Learning from Networks

Graph Neural Networks

Fabio Vandin

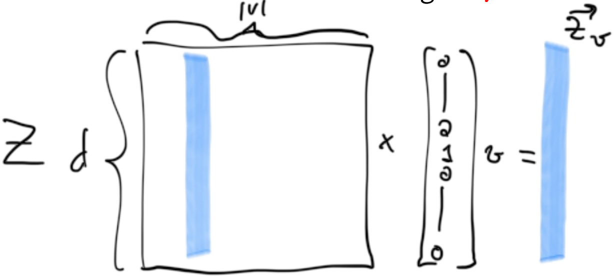
November 21st, 2024

Shallow Embeddings

Node embedding task: given graph $G = (V, E)$, represent each $v \in V$ as $ENC(v) = \mathbf{z}_v \in \mathbb{R}^d$ such that similarities $S(u, v)$ in G are approximated by (decoded) similarities $DEC(\mathbf{z}_u, \mathbf{z}_v)$ for all $u, v \in V$:

$$S(u, v) \approx DEC(\mathbf{z}_u, \mathbf{z}_v)$$

We have seen *shallow embeddings*: $\mathbf{z}_v = \mathbf{Z} \cdot \mathbf{v}$



Shallow Embeddings: Limitations

Large number of parameters: high *complexity* of the *model*

Inherently *transductive*: cannot generate embeddings for nodes that are not present during the *training* phase

We would like *inductive* embedding methods: can generate the embeddings for new nodes

Do not incorporate node features: many networks have node features that can/should be used to produce the embedding

They are *unsupervised* methods: learned embeddings are independent of the (downstream) ML task

Graph Neural Network Embeddings: Main Idea

Instead of learning $\mathbf{z}_v$ for each $v \in V$, learn a function $f : V \to \mathbb{R}^d$

f is computed by a *neural network* that depends on the structure of the graph $\Rightarrow$ *graph neural network* (GNN)

Given the graph G and features of the nodes $v \in V$, the GNN can compute the embedding of *any* node $u \in V$

Encoder-decoder framework:

- the encoder $ENC()$ is a neural network
- other components (similarity, decoder, loss function): as for shallow embeddings

$\Rightarrow$ the parameters to be learned from the data are in the neural network

GNN: Setup

We have node features:

- social networks: user profile info, activity info, . . .
- biological networks: gene expression profiles, functional information, . . .
- what if no (external) features available?
  - compute node features: clustering coefficient, centrality scores, etc.
  - *one-hot encoding of a node* :

$$\left.\begin{bmatrix}0\\ \vdots\\ 0\\ 1\\ 0\\ \vdots\\ 0\end{bmatrix}\begin{matrix}\\ \\ \\ v\\ \\ \\ \\ \end{matrix}\right\}|V|$$

GNN: Setup

We have node features:

- social networks: user profile info, activity info, ...
- biological networks: gene expression profiles, functional information, ...
- what if no (external) features available?
    - compute node features: clustering coefficient, centrality scores, etc.
    - *one-hot encoding of a node*
    - vector of all 1's

Given

- graph $G = (V, E)$, with $|V| = n$
- $\mathbf{A}$ is the adjacency matrix of G
- each node $v \in V$ has vector $\mathbf{x}_v \in \mathbb{R}^r$ of r *features*
- $\mathbf{X} \in \mathbb{R}^{r \times n}$ is the matrix of nodes features

Goal: we want to combine the features $\mathbf{x}_v$ of a node and the structural (topological) information from G to obtain encodings for each node $v \in V$

Ideas?

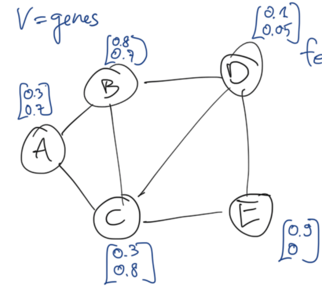
Naïve Approach

Idea:

- for each node $v \in V$, build an input vector that contains both the node features $\mathbf{x}_v$ and the *adjacency matrix vector* for v
- $ENC(v)$ is the output of a (deep) neural network whose input are the vectors above

Example

V=genes



$\begin{bmatrix}0.1\\0.05\end{bmatrix}$
$\begin{bmatrix}0.8\\0.7\end{bmatrix}$
$\begin{bmatrix}0.3\\0.7\end{bmatrix}$
A
B
C
D
E
$\begin{bmatrix}0.3\\0.8\end{bmatrix}$
$\begin{bmatrix}0.9\\0\end{bmatrix}$

