Machine learning on graphs

Machine learning

- Machine learning is a problem-driven study.
- We build models that can learn from data in order to solve particular tasks.
- Supervised task, the goal is to predict a target output given an input data
- Unsupervised task, the goal is to infer patterns

Machine learning on graphs

- Machine learning with graphs is no different, but the categories of supervised and unsupervised are not necessarily informative

- In particular, the problems are categorized into four tasks, 1. Node classification, 2. Relation prediction, 3. Clustering and community detection, 4. Graph classification, regression, and clustering

Node classification

- Suppose we are given a large social network dataset with millions of users, but we know that a significant number of these users are actually bots.
- If we manually exam every user to determine if they are a bot, it would be expensive and inefficient.
- So ideally we would like to have a model that could classify users as a bot or not (0 or 1, human or bot) with some manually labeled data.
- The goal is to predict the label y (could be a type, category, or attribute) associated with its the node v and node's feature x.
- Examples of node classification beyond social networks include classifying the function of proteins in the interactome (PPI data) and classifying the topic of documents based on hyperlink or citation graphs (Citation networks).

Not really supervised learning

- Node classification appears to be a supervised task, but there are in fact important differences.
- The most important difference is that the nodes in a graph are not independent and identically distributed (does not satisfy iid assumption).
- We build supervised machine learning models based on the assumption that each datapoint is statistically independent from all the other datapoints.
- If independent is not the case, we might need to model the dependencies between all our input points.
- If identically distributed is not the case, we cannot guarantee that the model will generalize to new datapoints.
- Node classification break iid assumption, so we model an interconnected set of nodes instead of a set of iid nodes.

The key idea of Node classification

- The key insight is to explicitly leverage the connections between nodes.
- One particularly popular idea is to study and increase homophily, which is the tendency for a node to share label with its neighbors in the graph.
- For example, people tend to form friendships with others who share the same interests.
- Beyond homophily there are also concepts such as structural equivalence, the idea that nodes with similar local neighborhood structures will have similar labels, as well as heterophily, which presumes that nodes will be connected to nodes with different labels.

Supervised -> Semi-supervised

- Researchers often refer node classification as semi-supervised, only a few nodes are provided with labels for training(<1% of nodes in a graph).
- This terminology is used because when we are training for classifying nodes, we usually have access to the full graph, including all the unlabeled nodes (we are only missing the labels of unlabeled nodes but still have their features for message passing). We can still use information about the test nodes, knowledge of their neighborhood in the graph in training.
- This is different from the usual supervised setting, in which unlabeled datapoints are completely unobserved during training. The general term used for models that combine labeled and unlabeled data during traning is semi-supervised learning.

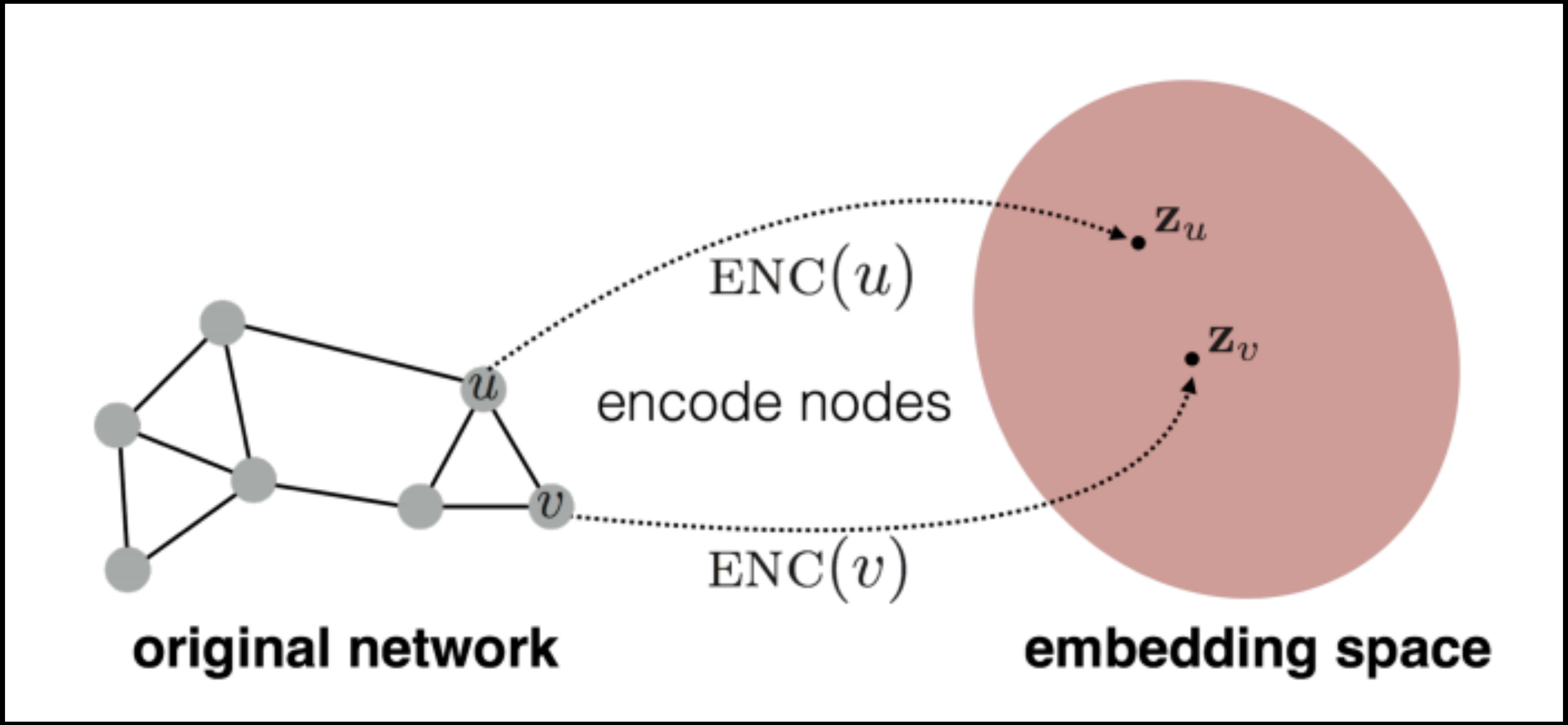
Graph classification, regression, and clustering

- For instance, given a graph representing the structure of a molecule, we might want to build a model that could predict that molecule's chemical properties.
- In these graph classification or regression applications, we seek to learn over graph data, but instead of making predictions over the individual components of a single graph.
- We are given a dataset of multiple different graphs and our goal is to make independent predictions specific to each graph.
- In the related task of graph clustering, the goal is to learn an unsupervised measure of similarity between pairs of graphs.

- Graph regression and classification are the most straightforward analogues of standard supervised learning.
- Each graph is an iid datapoint associated with a label, and the goal is to use a labeled set of training points to learn a mapping from datapoints to labels.
- Graph clustering is the straightforward extension of unsupervised clustering for graph data.

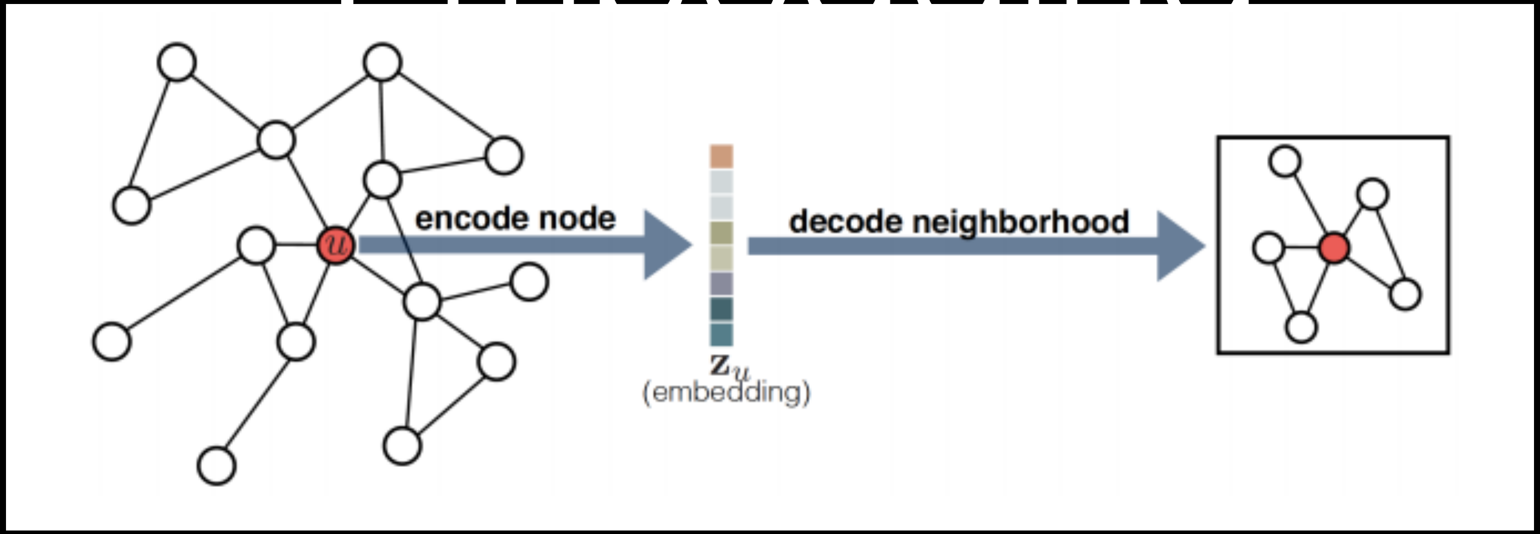
Node embedding, neighborhood reconstruction

- The goal of node embeddings is to encode nodes as low-dimensional vectors that summarize their graph position and the structure of their local graph neighborhood.
- We want a projection for nodes in latent space, where geometric relations in this latent space correspond to relationships in the origin graph.



- The goal is to learn an encoder(ENC) that maps nodes to a lower dimensional space. These embeddings are optimized so that distances in the embedding space reflect the relative positions of the nodes in the original graph.

Encoder-Decoder Embedding



- An encoder model maps each node in the graph into a low-dimensional vector or embedding

- A decoder model takes the low-dimensional node embeddings and uses them to reconstruct information about each node’s neighborhood in the original graph.

Encoder

- The encoder is a function that maps nodes v to vector embeddings z_v

- In the simplest case, the encoder takes node IDs as input to generate the node embeddings.

$$\mathrm{ENC} : \mathcal{V} \to \mathbb{R}^d$$

- In most work on node embeddings, the encoder function is simply an embedding lookup based on the node ID, called shallow embedding. Z is a matrix containing the embedding vectors for all nodes, and Z[v] denotes the row of Z corresponding to node v

$$\mathrm{ENC}(v) = \mathbf{Z}[v] \qquad \mathbf{Z} \in \mathbb{R}^{|\mathcal{V}| \times d}$$

Encoder and GNN

- For instance, the encoder can use node features or the local graph structure around each node as input to generate an embedding. The generalized encoder architectures are often called graph neural networks (GNNs).

Decoder

- Decoder is to reconstruct certain graph statistics from the node embeddings that are generated by the encoder.
- For example, given a node embedding z of a node v, the decoder might attempt to predict a set of its neighbors N(v) or its row A[v] in the adjacency.

Pairwise Decoders

- The most standard way to define a decoder is so called pairwise decoders.

- Pairwise decoders can be interpreted as predicting the relationship or similarity between pairs of nodes.

$$\mathrm{DEC} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^+$$

- For instance, a simple pairwise decoder could predict whether two nodes are neighbors in the graph.

Use of Pairwise Decoder

- Applying the pairwise decoder to a pair of embeddings (z_u,z_v) results in the reconstruction of the relationship between nodes u and v.

- The reconstruction perspective

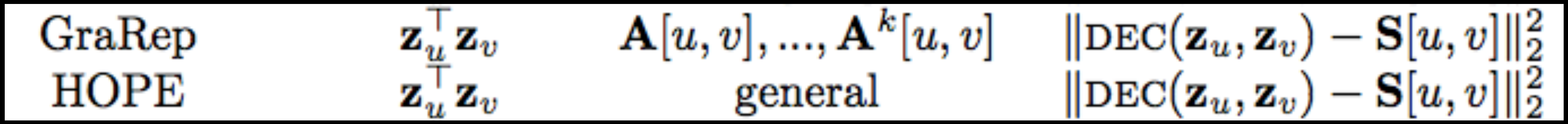
$$\text{DEC}(\text{ENC}(u), \text{ENC}(v)) = \text{DEC}(\mathbf{z}_u, \mathbf{z}_v) \approx \mathbf{S}[u, v]$$

- S[u, v] is a graph-based similarity measure between nodes.

