) = \frac{1}{N} \sum_{m=1}^{N} e^{it\Psi_{ma}} \tag{19}$$

- Finally, the structural embedding $\mathcal{X}_a$ is obtained by sampling a a-dimensional parametric function at d every spaced pints $t_1, ..., t_d$ and concatenating the values.

$$\mathcal{X}_a = [Re(\phi_a(t_i)), Im(\phi_a(t_i)]_{t_1,...,t_d} \tag{20}$$

6 Deep Learning on Graphs

There are multiple flavours to graph learning problems that are largely application-dependent. The approaches can be categorized in the following ways

- Node wise, graph wise
- Supervised and unsupervised
- transductive and inductive

7 GraphSAGE

Low-dimensional embedding of nodes in large graphs have proved extremely useful in a variety of prediction tasks. Previous approaches are inherently transductive and do not naturally generalize to unseen nodes. Here, we present GraphSage a general inductive framework that leverages node features to efficiently generate unseen nodes. This algorithm can also be applied to graphs without node features.

Idea
Instead of training a distinct embedding vector for each node, we train a set of aggregator function that learn to aggregate feature information from node's neighborhood. Each aggregator function aggregates information from a different number of hops, or search depth, away from a given node. The intuition behind is that at each iteration, or search depth, nodes aggregate information from their local neighbors, and as this process iterates, nodes incrementally gain more and more information from further reaches of the graph. At test time, the trained aggregator functions generate embedding for unseen node based on its neighborhood. We design an unsupervised loss function that allows GraphSage to be trained without task-specific supervision.

How to find embeddings?
We assume that we have learned the parameters of K aggregator functions $Aggr_k, \forall k \in [K]$, which aggregate information from node's neighborhood, as well as a set of weight matrices W^k, which are used to propagate information between different layers of the model or search depths.

How to define neighborhood?
We uniformly sample a fixed-size set of immediate neighbors and we draw different uniform samples at each search depth k.

How to learn W^k and parameters of $Aggr_k$?
The graph-based loss function encourages nearby nodes to have similar representation, while enforces that the representation of disparate nodes are highly distinct

$$J_G(z_u) = -\log(\sigma(z_u^T z_v)) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log(\sigma(-z_u^T z_{v_n})) \tag{21}$$

where v is a node that co-occures near u on fixed-length random walk, σ is a sigmoid function. P_n is a negative sampling distribution and Q defines the number of negative samples.

The loss function can be modified by a task-specific objective (cross-entropy loss) for down-stream task.

Variant forms of $Aggr$

- Mean aggregator

$$h_v^k \leftarrow \sigma(W \cdot MEAN(\{h_v^{k-1}\} \cup \{h_u^{k-1}, \forall u \in N(v)\} \tag{22}$$

- LSTM aggregator
- Pooling aggregator:

$$Aggr_k = \max(\{\sigma(Wh_u^k + b), \forall u \in N(v)\}) \tag{23}$$

Note: What if no node features?

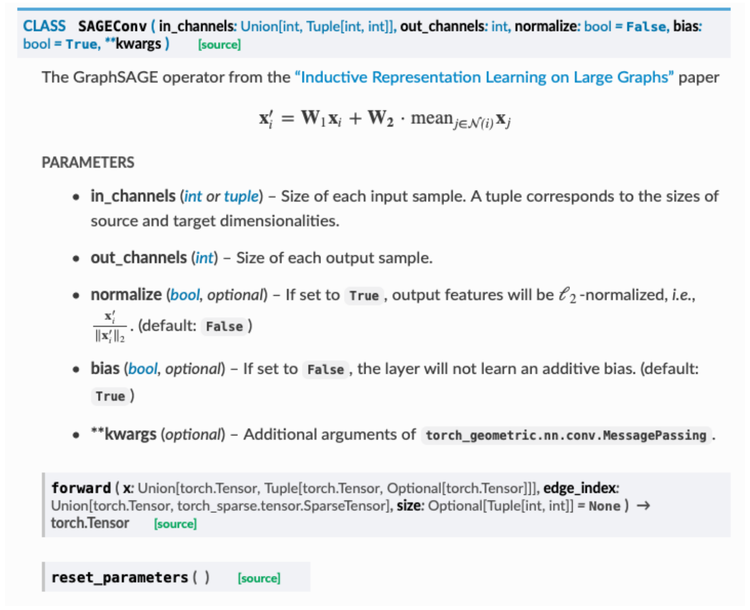
Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

```
Input  : Graph $\mathcal{G}(\mathcal{V},\mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth $K$; weight matrices
         $\mathbf{W}^k, \forall k \in \{1,...,K\}$; non-linearity $\sigma$; differentiable aggregator functions
         $\text{AGGREGATE}_k, \forall k \in \{1,...,K\}$; neighborhood function $\mathcal{N} : v \to 2^{\mathcal{V}}$
Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$
1 $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$ ;
2 for $k = 1...K$ do
3     for $v \in \mathcal{V}$ do
4         $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
5         $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
6     end
7     $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
8 end
9 $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$
```