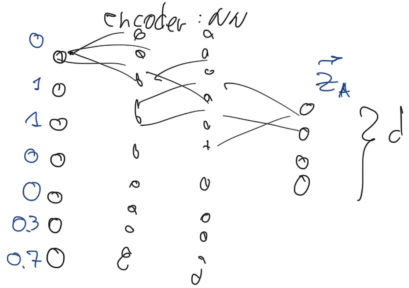
features: → activity in condition 1
→ activity in condition

adjacency matrix :

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 0 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 1 & 1 & 0 & 1 & 1 \\ D & 0 & 1 & 1 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

input vectors for the NN:

$$\begin{array}{ccccc} A & B & C & D & E \\ \begin{bmatrix}0\\1\\1\\0\\0\\ \hline 0.3\\0.7\end{bmatrix} & \begin{bmatrix}1\\0\\1\\1\\0\\ \hline 0.8\\0.7\end{bmatrix} & \begin{bmatrix}1\\1\\0\\1\\1\\ \hline 0.3\\0.8\end{bmatrix} & \begin{bmatrix}0\\1\\1\\0\\1\\ \hline 0.1\\0.05\end{bmatrix} & \begin{bmatrix}0\\0\\1\\1\\0\\ \hline 0.9\\0\end{bmatrix} \end{array}$$



Naïve Approach: Issues

Number of parameters? $\Omega(n)$ $\rightarrow$ because the input layer has $\Omega(n)$ nodes

Not applicable to graphs of different sizes $\rightarrow$ they have input layers of different sizes

Depends on the node ordering $\rightarrow$ if I change the order of nodes in the adjacency matrix, the embeddings of the nodes change

Permutation Invariance and Permutation Equivariance

We would like the embedding to be *independent* of the node ordering.

Let f be the function that given the adjacency matrix $\mathbf{A}$ of G and the feature matrix $\mathbf{X}$ of the nodes of G produces in output the embedding matrix $\mathbf{Z}$:

$$\mathbf{Z} = f(\mathbf{A}, \mathbf{X})$$

We would like f to be *permutation invariant* or *permutation equivariant*

Permutation Invariance and Permutation Equivariance (continue)

We would like it to be *permutation invariant* or *permutation equivariant*

Permutation matrix $\mathbf{P}$: each row/column has exactly one 1, all other entries are 0.

Definition

f is *permutation invariant* if $f(\mathbf{PAP}^T, \mathbf{XP}^T) = f(\mathbf{A}, \mathbf{X})$ where $\mathbf{P}$ is a *permutation matrix*.

Definition

f is *permutation equivariant* if $f(\mathbf{PAP}^T, \mathbf{XP}^T) = f(\mathbf{A}, \mathbf{X})\mathbf{P}^T$ where $\mathbf{P}$ is a *permutation matrix*.

Example

Neural Message Passing Framework

Idea

- the computation proceeds in iterations
- in iteration k, the (hidden) embedding $\mathbf{h}_v^{(k)}$ for node v is updated/computed according to the (hidden) embeddings of nodes $u \in \mathcal{N}(v)$
- the output embedding for node v is the embedding $\mathbf{h}_v^{(K)}$ after K iterations
- initialization: $\mathbf{h}_v^{(0)} = \mathbf{x}_v$ for all $v \in V$

Formally:

- $\texttt{AGGREGATE}^{(k)}\left(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\}\right) = \mathbf{m}_{\mathcal{N}(u)}^{(k)}$: function that given the (hidden) embeddings of neighbours of u at iteration k produces the *message* $\mathbf{m}_{\mathcal{N}(u)}^{(k)}$.
- $\texttt{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right)$

$\texttt{AGGREGATE}^{(k)}(\dots)$ and $\texttt{UPDATE}^{(k)}(\dots)$ are arbitrarily differentiable functions $\Rightarrow$ neural networks

Neural Message Passing Framework

Then for each $u \in V$:

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

The embedding $\mathbf{z}_u$ of a node $u \in V$ is the embedding after K iterations:

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}$$

Note: the iterations of message passing are also called *layers* of the GNN

Example

Example

Example

Neural Message Passing Framework

Question: are GNNs (as generally defined in the previous slide) sensible to permutations of the nodes?