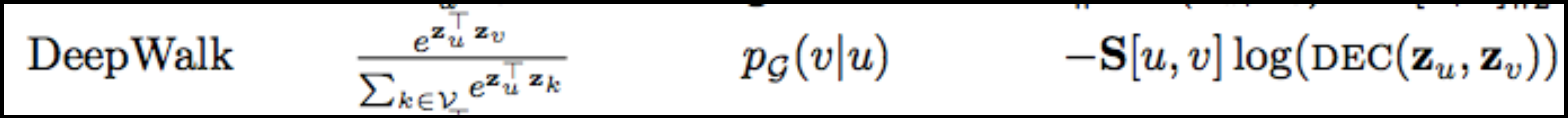
- The simplest reconstruction objective of predicting whether two nodes are neighbors would correspond to S=A. But S could be defined in more ways.

Optimizing Encoder-Decoder

- To achieve the reconstruction objective, minimize an empirical reconstruction loss L over a set of training node pairs D, where l:RxR->R is a loss function. $$\mathcal{L} = \sum_{(u,v)\in\mathcal{D}} \ell\left(\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v), \mathbf{S}[u,v]\right)$$
- l can be defined in many ways.



GraRep	$\mathbf{z}_u^\top \mathbf{z}_v$	$\mathbf{A}[u,v], ..., \mathbf{A}^k[u,v]$	$\|\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v) - \mathbf{S}[u,v]\|_2^2$
HOPE	$\mathbf{z}_u^\top \mathbf{z}_v$	general	$\|\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v) - \mathbf{S}[u,v]\|_2^2$



DeepWalk	$\frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{k \in \mathcal{V}} e^{\mathbf{z}_u^\top \mathbf{z}_k}}$	$p_{\mathcal{G}}(v\|u)$	$-\mathbf{S}[u,v] \log(\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v))$

- P(v|u) is for the probability of visiting v on a fixed-length random walk starting from u

Random walk embeddings

- DeepWalk and node2vec are kind of a recent work and still used as a baseline to compare different GNNs.
- DeepWalk and node2vec use a shallow embedding approach and an inner-product decoder.
- The key distinction in these methods is in how they define the notions of node similarity and neighborhood reconstruction.
- Instead of directly reconstructing the adjacency matrix A, these approaches optimize embeddings to encode the statistics of random walks.

$$\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v) \triangleq \frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{v_k \in \mathcal{V}} e^{\mathbf{z}_u^\top \mathbf{z}_k}} \approx p_{\mathcal{G},T}(v|u),$$

- The decoder calculates the probability of visiting v on a length T random walk starting at node u, with T usually defined in the range {2,…,10}.

- To train random walk embeddings, the general strategy is to use the decoder and minimize the cross-entropy loss,

$$\mathcal{L} = \sum_{(u,v)\in\mathcal{D}} -\log(\mathrm{DEC}(\mathbf{z}_u, \mathbf{z}_v))$$

- D denotes the training set of random walks, which is generated by sampling random walks starting from each node.

- In these approaches, the encoder model that maps nodes to embeddings is simply an embedding lookup, which trains a unique embedding for each node in the graph.
- This approach has achieved many successes in the past decade.
- However, it is also important to note that shallow embedding approaches suffer from some important drawbacks.

Limitations

- shallow embedding approaches do not leverage node features in the encoder. Many graph datasets have rich feature information, which could potentially be informative in the encoding process.
- Most importantly, these methods are transductive but not inductive, as we talked before, we are interested in semi-supervised learning with a few labeled data provided. Shallow embedding approaches can only generate embeddings for nodes that were present during the training phase. Generating embeddings for new nodes is not possible unless additional optimizations are performed to learn the embeddings for these nodes.

Beyond shallow encoding

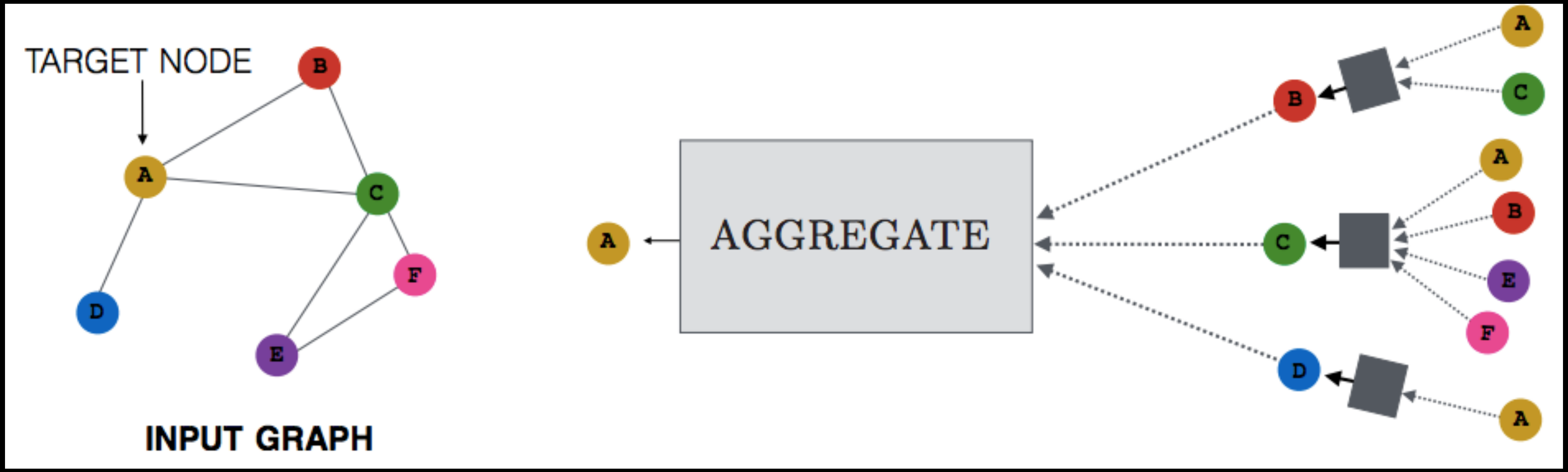
- To alleviate these limitations, shallow encoders can be replaced with more sophisticated encoders that depend more generally on the structure and attributes of the graph, GNN is introduced.

Neural Message Passing

- GNN is all about message passing.
- The defining feature of a GNN is that it uses a form of neural message passing in which vector messages are exchanged between nodes and updated using neural networks
- Beyond shallow embedding, we can take an input graph G along with a set of node features X, and use this information to generate node embeddings Z.

Message passing framework

- During each message-passing iteration in a GNN, a hidden embedding h_u corresponding to each node u is updated according to information aggregated from v's graph neighborhood N(u).



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

- UPDATE and AGGREGATE are two differentiable functions.
- MESSAGE means aggregated information from u's graph neighborhood N(u).
- At each iteration k of the GNN, the AGGREGATE function takes as input the set of embeddings of the nodes in u's graph neighborhood N(u) and generates a MESSAGE based on this aggregated neighborhood information.
- The update function UPDATE then combines the MESSAGE with the previous embedding.
- The initial embeddings at k = 0 are set to the input features X for all the nodes.
- After running K iterations of the GNN message passing, we can use the output of the final layer to define the embeddings Z for all nodes.

Simple intuition

- At each iteration, every node aggregates information from its local neighborhood, and as these iterations progress each node embedding contains more and more information from further reaches of the graph.

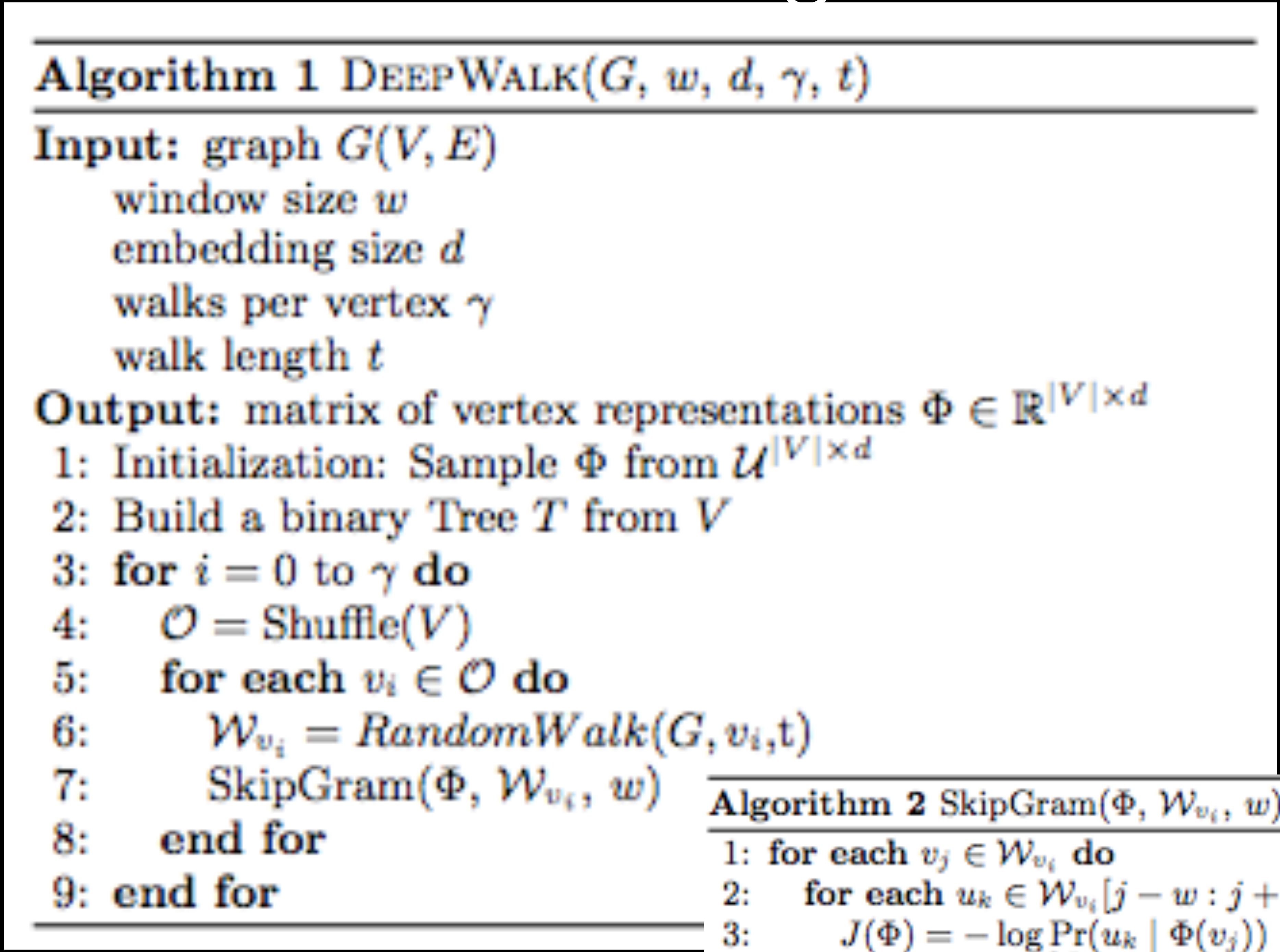
- After the first iteration (k = 1), every node embedding contains information from its 1-hop neighborhood; and every node embedding contains information from its k-hop neighborhood after the k iteration, and so on.

Why informative

- Generally, this information comes in two forms.
- On the one hand there is structural information about the graph, the embedding of node u might encode information about the degrees of all the nodes in u’s k-hop neighborhood after k iteration.
- The other key kind of information captured by GNN node embedding is feature-based. After k iterations of GNN message passing, the embeddings for each node also encode information about all the features in their k-hop neighborhood.