Figure 4: GraphSage Algorithm

7.1 GraphSAGE in Practice

At each iteration (K), the embedding of each node is updated by first aggregating its neighbors (1-hop or k-hop) and then combined the aggregated feature and its own feature. As the iteration processes, information from far way is propagated through neighbors. The algorithm is shown is Figure 4. The message passing scheme is implemented by module shown in Figure 5.



CLASS **SAGEConv** (**in_channels**: Union[int, Tuple[int, int]], **out_channels**: int, **normalize**: bool = `False`, **bias**: bool = `True`, ****kwargs**) [source]

The GraphSAGE operator from the "Inductive Representation Learning on Large Graphs" paper

$$\mathbf{x}'_i = \mathbf{W}_1 \mathbf{x}_i + \mathbf{W}_2 \cdot \text{mean}_{j \in \mathcal{N}(i)} \mathbf{x}_j$$

PARAMETERS

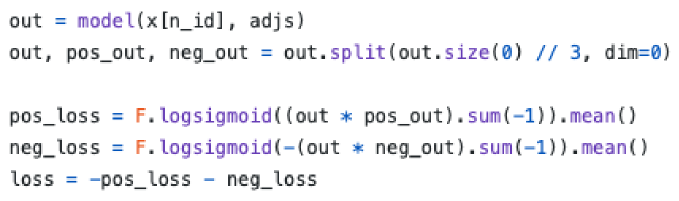
- **in_channels** (*int or tuple*) – Size of each input sample. A tuple corresponds to the sizes of source and target dimensionalities.
- **out_channels** (*int*) – Size of each output sample.
- **normalize** (*bool, optional*) – If set to `True`, output features will be ℓ_2-normalized, *i.e.*, $\frac{\mathbf{x}'_i}{\|\mathbf{x}'_i\|_2}$. (default: `False`)
- **bias** (*bool, optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)
- ****kwargs** (*optional*) – Additional arguments of `torch_geometric.nn.conv.MessagePassing`.

forward (**x**: Union[torch.Tensor, Tuple[torch.Tensor, Optional[torch.Tensor]]], **edge_index**: Union[torch.Tensor, torch_sparse.tensor.SparseTensor], **size**: Optional[Tuple[int, int]] = `None`) → torch.Tensor [source]

reset_parameters ( ) [source]

Figure 5: GraphSage Module

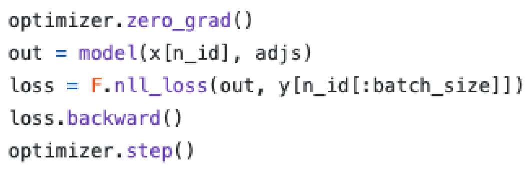
The loss objective of GraphSage could be the loss defined in Eq. 21 (shown in Figure 6) or the loss of downstream machine learning task (i.e, node classification Figure 7).



```
out = model(x[n_id], adjs)
out, pos_out, neg_out = out.split(out.size(0) // 3, dim=0)

pos_loss = F.logsigmoid((out * pos_out).sum(-1)).mean()
neg_loss = F.logsigmoid(-(out * neg_out).sum(-1)).mean()
loss = -pos_loss - neg_loss
```