Neural Message Passing: Motivation

Intuition: the local feature-aggregation behaviour of GNNs is analogous to the behavior of the convolutional filters in CNNs

Basic GNNs

The most basic version of a GNN is given by:

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

Therefore:

$$\begin{aligned}\mathbf{h}_u^{(k+1)} &= \texttt{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right)\\ &= \sigma\left(\mathbf{W}_{\text{self}}^{(k+1)}\mathbf{h}_u^{(k)} + \mathbf{W}_{\text{neigh}}^{(k+1)}\mathbf{m}_{\mathcal{N}(u)}^{(k)} + \mathbf{b}^{(k+1)}\right)\end{aligned}$$

Notes

- $\mathbf{W}_{\text{self}}^{(k+1)}$ and $\mathbf{W}_{\text{neigh}}^{(k+1)}$ are trainable parameters, with $\mathbf{W}_{\text{self}}^{(k+1)}, \mathbf{W}_{\text{neigh}}^{(k+1)} \in \mathbb{R}^{d^{(k+1)} \times d^{(k)}}$
- $\mathbf{b}^{(k+1)}$ is the *bias*; often omitted in the notation
- $\sigma()$ is an *elementwise* non-linear function (e.g., ReLU)

Basic GNNs (continue)

Putting all together, for each node u, in each iteration:

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

GNNs: a Theoretical Motivation

We will now see a theoretical motivation for GNNs, by comparing them with an algorithm for the *graph isomorphism problem*.

Definition

Given two graphs G_1 and G_2, the *graph isomorphism problem* requires to determine whether G_1 is isomorphic to G_2: $G_1 \simeq G_2$

Is it a hard problem? It is not known to be solvable in polynomial time nor to be NP-complete.

Note: ideally we would like a graph embedding technique to *solve* the graph isomorphism problem, that is: $\mathbf{z}_{G_1} = \mathbf{z}_{G_2}$ if and only if $G_1 \simeq G_2$

Weisfieler-Leman (WL) Algorithm

Algorithm for the graph isomorphism problem.

Note: does not always produce the correct answer!

- if WL outputs that $G_1 \not\simeq G_2$ then $G_1 \not\simeq G_2$
- if WL outputs that $G_1 \simeq G_2$, then it may be that $G_1 \not\simeq G_2$

It is known that WL produces the correct answer for a large class of graphs.

WL Algorithm

Builds on a *color* refinement algorithm for the vertices of a graph G.

Given a coloring C_V of the vertices V of $G = (V, E)$, let $P(C_V)$ the partition of the vertices V defined by the coloring C_V: two vertices u, v are in the same set of the partition if and only if u and v have the same color.

WL Algorithm (continue)

```
**Algorithm** ColorRefinement$(G)$
**Input:** graph $G = (V, E)$
**Output:** coloring of $V$
$C_{curr} \leftarrow$ assign the same color $u$ to all nodes $u \in V$;
repeat
    $C_{prev} \leftarrow C_{curr}$;
    $C_{curr} \leftarrow$ for each pair $u$ and $v$ where $u$ and $v$ have the
     same color in $C_{prev}$: assign different colors to $u$ and $v$ if
     and only if there is some color $c$ such that $u$ and $v$ have
     different number of neighbours of color $c$ in $C_{prev}$;
until $P(C_{prev}) = P(C_{curr})$;
return $C_{curr}$;
```

Analysis:

- ColorRefinement(G) stops after at most $|V|$ iterations;
- each iteration requires $O\left(|V|^2\right)$ operations
- the complexity of ColorRefinement(G) is $O\left(|V|^3\right)$

WL Algorithm (continue)

```
**Algorithm** WL($G_1, G_2$)
**Input:** graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$
**Output:** yes/no
$C_{G_1} \leftarrow$ ColorRefinement($G_1$);
$C_{G_2} \leftarrow$ ColorRefinement($G_2$);
**return** $histogram(P(C_{G_1})) = histogram(P(C_{G_2}))$;
```

Analysis: the complexity of WL(G_1, G_2) is $O\left(|V_1|^3 + |V_2|^3\right)$

Example

Example

Example

WL algorithm and GNNs

Theorem

Consider a GNN with K message passing layers of the following form:

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

where $\mathtt{AGGREGATE}$ is differentiable and permutation invariant and $\mathtt{UPDATE}$ is differentiable. Assume that the input is made of discrete features: $\mathbf{h}_u^{(0)} = \mathbf{x}_u \in \mathbb{Z}^d, \forall u \in V$.
Then $\mathbf{h}_u^{(K)} \neq \mathbf{h}_v^{(K)}$ only if u and v have different labels after K iterations of the WL algorithm.

Informally: GNNs are *no more powerful than* the WL algorithm when we have discrete information as node features.