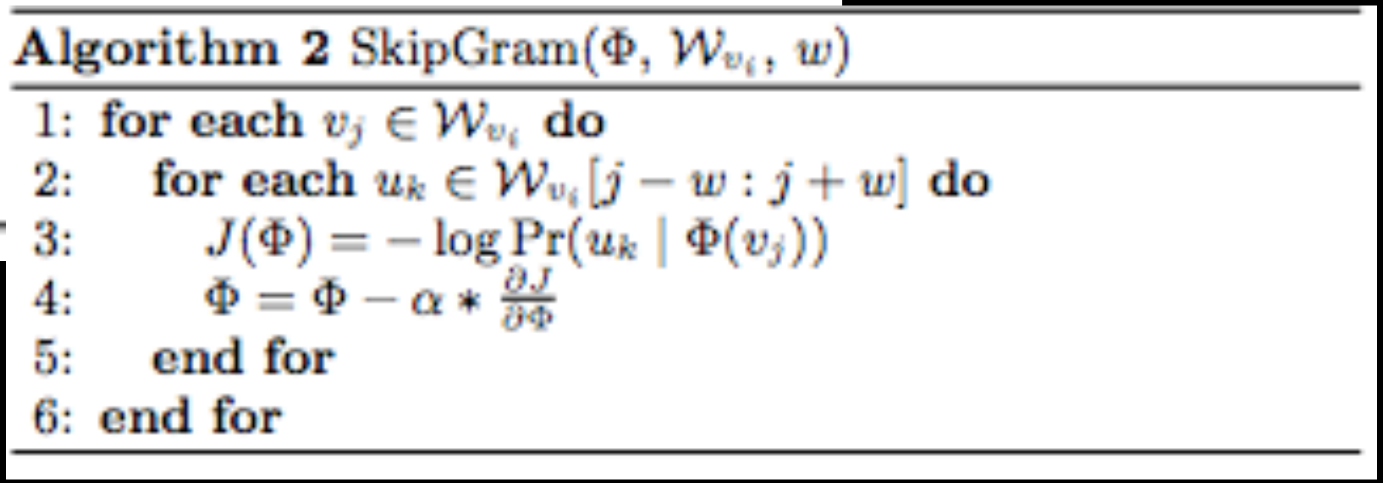
- Continue…

Shallow embedding without feature



Algorithm 1 DeepWalk(G, w, d, γ, t)

```
Input: graph $G(V, E)$
    window size $w$
    embedding size $d$
    walks per vertex $\gamma$
    walk length $t$
Output: matrix of vertex representations $\Phi \in \mathbb{R}^{|V| \times d}$
1: Initialization: Sample $\Phi$ from $\mathcal{U}^{|V| \times d}$
2: Build a binary Tree $T$ from $V$
3: for $i = 0$ to $\gamma$ do
4:     $\mathcal{O} = \text{Shuffle}(V)$
5:     for each $v_i \in \mathcal{O}$ do
6:         $\mathcal{W}_{v_i} = RandomWalk(G, v_i, \text{t})$
7:         SkipGram($\Phi$, $\mathcal{W}_{v_i}$, $w$)
8:     end for
9: end for
```



Algorithm 2 SkipGram(Φ, $\mathcal{W}_{v_i}$, w)

```
1: for each $v_j \in \mathcal{W}_{v_i}$ do
2:     for each $u_k \in \mathcal{W}_{v_i}[j - w : j + w]$ do
3:         $J(\Phi) = -\log \Pr(u_k \mid \Phi(v_j))$
4:         $\Phi = \Phi - \alpha * \frac{\partial J}{\partial \Phi}$
5:     end for
6: end for
```

The Basic GNN

- In order to translate the abstract GNN framework defined in previous message-passing equation into something we can implement, we give concrete instantiations to these UPDATE and AGGREGATE functions.

- We begin here with the most basic GNN framework, which is a simplification of the original GNN models proposed in 2005.

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

- W_self and W_neigh are trainable parameter matrices and sigma() denotes an element-wise activation function(non-linearity). The bias term b is not necessary, but including the bias term can be important to achieve strong performance.

- The message passing in the basic GNN framework is analogous to a standard multi-layer perceptron (MLP), as it relies on linear operations followed by a single element-wise non-linearity.

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v,$$

$$\text{UPDATE}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \sigma\left(\mathbf{W}_{\text{self}}\mathbf{h}_u + \mathbf{W}_{\text{neigh}}\mathbf{m}_{\mathcal{N}(u)}\right)$$

where we recall that we use

$$\mathbf{m}_{\mathcal{N}(u)} = \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})$$

GNN with self-loop

- As a simplification of the neural message passing approach, it is common to add self-loops to the input graph and omit the explicit update step. In this approach we define the message passing simply as

$$\mathbf{h}_u^{(k)} = \text{AGGREGATE}(\{\mathbf{h}_v^{(k-1)}, \forall v \in \mathcal{N}(u) \cup \{u\}\}),$$

- Now the aggregation is taken over the set N(u) UNION {u}, i.e., the node'sneighbors as well as the node itself.

- The benefit of this approach is that we no longer need to define an explicit update function, as the update is implicitly defined through the aggregation method.

Benefits vs. Drawbacks

- Simplifying the message passing in this way can often alleviate overfitting,
- But it also severely limits the expressivity of the GNN, as the information coming from the node’s neighbours cannot be differentiated from the information from the node itself.
- Problem: when we add new nodes to the graph (with new edges), we have to train the model for the whole new graph again. If we want our model to fit inductive learning, we have to explicitly state W_self and W_neigh.

- In the case of the basic GNN, adding self-loops is equivalent to sharing parameters between the W_self and W_neigh matrices, which gives the following graph-level update:

$$\mathbf{H}^{(t)} = \sigma\left((\mathbf{A}+\mathbf{I})\mathbf{H}^{(t-1)}\mathbf{W}^{(t)}\right)$$

Neighborhood Normalization

- One issue with previous approach is that it can be unstable and highly sensitive to node degrees.
- For instance, suppose node u has 100x as many neighbors as node u', then we would reasonably expect that $\|\sum_{v \in \mathcal{N}(u)} \mathbf{h}_v\| >> \|\sum_{v' \in \mathcal{N}(u')} \mathbf{h}_{v'}\|$
- This drastic difference in magnitude can lead to numerical instabilities as well as difficulties for optimization.

What we do

- One solution to this problem is to simply normalize the aggregation operation based upon the degrees of the nodes involved. The simplest approach is to just take an average rather than sum:

$$\mathbf{m}_{\mathcal{N}(u)} = \frac{\sum_{v \in \mathcal{N}(u)} \mathbf{h}_v}{|\mathcal{N}(u)|},$$

- In 2016, researchers have also found success with other normalization factors, such as the following symmetric normalization:

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}.$$

- For example, in a citation graph, information from very high-degree nodes (i.e., papers that are cited many times) may not be very useful for inferring community membership in the graph, since these papers can be cited thousands of times across diverse subfields.

- Moreover, combining the symmetric-normalized aggregation along with the basic GNN update function results in a first-order approximation of a spectral graph convolution.

Graph convolutional networks

- One of the most popular baseline graph neural network models—the graph convolutional network (GCN)—employs the symmetric-normalized aggregation as well as the self-loop update approach. The GCN model thus defines the message passing function as

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\right)$$

Two-layer GCN

- In the following, we consider a two-layer GCN for semi-supervised node classification on a graph with a symmetrically normalized adjacency matrix. The model then takes the simple form:

$$Z = f(X, A) = \mathrm{softmax}\Big(\hat{A}\ \mathrm{ReLU}\Big(\hat{A}XW^{(0)}\Big)\, W^{(1)}\Big)$$

Normalization?

- Proper normalization can be essential to achieve stable and strong performance when using a GNN.
- However, normalization can also lead to a loss of information.
- For example, after normalization, it can be hard (or even impossible) to use the learned embeddings to distinguish between nodes of different degrees, and various other structural graph features can be obscured by normalization.
- Usually, normalization is most helpful in tasks where node feature information is far more useful than structural information, or where there is a very wide range of node degrees that can lead to instabilities during optimization.

Aggregators, set poolings

- Perhaps, there is something more sophisticated than just summing over the neighbor embeddings.
- The neighborhood aggregation operation is fundamentally a set function where we map a set of given neighbor embeddings to a single vector MESSAGE.
- The fact that the embeddings are a set is quite important, there is no natural ordering of a nodes' neighbors, and any aggregation function we define must thus be permutation invariant.
- Permutation invariant means no matter how we permute the elements in a set, the output, MESSAGE, is the same.
- It is possible to replace the sum with an alternative reduction function, such as an element-wise maximum or minimum.

Neighborhood Attention

- In addition to more general forms of set aggregation, a popular strategy for improving the aggregation layer in GNNs is to apply attention.

- The idea is to assign an attention weight/ importance to each neighbor, which is used to weigh this neighbor’s influence during the aggregation step.

Graph Attention Network

- The first GNN model to apply this style of attention is Graph Attention Network in 2018, which uses attention weights to define a weighted sum of the neighbors: $\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \alpha_{u,v} \mathbf{h}_v$
- a_u,v denotes the attention on neighbor v in N(u) when we are aggregating information at node u.
- The attention weights are defined as

$$\alpha_{u,v} = \frac{\exp\left(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_v]\right)}{\sum_{v' \in \mathcal{N}(u)} \exp\left(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_{v'}]\right)}$$

- a is a trainable attention vector and W is a trainable matrix.

Graph Attention Network

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

- Once obtained, the normalized attention coefficients are used to compute a linear combination of the features corresponding to them, to serve as the final output features for every node:

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

Multi-head attention

- To stabilize the learning process of self-attention, they extend their mechanism to employ multi-head attention:
- Specifically, K independent attention mechanisms execute the transformation of previous equation and then their features are concatenated, resulting in the following output feature representation:

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

- W^k is the corresponding input linear transformation’s weight matrix by y the k-th attention mechanism.
- h’ will consist of KF’ features (rather than F’) for each node

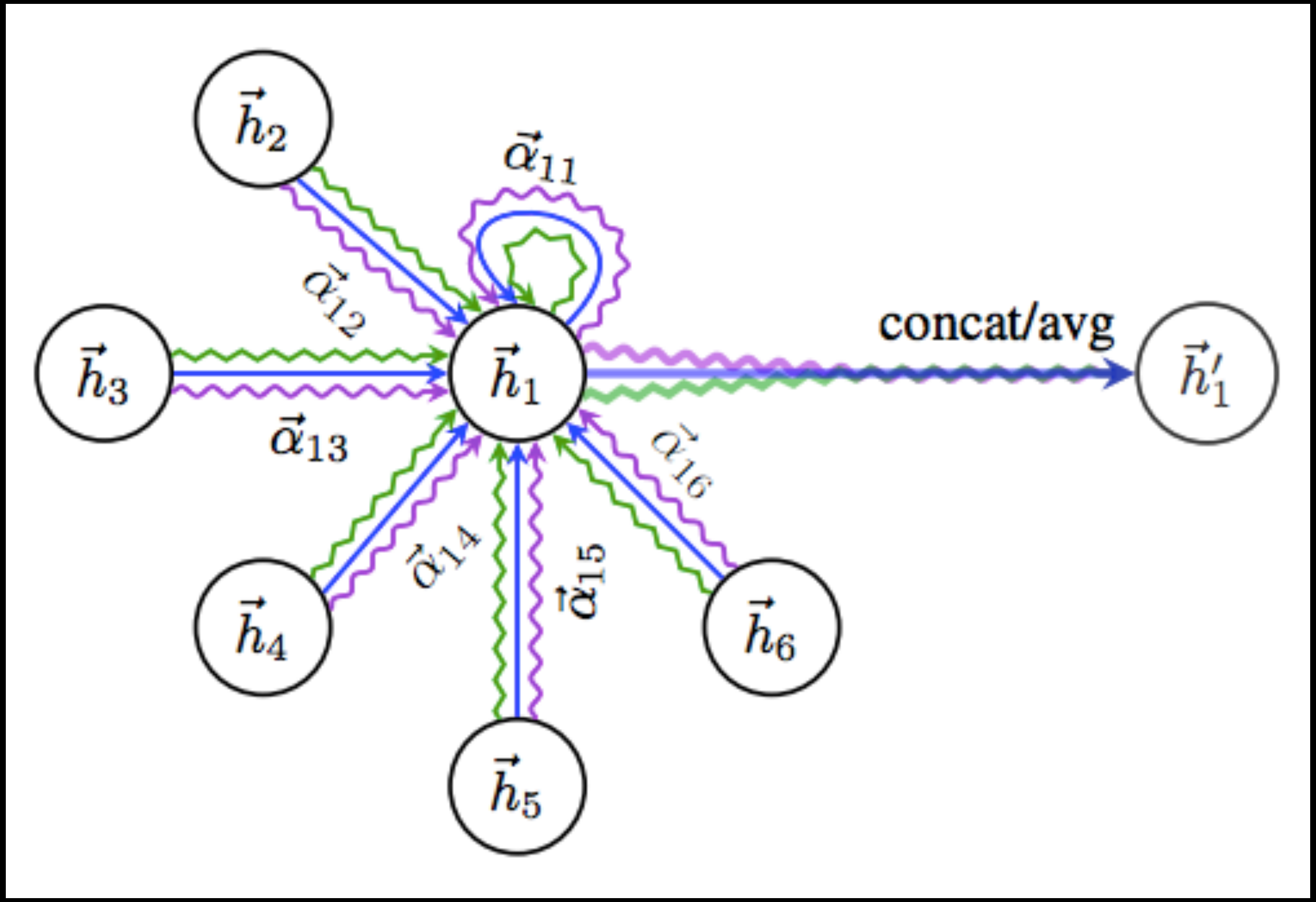
Output layer

- Specially, if we perform multi-head attention on the final (prediction) layer of the network, concatenation is no longer sensible.