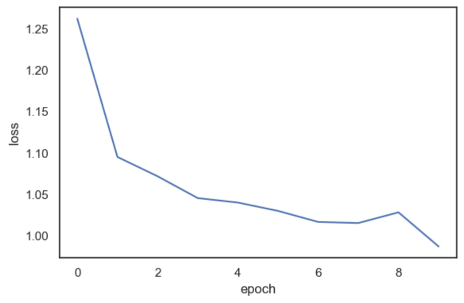
Figure 6: GraphSage Loss 1



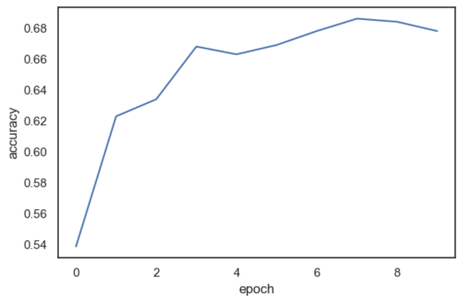
```
optimizer.zero_grad()
out = model(x[n_id], adjs)
loss = F.nll_loss(out, y[n_id[:batch_size]])
loss.backward()
optimizer.step()
```

Figure 7: GraphSage Loss 2

The algorithm is test on Cora dataset. The learned node embeddings are test on node classification problem via Logistic Regression. The learning curve and classification accuracy are shown in Figure 8.



(a) Loss



(b) Accuracy

Figure 8: GraphSage

8 GCN

Task: semi-supervised Node classification.
Assumption: Label information is smoothed over the graph via some form of explicit graph-based regularization.
Prapagation rule

$$H^{(l+1)} = \sigma(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}H^{(l)}W^{(l)}) \tag{24}$$

where $\tilde{A} = A + I_n$, $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ and $W^{(l)}$ is the layer-specific trainable weight matrix. $\sigma(\cdot)$ denotes an activation function, such as the $ReLU(\cdot) = \max(0, \cdot)$. $H^{(l)} \in \mathbb{R}^{n\times d}$ is the metric of activations in the l^{th} layer. $H^{(0)} = X$.

Example: Two-layer GCN
Let $S = \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2} \in \mathbb{R}^{N\times N}$, $X \in \mathbb{R}^{N\times C}$ in the input node features matrix, $W^{(0)} \in \mathbb{R}^{C\times H}$ the weight of the first layer, $W^{(1)} \in \mathbb{R}^{N\times F}$ the weight of second layer. $Z \in \mathbb{R}^{N\times F}$ is the output node embedding matrix.

$$Z = f(X, A) = softmax(S\sigma(SXW^{(0)})W^{(1)}) \tag{25}$$

where $\sigma(\cdot) = ReLU(\cdot)$.

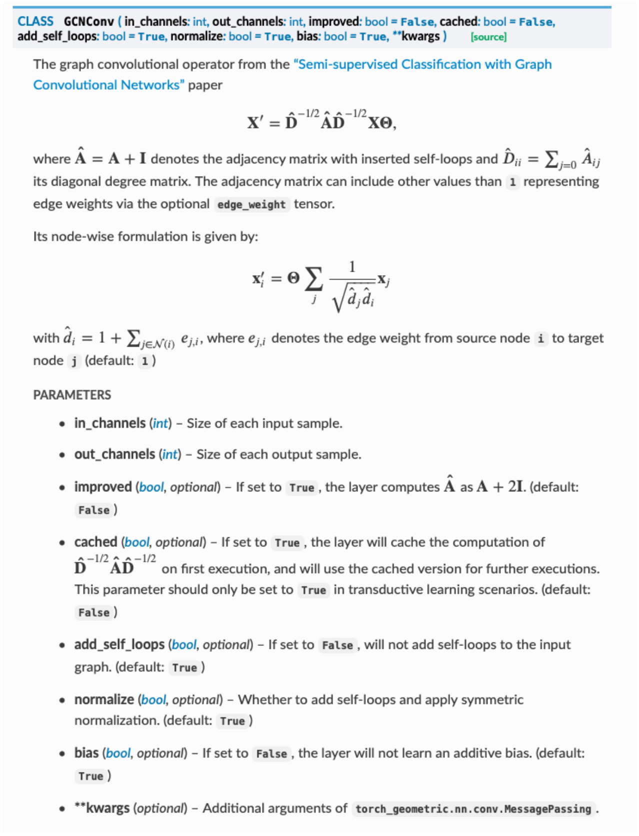
For semi-supervised node classification, we then evaluate the cross-entropy error over all labeled exmapes:

