Laplacian Pooling (LaPool)

By Jose Ezequiel Castro Elizondo

Introduction

After the popularity boost of deep learning methods on images and speech recognition because of their great effectiveness, a new question arises among data scientist: “is it possible to apply convolutional methods to graph such that we can leverage the huge amount of information that a single graph can hold?”, giving birth to what is now known as Graph Convolutional Networks (GCNs).

Jepsen (2019) defines a graph convolutional network as a neural network that operates on graphs. This neural network takes as input:

- A feature matrix $X \in \mathbb{R}^{n\,x\,d}$ where n is the number of nodes and d represents the embedding dimensions for each node.
- An n x n adjacency matrix that represents the graph structure.

We can, thus, express a hidden layer as $\boldsymbol{H^i = f(H^{i-1}, A)}$ where each layer $\boldsymbol{H^i}$ corresponds to a $\boldsymbol{n\,x\,d^i}$ feature matrix. At each layer, the features are aggregated to produce the next features by using the propagation rule “f(-)”.

Analogous to image learning, the input data can sometimes have immense size which may be difficult or even infeasible in some cases to process. According to Trudeau (2019) “Size is one problem that graph represents as a data structure, […] you can’t efficiently store a large social network in a tensor”. This presents a new challenge to the GCN field, how can we represent a huge structure of information? The same problem arises in the deep learning methods, for example, when an image is big enough, we use some pooling methods that extracts the important information of the image and shrinks the original size. Thus, the question to answer is: how to implement pooling into a GCN? It is true that there are some similarities between image processing deep learning and graph deep learning, but it is also true that there are some differences. One clear example of these differences is that in the graph domain, there is proper beginning or end, and two connected nodes are not necessarily close to each other, suggesting that graphs have no fixed location in space, representing a big challenge to pooling layers.

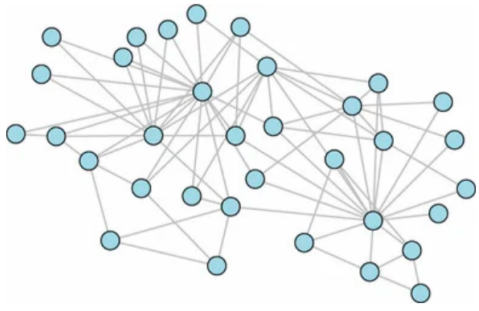

Fig. 1: Graphs, unlike images, have arbitrary structures (Trudeau,2019)

GNNs can be either spectral or spatial based. A spectral based implementation uses the graph convolution (GC) like a filtering operation, while a spatial based implementation uses the GC as a message passing method in order to aggregate node's embeddings. In whichever the case is, the updating process of node features can produce an actual decrease in the model's performance. This is due to a signal smoothing effect that each GC layer has. This results in an important reduction of the network's receptive field (Qimai Li et. al, 2018). Pooling methods can also solve this problem by reducing the graph size.

Conversely to other pooling methods, the Laplacian Pooling (LaPool) takes into account the graph structure and its node's features, making it great for molecular graph representations. In the next sections we will make an overview of the LaPool layer and comment over the python implementation of module.

LaPool Overview

We start by defining an undirected graph G as $\boldsymbol{G} = \langle \boldsymbol{V}, \boldsymbol{A}, \boldsymbol{X} \rangle$ where V is a set containing all its vertex $\boldsymbol{V} = \{\boldsymbol{v_1}, \boldsymbol{v_2}, \ldots, \boldsymbol{v_n}\}$, A denotes the adjacency matrix of the graph $\boldsymbol{A} = \{\boldsymbol{0}, \boldsymbol{1}\} \in \mathbb{R}^{\boldsymbol{n\,x\,n}}$ and X represent the node features $\boldsymbol{X} = \{\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}\} \in \mathbb{R}^{\boldsymbol{n\,x\,d}}$.
For any graph G, the unnormalized graph Laplacian (**L**) can be used as a measure of the smoothness of a signal s(f), that is, it can define how much a graph signal changes between two connected nodes.

The steps to follow for obtaining a Laplacian pooling are as follows:
Graph convolution: perform a smoothing of the graph signal (acting as a low-pass filter) to attenuate noise.

Centroid Selection: Nodes on the graph are selected based on their local signal variation ($\boldsymbol{s_i}$) for node $\boldsymbol{v_i}$.

$$s_i = \sum_{j \in \mathcal{N}(v_i)} A_{i,j} \|\mathbf{x}_i - \mathbf{x}_j\|_2, \qquad S = \left[s_1, \ldots s_n\right]^{\intercal} = \|L \cdot X\|_{\mathbf{R}^d}$$

Thus, $\boldsymbol{s_i}$ measures how different a node is from the average of its neighbors. The selection of the representative centroids will imply a high-pass filter on the graph's signals, which, in combination with the low-pass filter of the GC we obtain a band-pass filter which retains the most important signals of the graph. According to Noutahi et. al (2020) the selection of the centroids should be done in a dynamic matter; this is because stablishing a minimum number of centroids "k" can result in an overestimation of centroids. The dynamic selection, thus, should select nodes where the signal $\boldsymbol{s_i} > \boldsymbol{s_j}$ where $\boldsymbol{s_j}$ is the signal of every neighbor node of i.