- So employ averaging:

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

- The final nonlinearity is usually a softmax or logistic sigmoid for classification problems.



- An illustration of multihead attention by node 1 on its neighborhood (with K = 3 heads).

- The aggregated features from each head are concatenated or averaged to obtain h'_1

- So far, graph convolutional networks (GCNs) and GAT have only been applied in the transductive setting with fixed graphs

Why not transductive learning

- Most existing approaches require that all nodes in the graph are present during training of the embeddings; these previous approaches are inherently transductive and do not naturally generalize to unseen nodes.

- Transductive methods make predictions on nodes in a single fixed graph, these approaches do not naturally generalize to unseen data.

- When a new node is added to existing ones, it needs to be rerun to generate an embedding for the newcomer.

Inductive learning on large graphs

- The inductive node embedding problem is especially difficult, compared to the transductive setting, because generalizing to unseen nodes requires “aligning” newly observed subgraphs to the node embeddings that the algorithm has already optimized on.

- An inductive framework must learn to recognize structural properties of a node’s neighborhood that reveal both the node’s local role in the graph, as well as its global position.

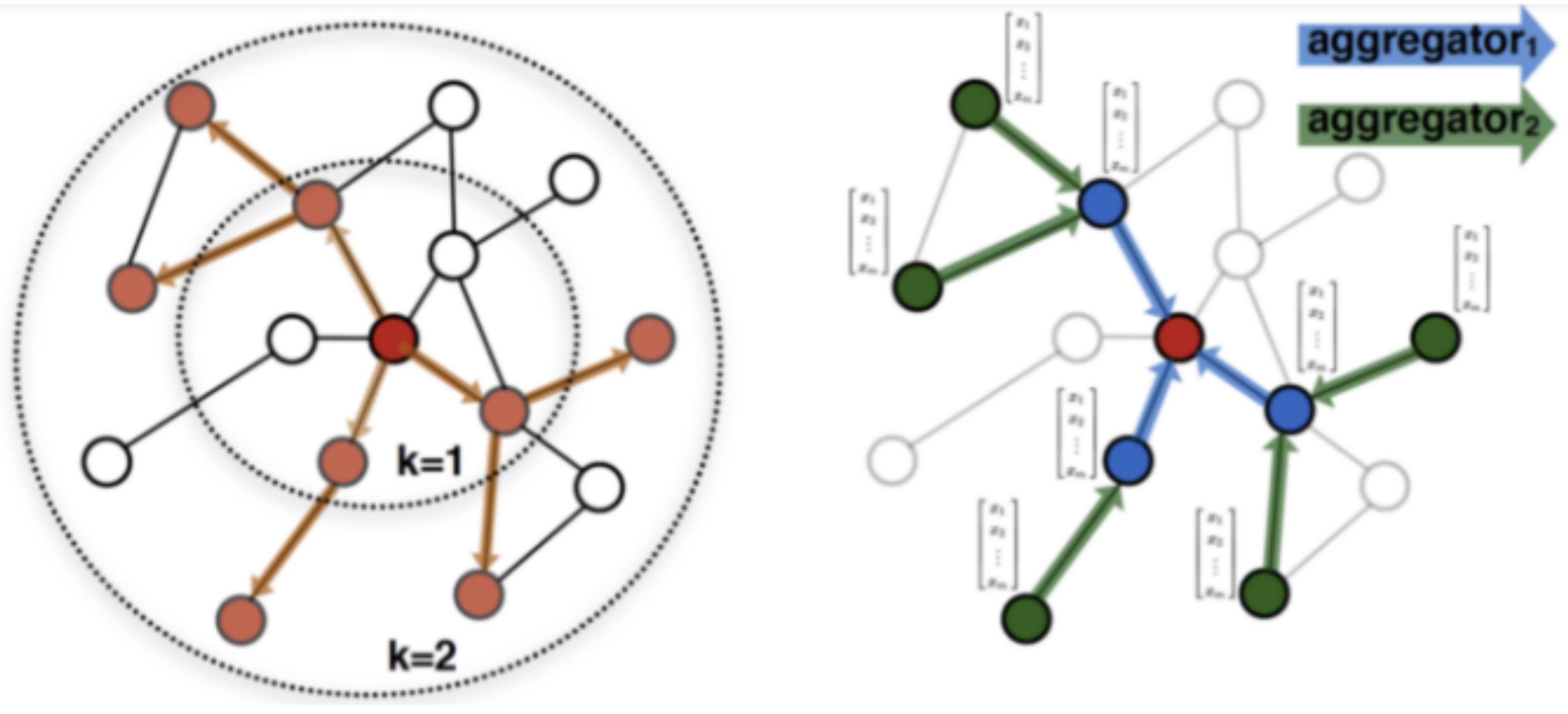
- Here GraphSAGE, a general inductive framework that leverages node feature information to efficiently generate node embeddings for previously unseen data.

- GraphSAGE is capable of predicting embedding of a new node, without requiring a re-training procedure.

- To do so, GraphSAGE learns aggregator functions that can induce the embedding of a new node given its features and neighborhood. This is called inductive learning.

- GraphSAGE assumes that nodes that reside in the same neighborhood should have similar embeddings.

- Similar to DeepWalk, the algorithm has a parameter K that controls the neighborhood depth. If K is 1, only the adjacent nodes are accepted as similar. If K is 2, the nodes at distance 2 are seen in the same neighborhood as well.

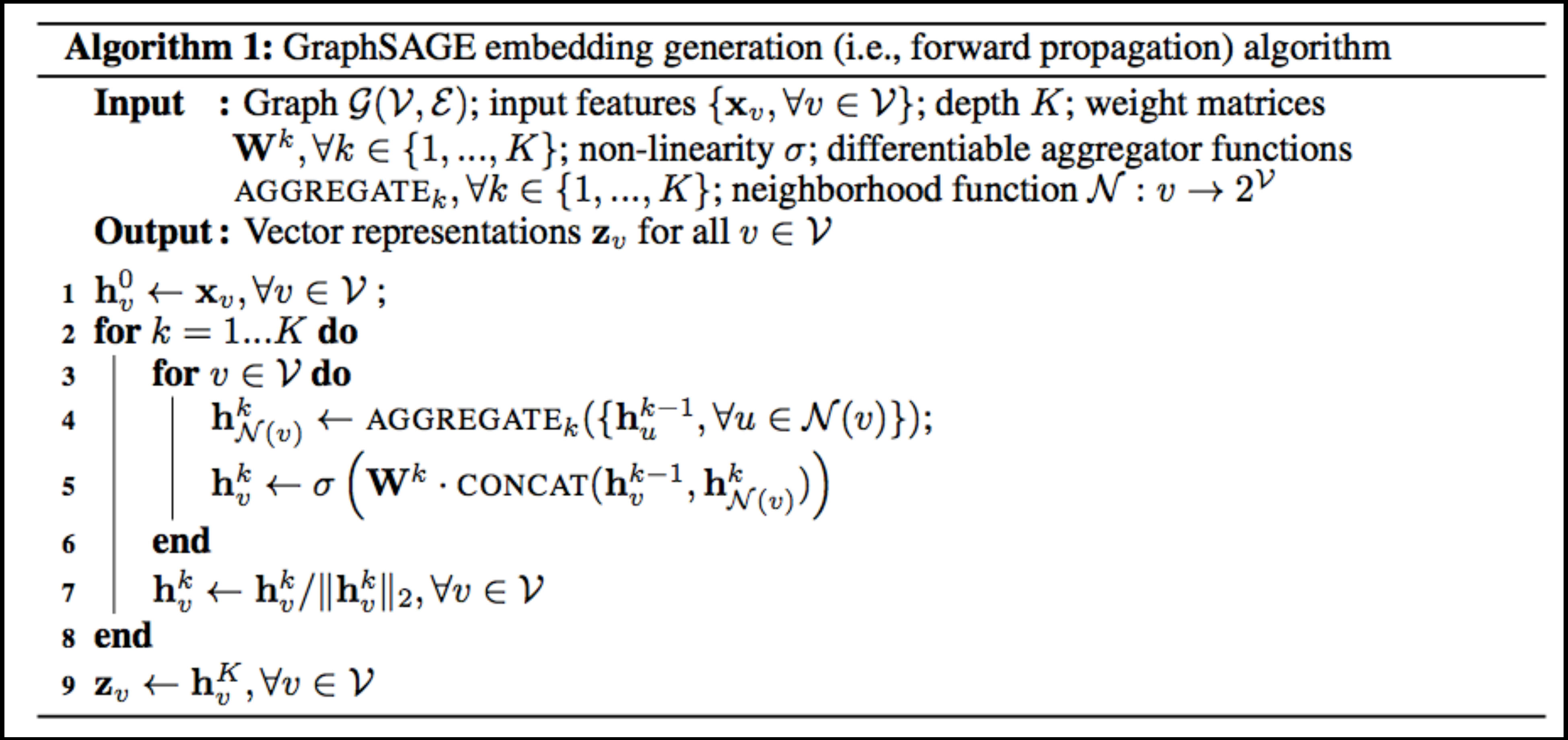


Neighborhood exploration and information sharing in GraphSAGE. [1]

Information Aggregation

- Aggregation functions or aggregators accept a neighborhood as input and combine each neighbor’s embedding with weights to create a neighborhood embedding.