$$\mathcal{L} = -\sum_{l\in\mathcal{Y}_L}\sum_{f=1}^{F} \ln Z_{lf} \tag{26}$$

where $\mathcal{Y}_L$ is the set of node indices that have labels.

8.1 GCN in Practice

The message passing scheme is implemented by GCN module. The loss function is defined as the node classification error.



CLASS GCNConv (in_channels: int, out_channels: int, improved: bool = False, cached: bool = False, add_self_loops: bool = True, normalize: bool = True, bias: bool = True, **kwargs) [source]

The graph convolutional operator from the "Semi-supervised Classification with Graph Convolutional Networks" paper

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{\Theta},$$

where $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency matrix with inserted self-loops and $\hat{D}_{ii} = \sum_{j=0} \hat{A}_{ij}$ its diagonal degree matrix. The adjacency matrix can include other values than `1` representing edge weights via the optional `edge_weight` tensor.

Its node-wise formulation is given by:

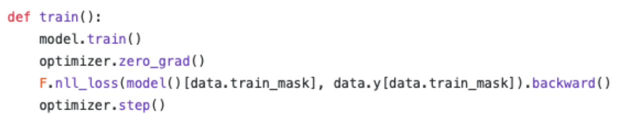
$$\mathbf{x}'_i = \mathbf{\Theta} \sum_j \frac{1}{\sqrt{\hat{d}_j \hat{d}_i}} \mathbf{x}_j$$

with $\hat{d}_i = 1 + \sum_{j \in \mathcal{N}(i)} e_{j,i}$, where $e_{j,i}$ denotes the edge weight from source node `i` to target node `j` (default: `1`)

PARAMETERS

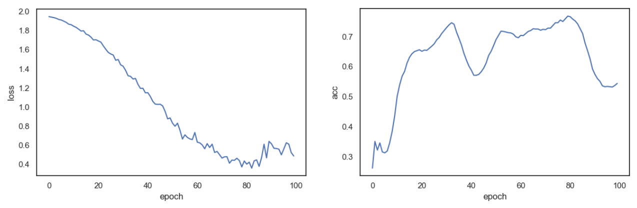
- **in_channels** (*int*) – Size of each input sample.
- **out_channels** (*int*) – Size of each output sample.
- **improved** (*bool*, *optional*) – If set to `True`, the layer computes $\hat{\mathbf{A}}$ as $\mathbf{A} + 2\mathbf{I}$. (default: `False`)
- **cached** (*bool*, *optional*) – If set to `True`, the layer will cache the computation of $\hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}$ on first execution, and will use the cached version for further executions. This parameter should only be set to `True` in transductive learning scenarios. (default: `False`)
- **add_self_loops** (*bool*, *optional*) – If set to `False`, will not add self-loops to the input graph. (default: `True`)
- **normalize** (*bool*, *optional*) – Whether to add self-loops and apply symmetric normalization. (default: `True`)
- **bias** (*bool*, *optional*) – If set to `False`, the layer will not learn an additive bias. (default: `True`)
- ****kwargs** (*optional*) – Additional arguments of `torch_geometric.nn.conv.MessagePassing`.

Figure 9: GCN Module



```
def train():
    model.train()
    optimizer.zero_grad()
    F.nll_loss(model()[data.train_mask], data.y[data.train_mask]).backward()
    optimizer.step()
```

Figure 10: GCN Loss objective



(a) Loss (b) Accuracy

Figure 11: GCN

9 GAT

We introduce an attention-based architecture to perform node classification of graph-structured data.

