Machine Learning with Graphs: Representation learning 3/3 - Graph Neural Networks

Arlei Silva

Spring 2022

GCN for semi-supervised learning

GCNs were originally proposed as a solution to the semi-supervised learning problem on attributed graphs—predicting the labels of nodes based on a few labeled ones. Let $X \in \mathbb{R}^{n \times D}$ be the matrix of node attributes and the set of labels/classes be $1, \dots C$. To simplify the notation, let $\hat{A} = \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$. Then a 2-layer GCN is defined as follows:

$$Z = \text{softmax}(\hat{A}\text{ReLU}(\hat{A}W^{(0)})W^{(1)})$$

where $W^{(0)} \in \mathbb{R}^{D \times h}$ and $W^{(1)} \in \mathbb{R}^{h \times C}$ are parameters to be learned (similar to Θ), $\text{ReLU}(x) = \max(0, x)$, and the $softmax$ function is applied to convert the output into probabilities over the C possible classes:

$$\text{softmax}(x_c) = \frac{\exp(x_c)}{\sum_{i=1}^{C} \exp(x_i)}$$

To learn the parameters, $W^{(1)}$ and $W^{(1)}$, we minimize the *cross-entropy loss* of the model:

$$\mathcal{L} = -\sum_{v \in Y_L} \sum_{c=1}^{C} [label(v) = c] \log(Z[v, c])$$

where Y_L is the set of labeled nodes and $[exp]$ returns 1 if exp is TRUE and 0, otherwise. We minimize $\mathcal{L}$ using SGD or your favorite optimization algorithm.

Notice that GCNs are *inductive*, as the learned weights $W^{(0)}, W^{(1)}$ can be applied to nodes unknown during training as long as node attributes remain the same. In particular, GCNs can be applied to classify nodes in a different graph than the one used during training.

Message Passing Neural Networks

Many alternative Graph Neural Network models have been proposed after GCNs. It is convenient to see these models as special cases of *Message-Passing Neural Networks (MPNNs)*, which have been proposed as a general framework generalizing multiple GNNs. The idea of MPNNs is to initialize Z—sometimes called node embeddings or representations—and then update embeddings via multiple iterations of *message-passing*. A *message* from node u to node v at the layer ℓ is defined as:

$$m_{uv}^{\ell} = f_e(\mathbf{z}_u^{\ell}, \mathbf{z}_v^{\ell}, e_{u,v})$$

where f_e is a (generic) function, $\mathbf{z}_u^{\ell} \in \mathbb{R}^{h_\ell}$ is the embedding of u, and e_{uv} are edge attributes.

A *pooling operator* is used to update the embedding of node u based on its own current embedding and the embedding of its neighbors:

$$\mathbf{z}_u^{\ell+1} = f_v(\mathbf{z}_u^{\ell}, \cup_{k \in N(u)} m_{kv}^{\ell})$$

where f_v is a (generic) function. Notice that the messages from the neighbors are given as a set. In general, it is desired for f_v to be *permutation-invariant* to any arbitrary order (e.g. as columns of an adjacency matrix) over the neighbors.

Note that f_e and f_e can be either given or learned (neural nets). Embeddings are usually initialized as $\mathbf{z}_v^0 = X[v]$. For T layers, the output are embeddings $\mathbf{z}_1^T, \dots \mathbf{z}_n^T$. We can now see how MPNNs generalize some existing GNNs.

GCN: Messages are $m_{uv}^{\ell} = \mathbf{z}_u^{\ell}$ and the pooling operator is:

$$\mathbf{z}_u^{\ell+1} = \sigma\left(\frac{1}{\sqrt{|N(u)|+1}} W^t \left(\frac{\mathbf{z}_u^{\ell}}{\sqrt{|N(u)|+1}} + \sum_{k \in N(u)} \frac{\mathbf{z}_k^{\ell}}{\sqrt{|N(k)|+1}}\right)\right)$$

GraphSAGE: Messages are $m_{uv}^{\ell} = \mathbf{z}_u^{\ell}$ and the pooling operator is:

$$\mathbf{z}_u^{\ell+1} = \sigma([\mathbf{z}_u^{\ell}; \text{AGGREGATE}(\cup_{k \in N(u)} \mathbf{z}_k^{\ell})] W^{(\ell)})$$

where σ can be any activation function (e.g. softmax), $[\cdot, \cdot]$ is concatenation, and AGGREGATE is a generic operator (e.g., mean, max, LSTM).

GraphSAGE also applies subgraph sampling to train its parameters via *mini-batch gradient descent*—the same approach works for other GNNs as well. A mini-batch contains a set of nodes $B \in V$ and the forward operation over a mini-batch accounts for a fixed number of node samples within the K-hop neighborhood of B. For each layer, ℓ, an embedding $\mathbf{z}_u^{\ell+1}$ is updated based on messages from the neighbors within ℓ hops from node u.

Graph Attention Network (GAT): Messages are $m_{uv}^{\ell} = \mathbf{z}_u^{\ell}$ and the pooling operator is:

$$\mathbf{z}_u^{\ell+1} = \sigma\left(\sum_{k \in N(u) \cup \{up\}} \alpha_{uk} \mathbf{z}_u^{\ell}\right)$$

where α_{uk} are edge attention weights:

$$\alpha_{uv} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^T[\mathbf{z}_u^{\ell} W, \mathbf{z}_v^{\ell} W]))}{\sum_{k \in N(u)} \exp(\text{LeakyReLU}(\mathbf{a}^T[\mathbf{z}_u^{\ell} W, \mathbf{z}_k^{\ell} W]))}$$

where $\mathbf{a} \in \mathbb{R}^{2p}$ and $W \in \mathbb{R}^{h_\ell \times p}$ are parameters to be learned.