GraphSAGE



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 $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$
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 $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$
```

- Algorithm 1 describes the embedding generation process in the case where the entire graph
- The aggregation of the neighbor representations can be done by a variety of aggregator architectures.
- Aggregator weights are either learned or fixed depending on the function.

- The advantage of learning aggregator functions to generate node embeddings, instead of learning the embeddings themselves, is inductivity.

- When the aggregator weights are learned, the embedding of an unseen node can be generated from its features and neighborhood.

- As a result, aggregators remove the necessity of re-training when new nodes are introduced to the graph. Note that this is quite common in social networks, web, citation networks and so on.

Aggregator Architectures

- Unlike machine learning over N-D lattices (e.g., sentences, images), , a node’s neighbors have no natural ordering; thus, the aggregator functions in Algorithm 1 must operate over an unordered set of vectors.
- Aggregators must be permutation invariant, while still being trainable and maintaining high representational capacity.

Mean aggregator

- They simply take the element-wise mean of the vectors in {h_u, for all u in N(v)}. The mean aggregator is nearly equivalent to the convolutional propagation rule used in the transductive GCN framework.

$$\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W} \cdot \text{MEAN}(\{\mathbf{h}_v^{k-1}\} \cup \{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\}).$$

- It is a rough, linear approximation of a localized spectral convolution.

Pooling aggregator

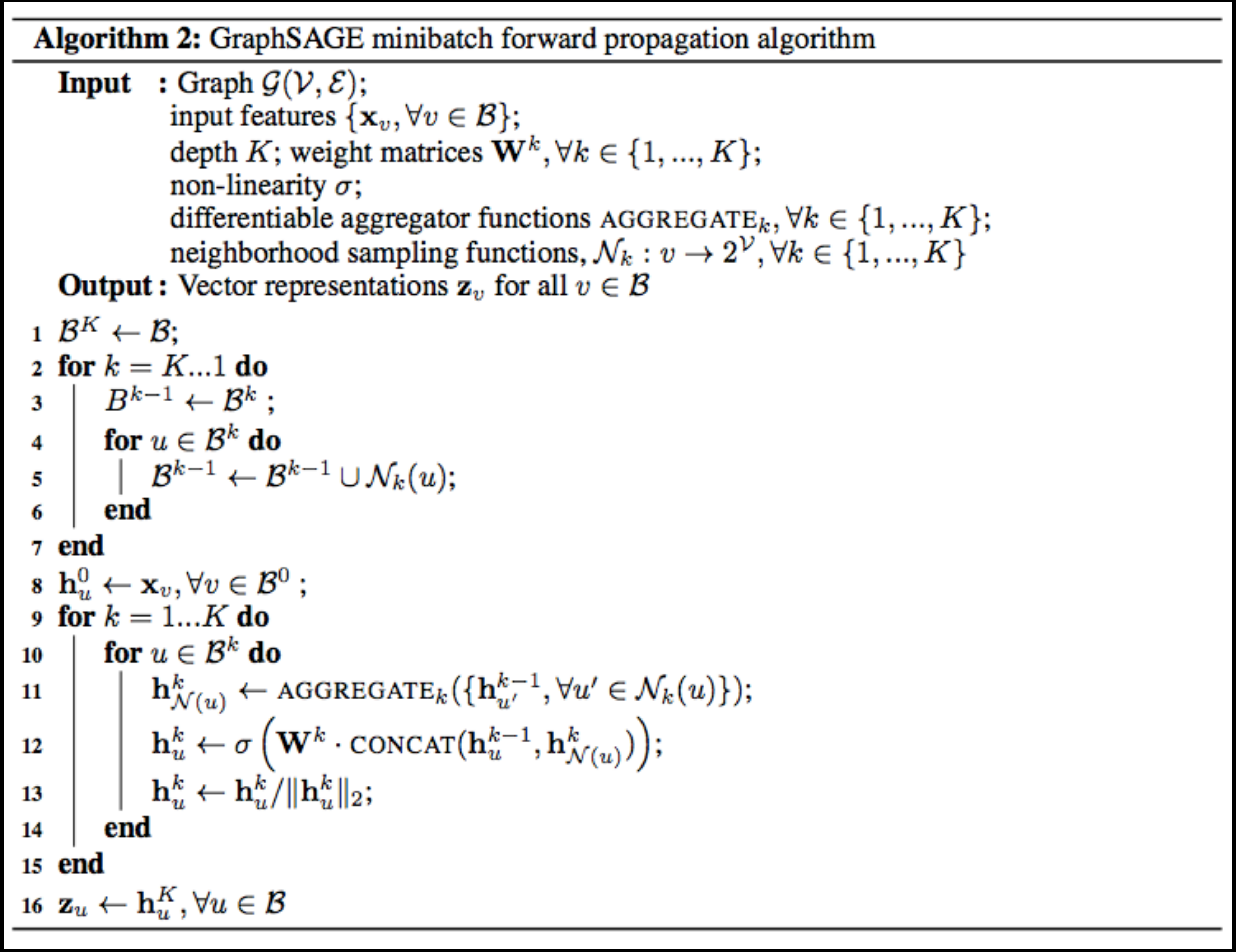
- Pooling aggregator is trainable.
- In this pooling approach, each neighbor’s vector is independently fed through a fully-connected neural network; following this transformation, an elementwise max-pooling operation is applied to aggregate information across the neighbor set: $\text{AGGREGATE}_k^{\text{pool}} = \max(\{\sigma\left(\mathbf{W}_{\text{pool}}\mathbf{h}_{u_i}^{k} + \mathbf{b}\right), \forall u_i \in \mathcal{N}(v)\})$
- max denotes the element-wise max operator and σ is a nonlinear activation function.
- By applying the max-pooling operator to each of the computed features, the model effectively captures different aspects of the neighborhoo.

Full batch to mini batch

- To extend Algorithm 1 to the minibatch setting(instead of computing the entire graph at once), given a set of input nodes,
- First sample the required neighborhood sets (up to depth K).
- Then run the inner loop (line 3 in Algorithm 1), but instead of iterating over all nodes, compute only the representations that are necessary to satisfy the recursion at each depth.
- Practically speaking they found that their approach could achieve high performance with $K = 2$ and $S\_k1 \cdot S\_k2 \leq 500$

GraphSAGE Neighborhood definition

- In their work, they uniformly sample a fixed-size set of neighbors, instead of using full neighborhood sets in Algorithm 1.
- Define N(v) as a fixed-size, uniform draw from the set {u in V | (u, v) in E}, and draw different uniform samples at each iteration, k, in Algorithm 1.
- Mini-batch saves computational space and time.



Algorithm 2: GraphSAGE minibatch forward propagation algorithm