Idea: The idea is to compute the hidden representations of each node in the graph, by attending over its neighbors, following a self-attention strategy.

Self-attention: When an attention mechanism is used to compute a representation of a single sequence, it is commonly referred to as self-attention or intra-attention.

GAT layer
The input is a set of node features, $x = \{x_1, x_2, ..., x_n\}, x_i \in \mathbb{R}^D$, where n is the number of nodes and D is the dimensionality of node feature. The GAT layer produces a new set of node embedding $h = \{h_1, h_2, ..., h_n\}, h_i \in \mathbb{R}^d$ (of potentially different dimensionality $d \neq D$).

Step 1: Compute attention coefficient

$$e_{ij} = a(Wx_i, Wx_j) \tag{27}$$

where $W \in \mathbb{R}^{d \times D}$ is a weight matrix. a is attention mechanism: $\mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$.

The attention coefficient e_{ij} indicates the importance of node j's features to node i.

We inject the graph structure $\mathcal{G}$ into the mechanism by performing masked attention- we only compute e_{ij} for nodes $j \in \mathcal{N}_i$ where $\mathcal{N}_i$ is the neighborhood of node i. (e.g., one-hop, two-hop).

Step 2: Normalize coefficient across neighbors.

$$\alpha_{ij} = softmax(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})} \tag{28}$$

The neural network realization of **Step 2**.

$$\alpha_{ij} = \frac{\exp\left(LeakyReLU\left(\mathbf{a}^T[Wx_i||Wx_j]\right)\right)}{\sum_{k \in \mathcal{N}_i} \exp\left(LeakyReLU\left(\mathbf{a}^T[Wx_i||Wx_k]\right)\right)} \tag{29}$$

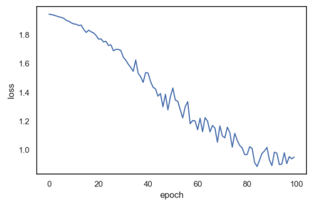
where $||$ is the concatenation operation.

Step 3: Final output features

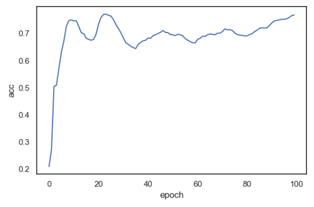
$$h_i = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} W x_j\right) \tag{30}$$

Variant of step 3: multi-head attention (concatenation). $h_i = ||_{k=1}^{K} \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^k W^k x_j\right)$ Here $h_i \in \mathbb{R}^{Kd}$. Or multi-head attention (average): $h_i = \sigma(\frac{1}{K} \sum_{k=1}^{K} \sum_{j \in \mathcal{N}_i} \alpha_{ij}^k W^k x_j)$

9.1 GAT in Practice



(a) Loss



(b) Accuracy

Figure 12: GAT

10 TGN

Challenge:
Despite the plethora of different model for deep learning on graphs, few approaches have been proposed thus far for dealing with graph that present some sort of dynamic nature.

Task:
In this paper, we present Temporal Graph Networks (TGNs), a generic, efficient framework for deep learning on dynamic graphs represented as sequences of timed events.

Learning on static graphs:
A static graph $\mathcal{G} = (V, E)$ comprises nodes set V and edge set E, which are endowed with features, denoted by v_i and e_{ij}, respectively. A typical graph neural network GNN creates an embedding z_i of each node by learning a local aggregation rule of form

$$z_i = \sum_{j \in \mathcal{N}(i)} h(m_{ij}, v_i), \;\; m_{ij} = msg(v_i, v_j, e_{ij}) \tag{31}$$

which is interpreted as message passing from the neighbors h of i. Here, $\mathcal{N}(i)$ is the neighborhood of node i. and $msg()$ and $h()$ are learning messaging passing function and aggregation function.