The result generalizes to graphs: if the WL algorithm does not distinguish (non-isomorphic) graphs G_1 and G_2, then any GNN (with the form as above) is incapable of distinguishing G_1 and G_2.

WL algorithm and GNNs (continue)

Theorem

There exists a GNN with the form defined in the previous theorem such that $\mathbf{h}_u^{(K)} = \mathbf{h}_v^{(K)}$ only if u and v have the same labels after K iterations of the WL algorithm.

Informally: there are GNNs that are *as powerful as* the WL algorithm.

Which GNNs are as powerful as the WL algorithm?

- the basic GNN? YES!
- basic GNN with neighborhood normalization? NO!
- GCN? NO!
- GraphSAGE? depends on the `AGGREGATION` operator

GNNs for Node Embeddings

In general: GNNs can be used to obtain node embeddings.

In terms of the encoder-decoder framework:

- *encoder*: GNN
- *similarity function*: see the ones for shallow embeddings
- *decoder*: see the ones for shallow embeddings
- *loss*: see the ones for shallow embeddings

However: GNNs can also be used for *supervised* tasks.

GNNs for Supervised-Tasks

Common supervised tasks for graphs:

- node classification
- graph classification

GNNs: Node Classification

Input: some nodes have a label and can be used to train the GNN.

Each node u in the training set has a label, encode by a $0-1$ vector $\mathbf{y}_u$ of dimension c.

Loss function: negative log-likelihood loss of softmax classification function

Given the embedding $\mathbf{z}_u$ of node u and the corresponding vector $\mathbf{y}_u$, the softmax classification function is

$$\text{softmax}(\mathbf{z}_u, \mathbf{y}_u) = \sum_{i=1}^{c} \mathbf{y}_u[i] \frac{e^{\mathbf{z}_u^T \mathbf{w}_i}}{\sum_{j=1}^{c} e^{\mathbf{z}_u^T \mathbf{w}_j}}$$

with $\mathbf{w}_i$ for $i = 1, \ldots, c$ trainable parameters.

GNNs: Node Classification (continue)

Then the loss function is:

$$\mathcal{L} = \sum_{u \in V_{\text{train}}} - \log \left(\text{softmax}(\mathbf{z}_u, \mathbf{y}_u) \right)$$

where V_{train} is the set of *training nodes*.

Note: there are different types of nodes for node classification

- the set V_{train} of *training nodes*: both the nodes and the labels are used during training (i.e., to learn the GNN parameters)
- the set V_{trans} of transductive test nodes: the nodes, but not their labels, are used during training. These nodes do not (directly) contribute to $\mathcal{L}$
- the set V_{ind} of inductive test nodes: the nodes are completely *unobserved* during training

GNNs: Graph Classification

Input: set of graphs, where each graph G has a label represented by a $0-1$ vector $\mathbf{y}_G$ of dimension c

Loss: same as for node classification, where the embedding $\mathbf{z}_G$ is computed for a graph G.

$$\text{softmax}(\mathbf{z}_G, \mathbf{y}_G) = \sum_{i=1}^{c} \mathbf{y}_G[i] \frac{e^{\mathbf{z}_G^T \mathbf{w}_i}}{\sum_{j=1}^{c} e^{\mathbf{z}_G^T \mathbf{w}_j}}$$

$$\mathcal{L} = \sum_{G \in \mathcal{G}_{\text{train}}} - \log\left(\text{softmax}(\mathbf{z}_G, \mathbf{y}_G)\right)$$

where $\mathcal{G}_{\text{train}}$ is the set of graphs used for training.

GNNs: other tasks

GNNs can be used for several other tasks

- edge prediction: predict *missing edges*
- regression task for nodes
- regression task for graphs

GNNs: Additional Notes

What algorithm is used to learn GNN parameters? **SGD**

Training all nodes of a GNN simultaneously can require a lot of resources (time, memory) $\Rightarrow$ **Mini-batching:** obtain the final representation z_u for a small set of nodes at the time

Problem: how do we make sure that the computation graph is connected but not all nodes are used? $\Rightarrow$ Start from the target nodes and use **subsampling**: sample a fixed number of neighbours for each node.

The loss function can be combined with *regularization* (e.g., ℓ_2 regularization)

GNNs: Frameworks

If you are starting to use GNNs, here are some pointers to GNN Python libraries:

PyTorch Geometric
https://pytorch-geometric.readthedocs.io/en/latest/

Deep Graph Library https://www.dgl.ai/

Graph Nets https://github.com/deepmind/graph_nets

Spektral https://graphneural.network/