```
Input  : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$;
         input features $\{\mathbf{x}_v, \forall v \in \mathcal{B}\}$;
         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 sampling functions, $\mathcal{N}_k : v \to 2^{\mathcal{V}}, \forall k \in \{1, ..., K\}$
Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{B}$
1  $\mathcal{B}^K \leftarrow \mathcal{B}$;
2  for $k = K...1$ do
3      $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^k$ ;
4      for $u \in \mathcal{B}^k$ do
5          $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^{k-1} \cup \mathcal{N}_k(u)$;
6      end
7  end
8  $\mathbf{h}_u^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{B}^0$ ;
9  for $k = 1...K$ do
10     for $u \in \mathcal{B}^k$ do
11         $\mathbf{h}_{\mathcal{N}(u)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_{u'}^{k-1}, \forall u' \in \mathcal{N}_k(u)\})$;
12         $\mathbf{h}_u^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_u^{k-1}, \mathbf{h}_{\mathcal{N}(u)}^k)\right)$;
13         $\mathbf{h}_u^k \leftarrow \mathbf{h}_u^k / \|\mathbf{h}_u^k\|_2$;
14     end
15 end
16 $\mathbf{z}_u \leftarrow \mathbf{h}_u^K, \forall u \in \mathcal{B}$
```

- The main idea is to sample all the nodes needed for the computation first.
- Each set B^k contains the nodes that are needed to compute the representations of nodes v ∊ B^(k+1)

- Continue…

GNN in Practice

- We will discuss a representative application and how GNNs are generally optimized in practice and introduce common techniques used to regularize and improve the efficiency of GNNs.

Node classification

- Node classification is one of the most popular benchmark tasks for GNNs as introduced above.
- The standard way to apply GNNs to such a node classification task is to train GNNs in a fully-supervised manner, where we define the loss using a softmax classification function and negative log-likelihood loss.

Loss function

$$L = \Sigma_{u \in V_{train}} - log(softmax(z_u, y_u))$$

$$softmax(z_u, y_u) = \Sigma_{i=1}^{c} y_u[i] \frac{e^{z_u^T w_i}}{\Sigma_{j=1}^{c} e^{z_u^T w_j}},$$

- where z_u is the output embedding by the final layer of a GNN, y_u denotes the the real label in one-hot encoded vector for a node u, and w_i,w_j are trainable parameters.

- We will assume that the gradient of the loss is backpropagated through the parameters of the GNN using stochastic gradient descent.

Subsampling and Mini-batching

In order to limit the memory footprint of a GNN and facilitate mini-batch training, one can work with a subset of nodes during message passing.

Mathematically, we can think of this as running the node-level GNN equations for a subset of the nodes in the graph in each batch.

Redundant computations can be avoided through careful engineering to ensure that we only compute the embedding h_u for each node u in the batch at most once when running the model.

The Weisfieler-Lehman Algorithm

The WL approach is more broadly known as one of the most successful and well-understood frameworks for approximate isomorphism testing.

The simplest version of the WL algorithm—commonly known as the 1-WL—consists of the following 4 steps.

step1

- Given two graphs, G_1 and G_2, we assign an initial label $l^{(0)}_{\mathcal{G}_i}(v)$ to each node in each graph.
- In most graphs, this label is simply the node degree, but if we have discrete features associated with nodes, then we can use these features to define the initial labels.

step2

Next, we iteratively assign a new label to each node in each graph by hashing the multi-set of the current labels within the node's neighborhood, as well as the node's current label:

$$l_{\mathcal{G}_i}^{(i)}(v) = \text{HASH}(l_{\mathcal{G}_i}^{(i-1)}(v), \{\{l_{\mathcal{G}_i}^{(i-1)}(u)\ \forall u \in \mathcal{N}(v)\}\}),$$

step3

We repeat Step 2 until the labels for all nodes in both graphs converge,

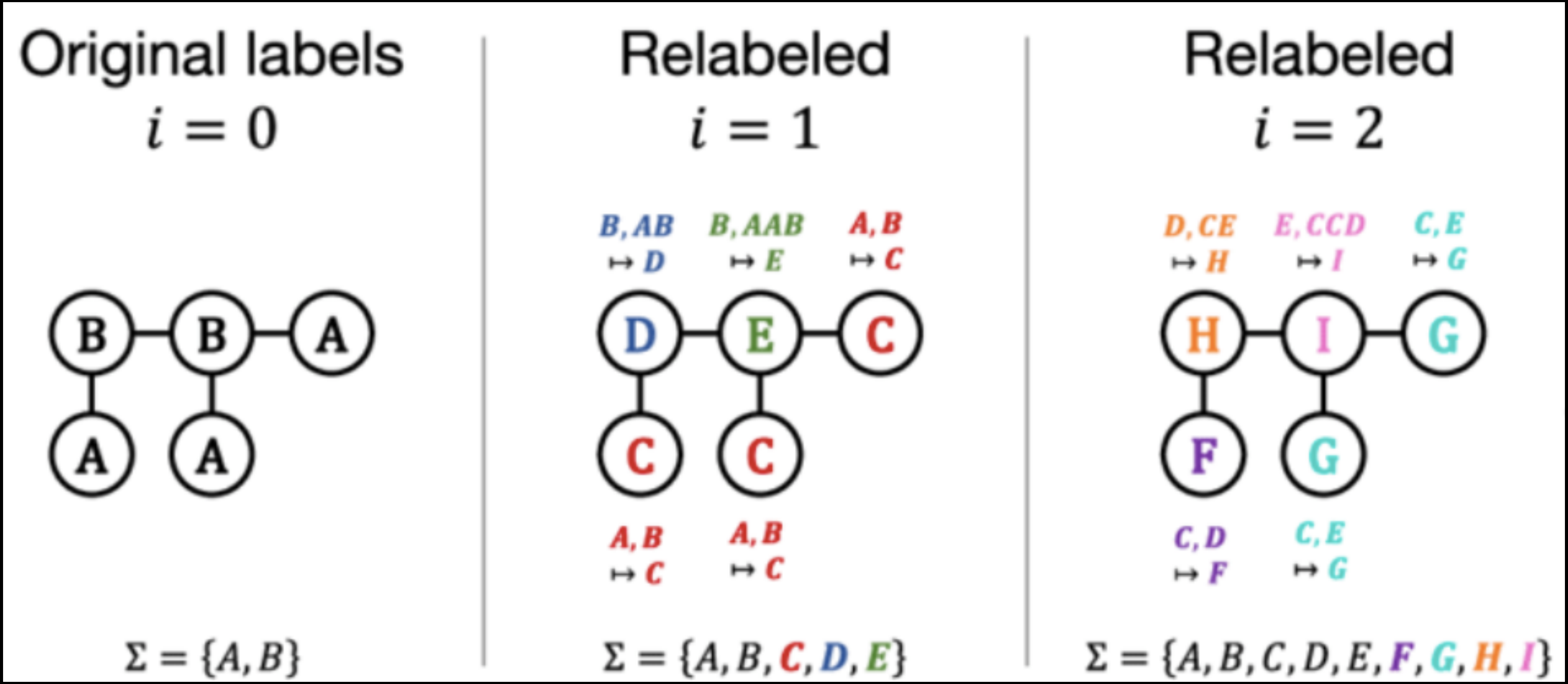
$$l_{\mathcal{G}_j}^{(K)}(v) = l_{\mathcal{G}_i}^{(K-1)}(v), \forall v \in V_j, j = 1, 2.$$

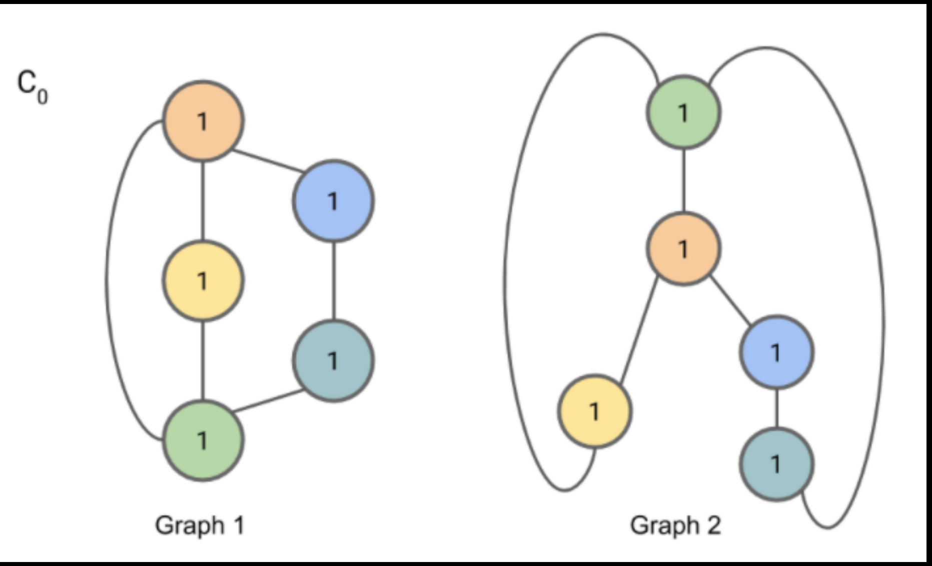
step4

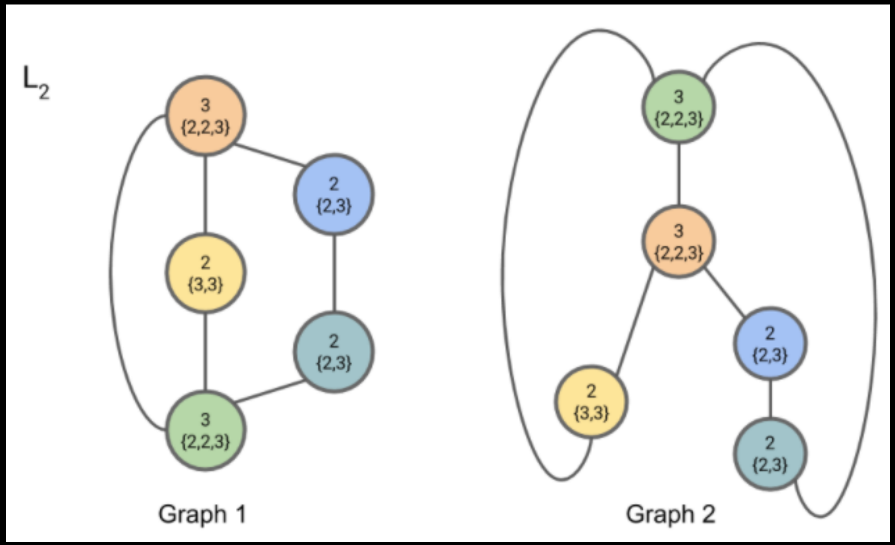
Finally, we construct multi-sets

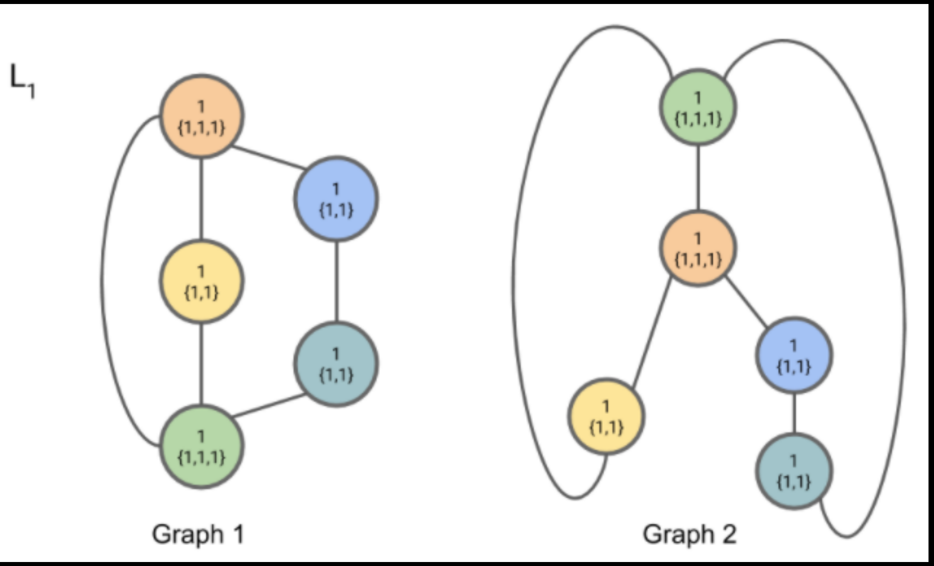
$$L_{\mathcal{G}_j} = \{\{l_{\mathcal{G}_j}^{(i)}(v), \forall v \in \mathcal{V}_j, i = 0, ..., K-1\}\}$$

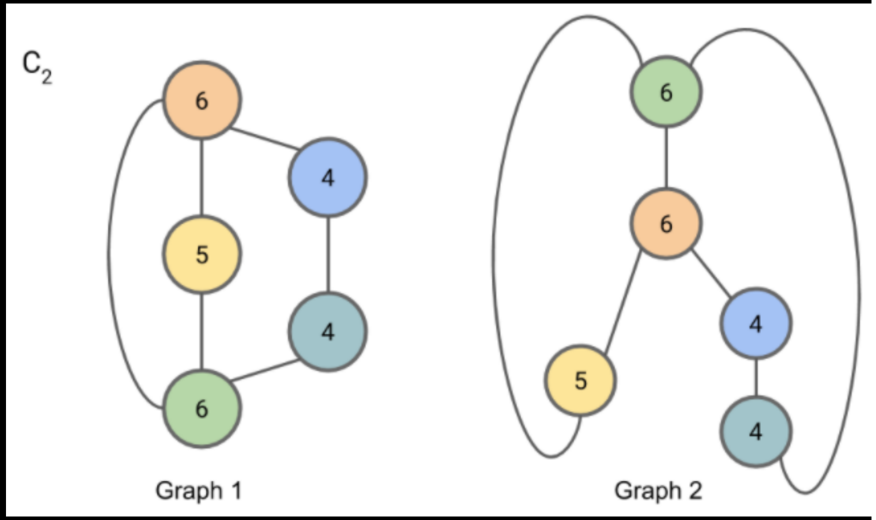
We declare G_1and G_2 to be isomorphic if and only if the multi-sets for both graphs are identical.

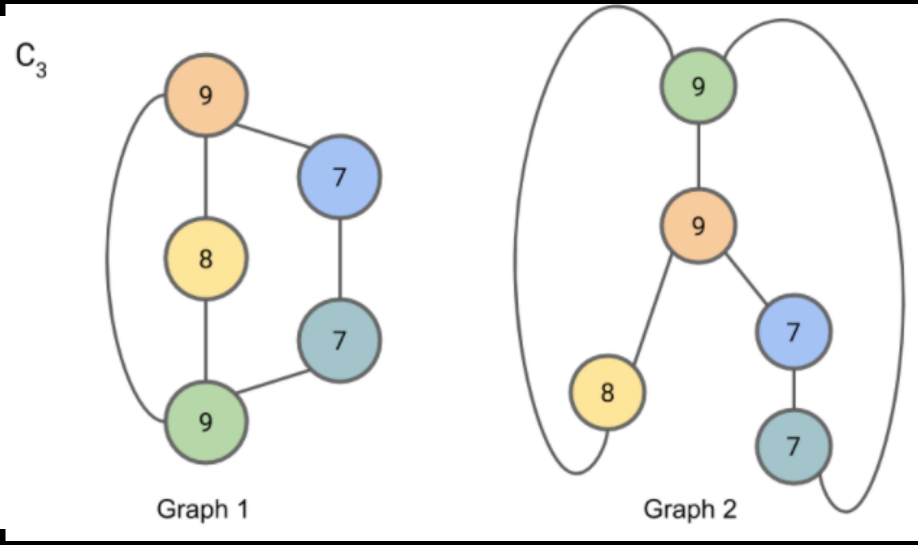

Original labels
$i = 0$
B
B
A
A
A
$\Sigma = \{A, B\}$
Relabeled
$i = 1$
$B, AB \mapsto D$
$B, AAB \mapsto E$
$A, B \mapsto C$
D
E
C
C
C
$A, B \mapsto C$
$A, B \mapsto C$
$\Sigma = \{A, B, C, D, E\}$
Relabeled
$i = 2$
$D, CE \mapsto H$
$E, CCD \mapsto I$
$C, E \mapsto G$
H
I
G
F
G
$C, D \mapsto F$
$C, E \mapsto G$
$\Sigma = \{A, B, C, D, E, F, G, H, I\}$

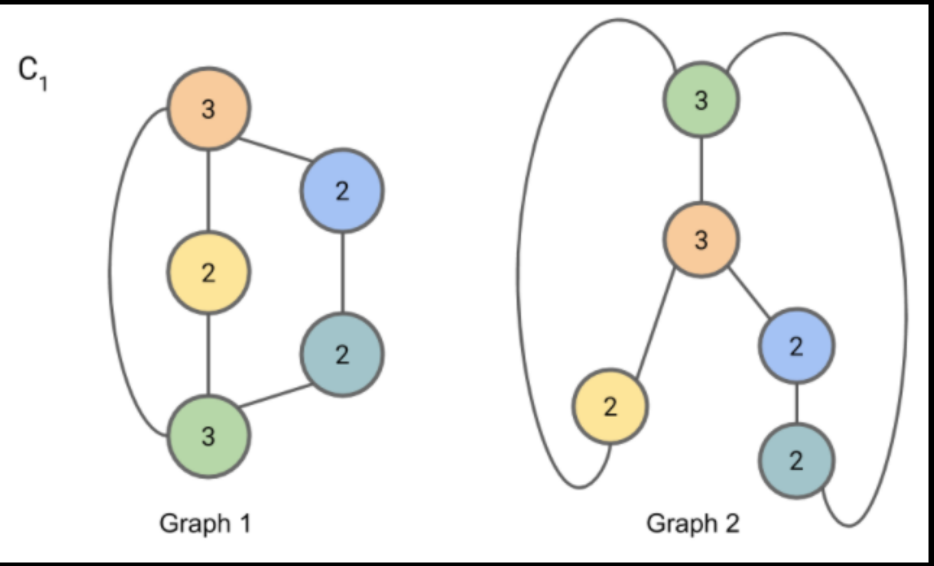

C0
1
1
1
1
1
Graph 1
1
1
1
1
1
Graph 2

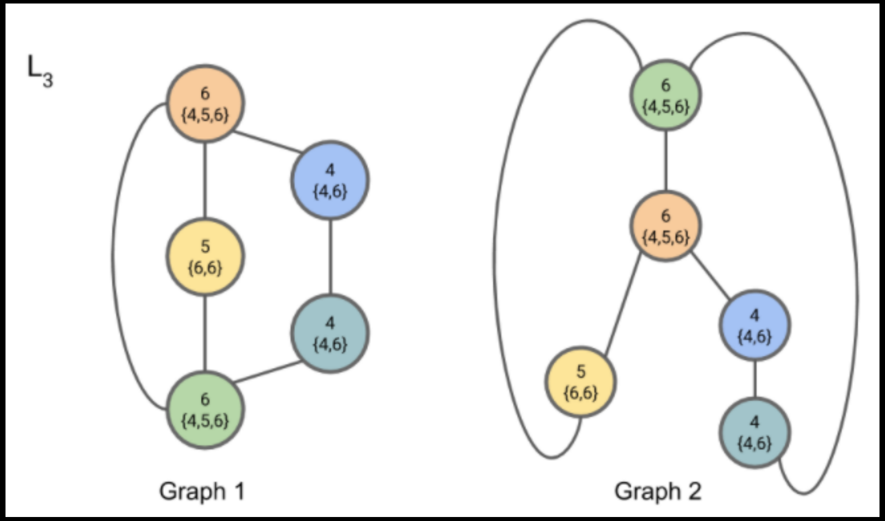