Dynamic graph:
Two main classes of dynamic graph

- Discrete-time dynamic graphs (DTDG): are sequence of static graphs snapshots taken at intervals in time.
- Continuous-time dynamic graph (CTDG): are more general and can be represented as timed lists of events, which may include edge addition or deletion, node addition or deletion and node or edge feature transformations.

In this work, we do not consider deletion events.

Our temporal graph model:
A sequence of time-stamped events $\mathcal{G} = \{x(t_1), x(t_2), ...\}$ representing addition of change of a node or interaction between a pair of nodes (edge addition) at time $0 \leq t_1 \leq t_2 \leq ....$

An event can be two types:

- A node-wise event is represented by $v_i(t)$.
- An interaction event between node i and j is represented by a (directed) temporal edge $e_{ij}(t)$.

We denote by $\mathcal{V}(T) = \{i : \exists v_i(t) \in \mathcal{G}, t \in T\}$ and $\mathcal{E}(T) = \{(i,j) : \exists e_{ij}(t) \in \mathcal{G}, t \in T\}$ the temporal set of nodes and edges, respectively and by $\eta_i(T) = \{j : \exists \mathcal{E}(T)\}$ the neighborhood of node i in time interval T. $\eta_i^k(T)$ denotes the k-hop neighborhood. A snapshot of the temporal graph $\mathcal{G}$ at time t is the graph $\mathcal{G}(t) = (\mathcal{V}[0,t], \mathcal{E}[0,t])$ with $n(t)$ nodes.

Temporal Graph Network:
A neural model for dynamic graphs can be regarded as an encoder-decoder pair, where an encoder is a function that maps from a dynamic graph to node embeddings, and a decoder takes as input one or more node embeddings and makes a prediction based these information. e.g., node classification or edge prediction.

This work contribution:
The key contribution of this paper is a novel TGN encoder applied to a continuous-time dynamic graph (CTDG) represented as a sequence of time-stamped events and producing, for each time t, the embedding of nodes $Z(t) = (z_1(t), ..., z_n(t))$.
Core modules:

- Node feature $v_i(t)$.
- Node state $s_i(t)$, initialized as zero vector.
- Node messages $m_i(t)$ updated when an event happens e.g. interaction between node i and j.

$$m_i(t) = Msg(s_i(t^-), s_j(t^-), t, e_{ij}(t)), \;\; m_j(t) = Msg(s_j(t^-), s_i(t^-), t, e_{ij}(t)) \tag{32}$$

or Node feature transform

$$m_i(t) = Msg(s_i(t^-), v_i(t)) \tag{33}$$

Note: In batch setting, message of one node at various time are aggregated by most recent or mean. $\bar{m}_i(t) = agg(m_i(t_1), ..., m_i(t_b))$

- node state $s_i(t)$ is updated based on message and old state.

$$s_i(t) = memory(m_i(t), s_i(t^-)) \tag{34}$$

- Embedding z_i: The embedding is updated based the node's state $s_i(t)$ feature $v_i(t)$ and its neighborhood $s_j(t)$ and $v_j(t)$ and edge feature $e_{ij}(t)$.
  1. Define the neighborhood: $j \in \mathcal{N}_i^k[0,t]$. During the time interval if there is node j locates in $k-hop$ of node i, node j is one of the neighbor of node i.
  2. Update embedding

$$z_i(t) = emb(i,t) = \sum_{j \in \mathcal{N}_i^k([0,t])} h(s_i(t), s_j(t), e_{ij}(t), v_i(t), v_j(t)) \tag{35}$$

  where $h(\cdot)$ is a learnable function.

11 Dimensionality Reduction

Manifold approaches for dimensionality reduction share some similarity with graph node embedding algorithms.

Manifold learning: Many dimensionality reduction algorithms work by modeling the manifold on which the training instances lie. It relies on the manifold assumption that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold.

12 LLE

LLE (Locally Linear Embedding) is an very powerful nonlinear dimensionality reduction technique. It first measure how each training instance linearly relates to its closest neighbors and then looking for a low-dimensional representation of the training set where these local relationships are best preserved.