Other tasks

We have described how to apply GCNs for semi-supervised learning. Other GNN architectures can be applied for the same task in a similar fashion. Here, we will discuss how GNNs can be applied to other tasks, graph classification, and link prediction.

Graph classification: Here, the goal is to learn how to classify entire graphs from a subset of labeled graphs. This can be performed using, again, a pooling operator, which computes graph-level representations $\mathbf{z}_G \in \mathbb{R}^h$ from the node representations $\mathbf{z}_1, \ldots \mathbf{z}_n$, where $\mathbf{z}_i \in \mathbb{R}^h$. Examples of such an operator include the (column-wise) mean, max, and sum. Graph representations $\mathbf{z}_G$ can then be given to a classifier (e.g. Multi-layer Perceptron) to produce graph labels.

Link prediction: The task consists of predicting new (or missing) links based on observed ones. There are many GNN-based approaches for link prediction. The simples ones is to predict an edge (u, v) based on $\sigma(\langle \mathbf{z}_u, \mathbf{z}_v \rangle)$, where σ is the *sigmoid function*:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

Advanced topics

Graph Neural Networks became popular quite recently are still a topic of active research. We will briefly discuss two advanced topics related to GNNs. For more, check recent papers at the top conferences in machine learning, data mining, computer vision, etc.

Oversmoothing

Neural networks (e.g. CNNs) often benefit from depth– i.e. multiple layers– to learn complex representations for data. One would expect the same for graphs.

However, it has been found that GNNs suffer from *oversmoothing*—i.e. node feature information is lost for a large number of layers. It is easier to show oversmoothing for the case of L linear layers. Then, we can write the GNN output as:

$$\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}(\ldots\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}XW^{(0)}\ldots)W^{(L-2)})W^{(L-1)}$$

The above expression can be simplified to:

$$(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2})^{L}XW$$

where $W = W^{(0)} \times W^{(1)} \times \ldots W^{(L-1)}$.

We are interested in the GNN representations as $L \to \infty$. First, let us compute the following limit (without the weights):

$$\lim_{L\to\infty}(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2})^{L}X$$

Let $L = (I - \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}) = U\Lambda U^{T}$. Then $\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2} = (I - L) = U(I - \Lambda)U^{T}$ and $(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2})^{L} = (I - L) = U(I - \Lambda)^{L}U^{T}$. Remember that the eigenvalues of $\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$ are upper bounded by 2. Moreover, the smallest eigenvalue of the same matrix is 0—with eigenvector $D^{1/2}\mathbf{1}_n$. It follows that $(I - \Lambda) = \mathbf{diag}(\lambda_1, \ldots \lambda_n)$ is such that $-1 < \lambda_1 \leq \ldots \lambda_n = 1$ and, finally:

$$\begin{aligned}\lim_{L\to\infty}(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2})^{L}X &= U\mathbf{diag}(0,\ldots,0,1)U^{T}X \\ &= \mathbf{u}_1\mathbf{u}_1^{T}X \\ &= \mathbf{u}_1.[\langle\mathbf{u}_1, X[:,1]\rangle,\ldots,\langle\mathbf{u}_1, X[:,D]\rangle]\end{aligned}$$

where $X[:,i]$ is a column of the feature matrix X.

Notice that, regardless of weights W, resulting node representations do not capture the node features. The vector $\mathbf{u}_1 = D^{1/2}\mathbf{1}_n$ and $\langle\mathbf{u}_1, X[:,i]\rangle$ has the same value for every vertex.There are approaches to address oversmoothing in GNNs (mostly regularization) [3].

Expressive power

The *expressive power* of a neural network is its capacity to approximate functions—i.e. map inputs to outputs. The so-called *universal approximation theorem* states that neural networks with one hidden layer can approximate any continuous function. Similar questions have been asked about the expressive power of GNNs in terms of distinguishing non-isomorphic graphs.

Two graphs $G = (V, E)$ and $G' = (V', E')$ are said to be *isomorphic* if there is a bijection $\phi : V \to V'$ such that $\{u, v\} \in E$ iff $\{\phi(u), \phi(v)\} \in E'$. The isomorphism problem consists of checking whether two graphs are isomorphic. It is not known whether there exist poly-time algorithms for graph isomorphism—there is no NP-hardness proof either.

The *Weisfeiler-Lehman test (WL)* is an efficient algorithm for isomorphism checking. If the result of the test is false, the input graphs are known to be non-isomorphic. However, the graphs are not necessarily isomorphic if the result is true. The idea is to assign a label to each node as a function of the labels of its neighbors and its own label. This step is repeated until the labels are not changed anymore. The test returns true if the two graphs have the same number of nodes with each label at the end and false otherwise. This test is known as *1-WL* and generalizations (*k-WL*) operate on k-tuples of nodes.

MPNNs are related to the WL test. More specifically, we can think of node representations as labels and pooling as a function of the labels of neighbors. A graph-level pooling operator can then be used to compute a graph-level representation. The test returns true if the (graph-level) representation of the input graphs is the same. It has been shown that if the MPNN applies operators that are injective, then it is as expressive as the 1-WL test [4].

References

[1] William L Hamilton. *Graph representation learning*. Morgan & Claypool, 2020.

[2] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[3] Yao Ma and Jiliang Tang. *Deep learning on graphs*. Cambridge University Press, 2021.

[4] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2018.