L2
3 {2,2,3}
2 {2,3}
2 {3,3}
2 {2,3}
3 {2,2,3}
Graph 1
3 {2,2,3}
3 {2,2,3}
2 {2,3}
2 {3,3}
2 {2,3}
Graph 2


L1
1 {1,1,1}
1 {1,1}
1 {1,1}
1 {1,1}
1 {1,1,1}
Graph 1
1 {1,1,1}
1 {1,1,1}
1 {1,1}
1 {1,1}
1 {1,1}
Graph 2


C2
6
4
5
4
6
Graph 1
6
6
4
5
4
Graph 2

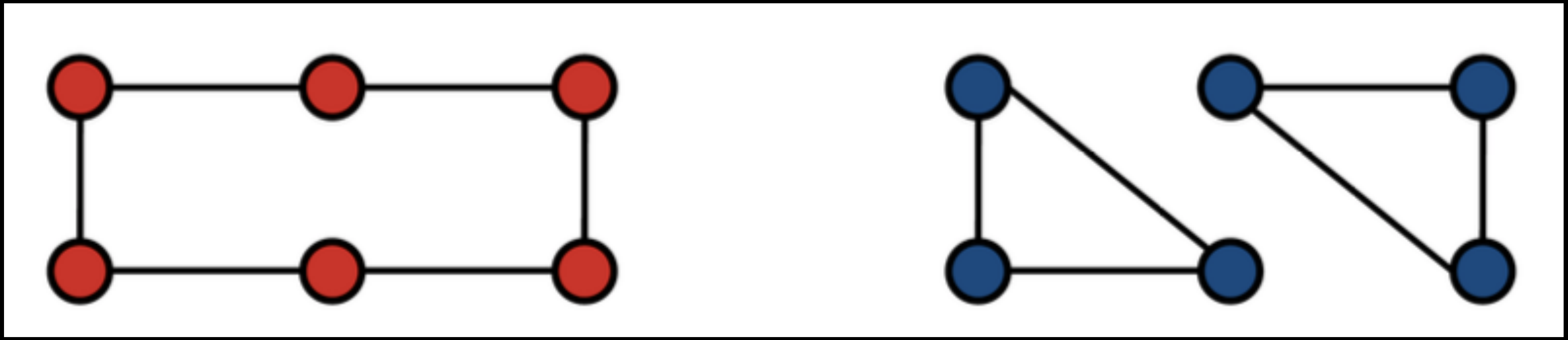

C3
9
7
8
7
9
Graph 1
9
9
7
8
7
Graph 2


C1
3
2
2
2
3
Graph 1
3
3
2
2
2
Graph 2


L3
6 {4,5,6}
4 {4,6}
5 {6,6}
4 {4,6}
6 {4,5,6}
Graph 1
6 {4,5,6}
6 {4,5,6}
4 {4,6}
5 {6,6}
4 {4,6}
Graph 2

The WL algorithm is known to converge in at most |V| iterations and is known to known to successfully test isomorphism for a broad class of graphs.

There are, however, well known cases where the test fails, such as the simple example

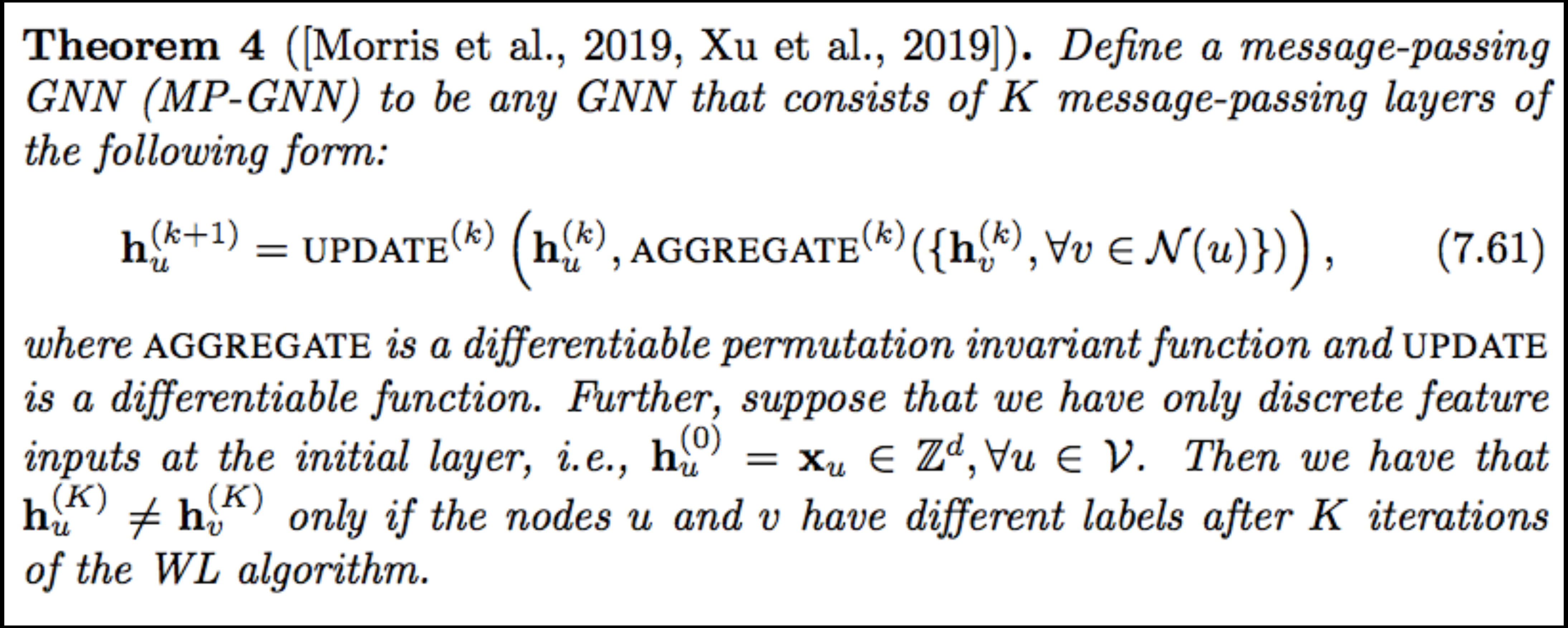


GNNs and the WL Algorithm

In both GNN and WL approaches, we iteratively aggregate information from local node neighborhoods and use this aggregated information to update the representation of each node.

The key distinction between the two approaches is that the WL algorithm aggregates and updates discrete labels (using a hash function) while GNN models aggregate and update node embeddings using neural networks.

The relationship between them can be formalized in the following theorem:



Theorem 4 ([Morris et al., 2019, Xu et al., 2019]). *Define a message-passing GNN (MP-GNN) to be any GNN that consists of K message-passing layers of the following form:*

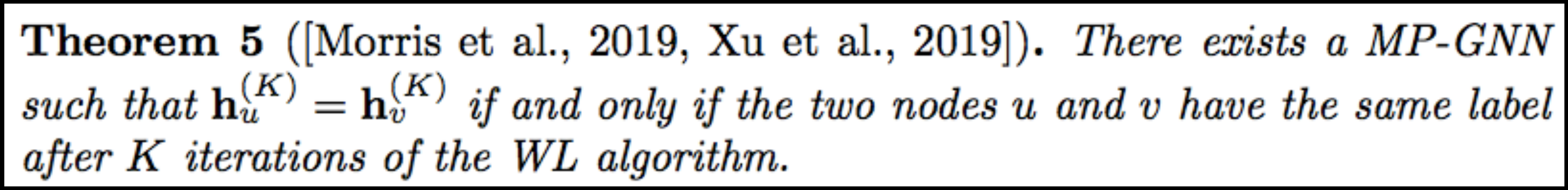
$$\mathbf{h}_u^{(k+1)} = \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right), \tag{7.61}$$

where AGGREGATE *is a differentiable permutation invariant function and* UPDATE *is a differentiable function. Further, suppose that we have only discrete feature inputs at the initial layer, i.e.,* $\mathbf{h}_u^{(0)} = \mathbf{x}_u \in \mathbb{Z}^d, \forall u \in \mathcal{V}$. *Then we have that* $\mathbf{h}_u^{(K)} \neq \mathbf{h}_v^{(K)}$ *only if the nodes* u *and* v *have different labels after* K *iterations of the WL algorithm.*

- If the WL algorithm assigns the same label to two nodes, then any message-passing GNN will also assign the same embedding to these two nodes.

- In theory, GNNs are no more powerful than the WL algorithm when we have discrete information as node features since GNNs have the same analogy as WL algorithm

If the WL test cannot distinguish between two graphs, then a MP-GNN is also incapable of distinguishing between these two graphs.



Theorem 5 ([Morris et al., 2019, Xu et al., 2019]). *There exists a MP-GNN such that* $\mathbf{h}_u^{(K)} = \mathbf{h}_v^{(K)}$ *if and only if the two nodes u and v have the same label after K iterations of the WL algorithm.*

Theoretical motivations of GNN

Consider a graph $G =< V, E, A >$, where V denotes a set of nodes, E is the set of edges, and A corresponds to G's adjacency matrix in which a_ij= 1 if there is an edge $e \in E$ between node i, j and a_ij= 0 otherwise. Its degree matrix D is with D_ii= Σa_ij w.r.t. the sum of row j. The combinatorial Laplacian of G is calculated by $L = D - A$ where L happens to be a positive semi-definite matrix.

There can also be a real function f defined on graph G and f_i is the function value defined on node i.

The graph Fourier transform

$$L = \mathbf{U}\Lambda\mathbf{U}^T$$

$$\mathbf{U} = [u_1, u_2, \ldots, u_{|v|}], \quad \lambda = diag(\lambda_1, \lambda_2, \ldots, \lambda_{|v|}),$$

In particular, we can generalize the notion of a Fourier transform by considering the eigendecomposition of the general graph Laplacian.

U are called graph Fourier basis and λ are called the graph frequencies. The matrix Λ is assumed to have the corresponding eigenvalues along the diagonal, and these eigenvalues provide a graph-based notion of different frequency values.

- The Fourier transform of f on a graph can be computed as

$$s = \mathbf{U}^T f$$

- or inversely

$$f = \mathbf{U}s.$$

Graph convolutions

- Graph convolutions in the spectral domain are defined via point-wise products in the transformed Fourier space.

$$f *_G h = \mathbf{U}((\mathbf{U}^T h) \circ (\mathbf{U}^T f))$$

To ensure that the spectral filter corresponds to a meaningful convolution on the graph, a natural solution is to parameterize the filter based on the eigenvalues of the Laplacian.

Graph convolutions

we can define the spectral filter as

$$\mathbf{U} g_\theta(\Lambda) \mathbf{U}^T f = g_\theta(L) f,$$

so that it is a degree N polynomial of the Laplacian.

our convolution commutes with the Laplacian.

we can recover the key idea that graph convolutions can be represented as polynomials of the Laplacian or its variants

Convolution-Inspired GNNs

The key insight of the GCN approach is that we can build powerful models by stacking very simple graph convolutional layers.

$$H^k = \sigma(\hat{A} H^{k-1} W^k),$$

.

Over-smoothing