How does it work?
First, for each instance x, the algorithm identifies its k closest neighbors, then tries to reconstruct x as linear function of these neighbors. It finds the weight w_{ij} such that the squared distance between x_i and $\sum_{j=1}^k w_{ij}x_j$ is as small as possible. The objective function of the first step is

$$\hat{W} = \arg\min_W \sum_{i=1}^m ||x_i - \sum_{j=1}^k w_{ij}x_j||_2^2, \;\; s.t. \;\; \sum_{j=1}^k w_{ij} = 1, \; \forall i \in [m] \tag{36}$$

After this step, the weight matrix $\hat{W}$ encodes the local linear relationships between the training instances.

The second step is to map the training instances into d-dimensional space while preserves these local relationships as much as possible. If $z_i \in \mathbb{R}^d$ is the image of $x_i \in \mathbb{R}^m$, the objective function is

$$\hat{Z} = \arg\min_Z \sum_{i=1}^m ||z_i - \sum_{j=1}^k \hat{w}_{ij}z_j||_2^2 \tag{37}$$

13 SNE

Distributed Stochastic Neighbor Embedding (t-SNE) reduces dimensionality while trying to keep similar instances close and dissimilar instances apart.

- Step 1: SNE (stochastic neighbor embedding) starts to converting the high-dimensional euclidean distance between instances into conditional probability.

$$p_{ij} = \frac{\exp(-||x_i - x_j||_2^2 / 2\sigma_i^2)}{\sum_{k \neq i} \exp(-||x_i - x_k||_2^2 / 2\sigma_i^2)} \tag{38}$$

- For the low-dimensional embedding z_i, it is possible to compute the similar conditional probability

$$q_{ij} = \frac{\exp(-||z_i - z_j||_2^2)}{\sum_{k \neq i} \exp(-||z_i - z_k||_2^2)} \tag{39}$$

- SNE attempts to find Z which minimizes the KL-divergence between p_{ij} and q_{ij}.

$$\hat{Z} = \arg\min_Z \sum_i KL(P_i||Q_i) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}} \tag{40}$$

14 t-SNE

t-SNE is an extension of SNE: To measure the minimization of sum of difference of conditional probability SNE minimizes the sum of KL-divergence overall data points using a gradient descent method. It is very difficult (computationally inefficient) to optimize this cost function.

So t-SNE also tries to minimize the sum of the difference in conditional probabilities. But it does that by using the symmetric version of the SNE cost function, with simple gradients. Also, t-SNE employs a heavy-tailed distribution in the low-dimensional space to alleviate both the crowding problem (the area of the two-dimensional map that is available to accommodate moderately distant data points will not be nearly large enough compared with the area available to accommodate nearby data points) and the optimization problems of SNE. t-SNE performs a binary search for the value of σ_i.

In high dimensional space, the probability is defined as

$$p_{ij} = \frac{\exp(-||x_i - x_j||_2^2 / 2\sigma_i^2)}{\sum_{k \neq i} \exp(-||x_i - x_k||_2^2 / 2\sigma_i^2)} \tag{41}$$

In low dimensional space, let the embedding as z_i, the conditional probability is defined according to student t-distribution

$$q_{ij} = \frac{(1 + ||z_i - z_j||_2^2)^{-1}}{\sum_{k \neq l} (1 + ||y_k - y_l||_2^2)^{-1}} \tag{42}$$

The embedding is learned by minimizing the KL-divergence

$$\hat{Z} = \arg\min_Z \sum_i KL(P_i||Q_i) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}} \tag{43}$$

15 Applications

To conduce experiments, we use Pytorch-geometric. Here, we give a brief overview. Pytorch-geometric provides a collection of graph datasets and off-the-shelf GNN algorithms. The essence of Pytorch-geometric is Message-passing module which allows to design new graph operator layer. This module consists of four functions: Propagate, message, aggregate and update. Calling propagate will consequently call message, aggregate and update in sequel.

The standard procedure of processing graph data is the following

- Create PyG dataset
- Create DataLoader
- Create MessagePassing module
- Train Model
- Test Model

The Tutorial is a great example of implementing PyG models which walks you through the whole procedure.

References