The intuitive idea in over-smoothing is that after too many rounds of message passing, the embeddings for all nodes begin to look identical and are relatively uninformative.

The key intuition is that stacking multiple rounds of message passing in a basic GNN is analogous to applying a low-pass convolutional filter, which produces a smoothed version of the input signal on the graph.

Over-smoothing

We simply consider the update function without non-linearities:

$$H^k = A_{sym} H^{k-1} W^k,$$

This model is similar to the simple GCN and essentially amounts to taking the average over the neighbor embeddings at each round of message passing.

Over-smoothing

- After *K* rounds of message passing , we will end up with a representation that depends

$$H^K = A_{sym}^K XW,$$

- W is a linear operator and X is the input feature
- We can interpret the multiplication A^kX as as convolutional filter based on the lowest-frequency signals of the graph Laplacian.
- This corresponds to a convolutional filter based on the lowest eigenvalues/ frequencies.
- In worst case, all the node representations to constant values within connected components on the graph and no longer provide useful signals.

GNN without message-passing

Inspired by connections to graph convolutions, several recent works have also proposed to simplify GNNs by removing the iterative message passing process. In these approaches, the models are generally defined as

$$Z = MLP_1(f(A)MLP_2(X)),$$

$$f : R^{|V|\times|V|} \rightarrow R^{|V|\times|V|}$$

f is a function about the adjacency

- For instance, one can define f on A as:

$$f(A) = \hat{A}^k,$$

- Or inspired by the PageRank theorem

$$f(A) = \alpha(I - (1 - \alpha)\hat{A})^{-1} = \alpha\Sigma_{k=0}^{\infty}(I - \alpha\hat{A})^k.$$

The intuition behind these approaches is that we often do not need to interleave trainable neural networks with graph convolution layers. Instead, we can simply use neural networks to learn feature transformations at the beginning and end of the model and apply a deterministic convolution layer to leverage the graph structure.

Breaking the bottleneck of message passing

This message passing formalism—where nodes aggregate messages from neighbors and then update their representations in an iterative fashion—is at the heart of current GNNs and has become the dominant paradigm in graph representation learning.

These message-passing GNNs are theoretically related to simple convolutional filters, which can be formed by polynomials of the (normalized) adjacency matrix.

Empirically, researchers have continually found message-passing GNNs to suffer from the problem of over-smoothing, and this issue of over-smoothing can be viewed as a consequence of the neighborhood aggregation operation.

Simple Graph Convolution

The authors hypothesize that the nonlinearity be-
tween GCN layers is not critical, the benefit is
mainly from the local averaging within a
neighborhood.

They therefore remove the nonlinear transition
functions between each layer and only keep the
final softmax.

Linearization

The resulting model is linear, but still has the same increased “receptive field” of a K-layer GCN,

$$\hat{\mathbf{Y}} = \mathrm{softmax}\left(\mathbf{S}\ldots\mathbf{S}\mathbf{S}\mathbf{X}\boldsymbol{\Theta}^{(1)}\boldsymbol{\Theta}^{(2)}\ldots\boldsymbol{\Theta}^{(K)}\right)$$

$$\hat{\mathbf{Y}}_{\mathrm{SGC}} = \mathrm{softmax}\left(\mathbf{S}^{K}\mathbf{X}\boldsymbol{\Theta}\right)$$

S is symmetrically normalized adjacency with self-loop

SGC naturally scales to very large graph sizes and the training of SGC is drastically faster than that of GCNs.

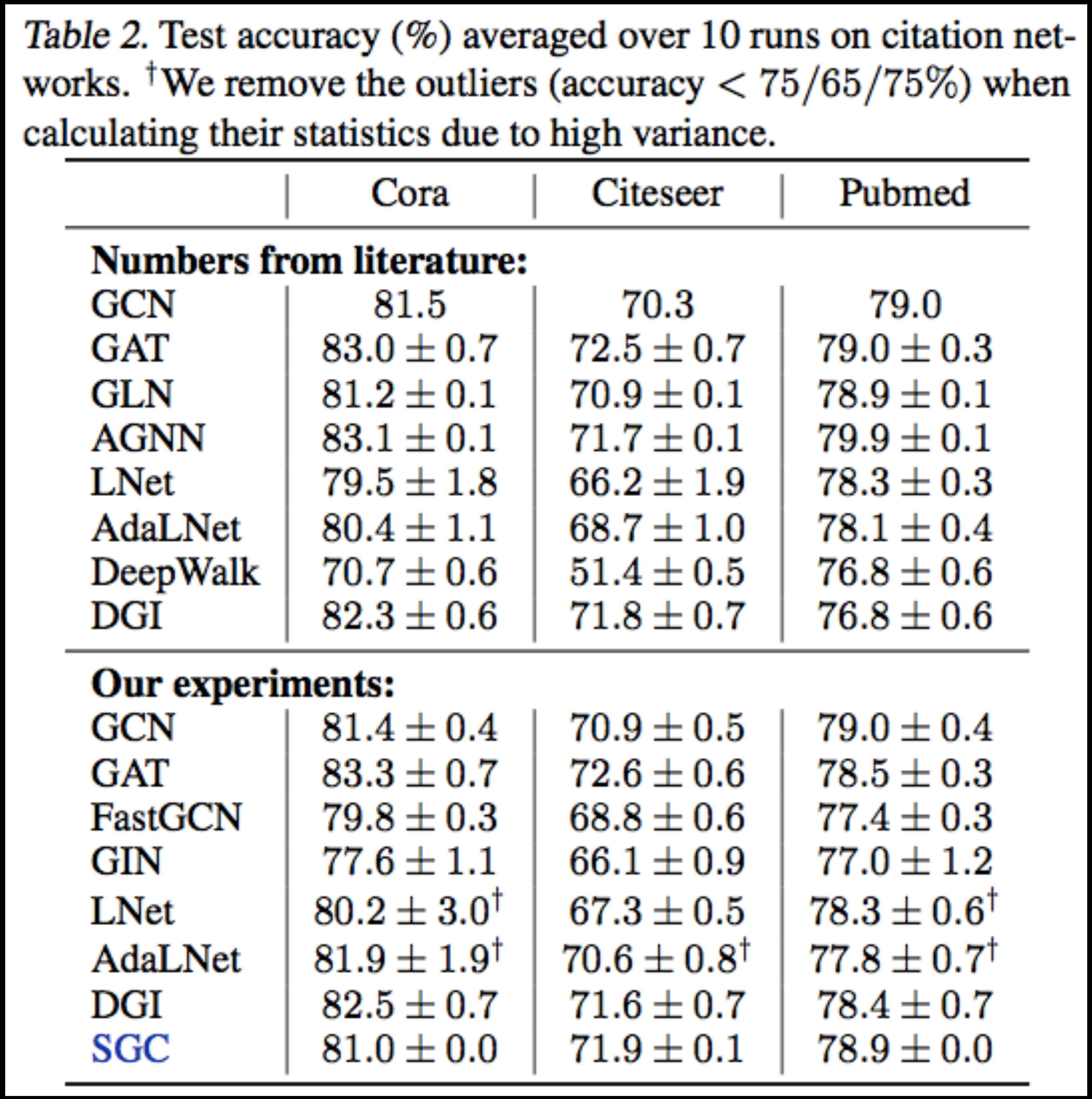


Table 2. Test accuracy (%) averaged over 10 runs on citation networks. [†]We remove the outliers (accuracy $< 75/65/75\%$) when calculating their statistics due to high variance.

	Cora	Citeseer	Pubmed
Numbers from literature:			
GCN	81.5	70.3	79.0
GAT	83.0 ± 0.7	72.5 ± 0.7	79.0 ± 0.3
GLN	81.2 ± 0.1	70.9 ± 0.1	78.9 ± 0.1
AGNN	83.1 ± 0.1	71.7 ± 0.1	79.9 ± 0.1
LNet	79.5 ± 1.8	66.2 ± 1.9	78.3 ± 0.3
AdaLNet	80.4 ± 1.1	68.7 ± 1.0	78.1 ± 0.4
DeepWalk	70.7 ± 0.6	51.4 ± 0.5	76.8 ± 0.6
DGI	82.3 ± 0.6	71.8 ± 0.7	76.8 ± 0.6
Our experiments:			
GCN	81.4 ± 0.4	70.9 ± 0.5	79.0 ± 0.4
GAT	83.3 ± 0.7	72.6 ± 0.6	78.5 ± 0.3
FastGCN	79.8 ± 0.3	68.8 ± 0.6	77.4 ± 0.3
GIN	77.6 ± 1.1	66.1 ± 0.9	77.0 ± 1.2
LNet	80.2 ± 3.0[†]	67.3 ± 0.5	78.3 ± 0.6[†]
AdaLNet	81.9 ± 1.9[†]	70.6 ± 0.8[†]	77.8 ± 0.7[†]
DGI	82.5 ± 0.7	71.6 ± 0.7	78.4 ± 0.7
SGC	81.0 ± 0.0	71.9 ± 0.1	78.9 ± 0.0

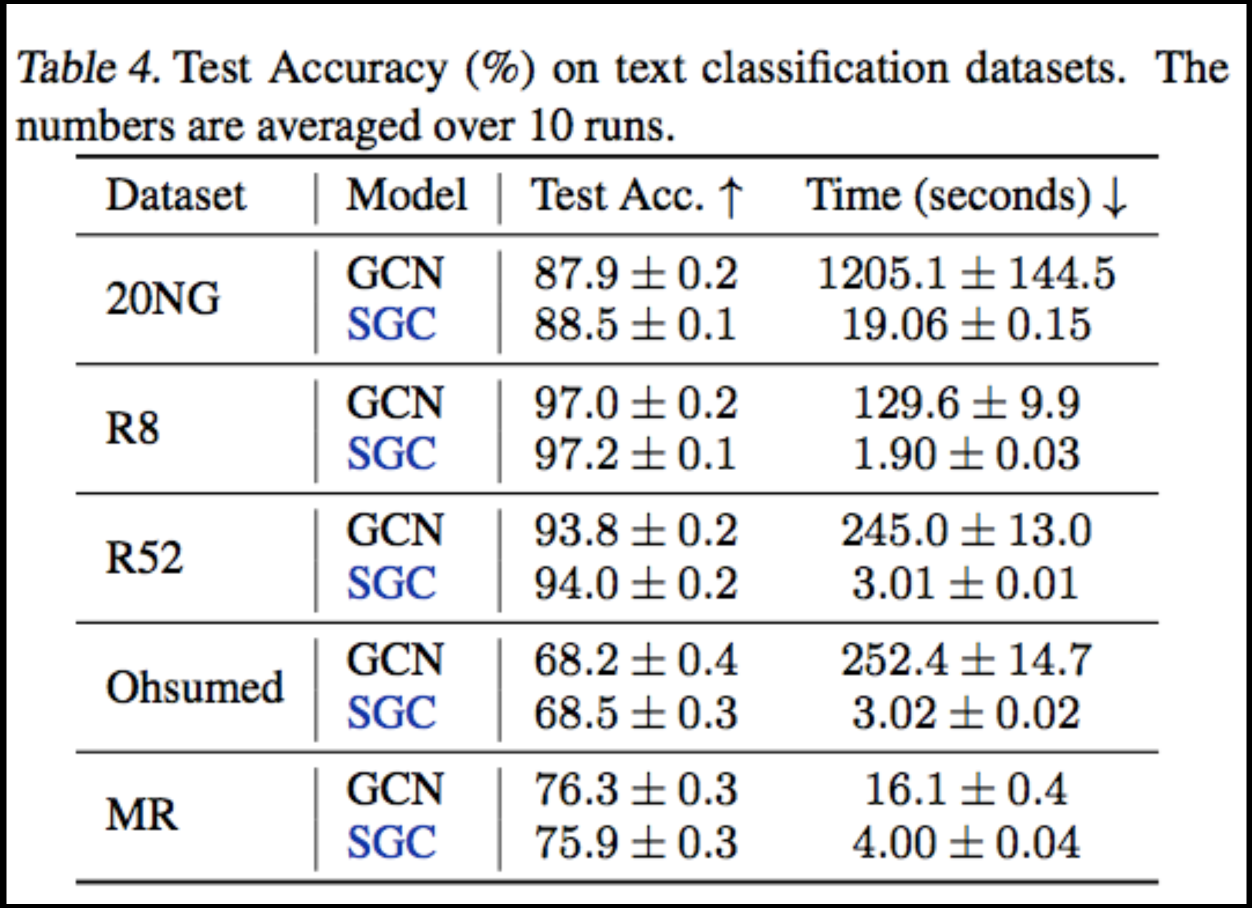


Table 4. Test Accuracy (%) on text classification datasets. The numbers are averaged over 10 runs.

Dataset	Model	Test Acc. ↑	Time (seconds) ↓
20NG	GCN	87.9 ± 0.2	1205.1 ± 144.5
	SGC	88.5 ± 0.1	19.06 ± 0.15
R8	GCN	97.0 ± 0.2	129.6 ± 9.9
	SGC	97.2 ± 0.1	1.90 ± 0.03
R52	GCN	93.8 ± 0.2	245.0 ± 13.0
	SGC	94.0 ± 0.2	3.01 ± 0.01
Ohsumed	GCN	68.2 ± 0.4	252.4 ± 14.7
	SGC	68.5 ± 0.3	3.02 ± 0.02
MR	GCN	76.3 ± 0.3	16.1 ± 0.4
	SGC	75.9 ± 0.3	4.00 ± 0.04