$$V_C = \{v_i \in V \mid \forall\, v_j \in \mathcal{N}(v_i)\, s_i > s_j\}$$

Node-to-cluster mapping: After obtaining the centroids of the graph ($\boldsymbol{V_c} \in \mathbb{R}^{n\,x\,m}$) we can compute the cosine similarity between the node features X and the node features that are centroids $\boldsymbol{X_c}$. These similarities will conform a cluster assignment matrix $C = [\boldsymbol{c_1}, \boldsymbol{c_2}, \ldots, \boldsymbol{c_n}]^T \in [\boldsymbol{0}, \boldsymbol{1}]^{n\,x\,m}$ where each row ($\boldsymbol{c_i}$) represents the affinity of node $\boldsymbol{v_i}$ to each of the centroid clusters in $\boldsymbol{v_c}$. The elements of the affinity matrix $\boldsymbol{C} \in \mathbb{R}^{n\,x\,m}$ are computed as follows:

$$\mathbf{c_i} = \begin{cases} \delta_{i,j} & \text{if } v_i \in V_C \\ \text{sparsemax}\left(\beta_i \frac{\mathbf{x_i}^{(l)} \cdot X_C^{(l)}}{\|\mathbf{x_i}^{(l)}\| \|X_C^{(l)}\|}\right) & \text{otherwise} \end{cases}$$

Where $\delta_{i,j}$ represents the Kronecker delta function defined as:

$$\delta_{ij} = \begin{cases} 0 & \text{if } i \neq j, \\ 1 & \text{if } i = j. \end{cases}$$

On the other hand, the "sparsemax" function ensures the assignment of each node to a single centroid, providing the option of dispensing the need for entropy minimization. "The idea is to set the smallest probabilities of a vector "z" to zero and keep only probabilities of the highest values of z" (Larionov, 2020).
β_i regularizes the amount of attention for each node, thus, it's natural for it to inversely depend on the shortest path distance between a node $i \notin \boldsymbol{V_c}$ (represented as $\boldsymbol{d_{i,v_c}}$) and any centroid node in $\boldsymbol{V_c}$.

$$\beta_i = \frac{\mathbf{1}}{\boldsymbol{d_{i,v_c}}}$$

Pooling: Finally, in order to produce a coarsened representative graph $\boldsymbol{G^{(l+1)}} = \langle \boldsymbol{V^{(l+1)}}, \boldsymbol{A^{(l+1)}}, \boldsymbol{X^{(l+1)}} \rangle$ we need to update our current adjacency matrix $\boldsymbol{A^{(l+1)}}$ and our current node embeddings $\boldsymbol{X^{(l+1)}}$ as follows.

$$A^{(l+1)} = C^{(l)\intercal} A^{(l)} C^{(l)} \in \mathbb{R}^{|V_C^{(l)}| \times |V_C^{(l)}|}, \qquad X^{(l+1)} = M_\Psi\left(C^{(l)\intercal} X^{(l)}\right)$$

$M_\psi$is a neural network with trainable parameters ψ.
Notice that our adjacency matrix of the new layer $\boldsymbol{A^{(l+1)}}$ is now a square matrix containing the "m" centroid nodes previously chosen, confirming that the new graph has shrunk.

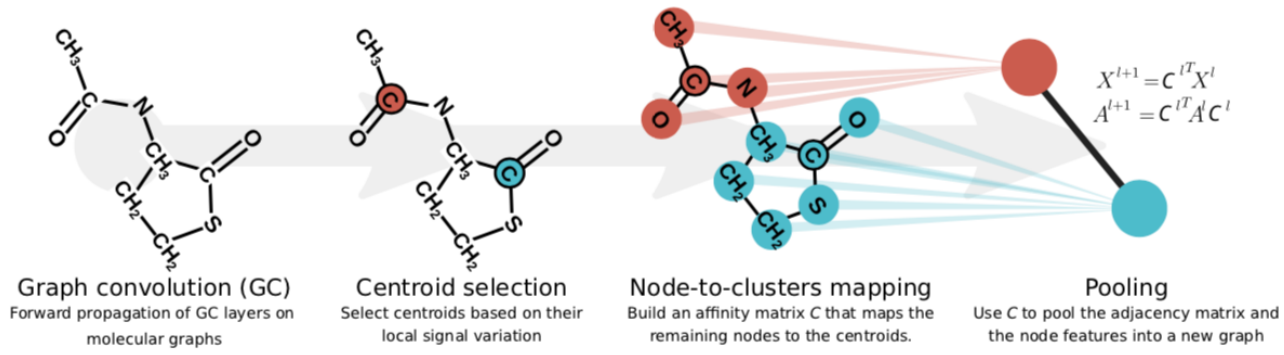

Fig. 2: Pooling process (Noutahi et. al, 2020)

Implementation details

Libraries and versions:
The python implementation of the Laplacian Pooling was developed with the torch-geometric library of version 1.6.3. Please refer to the following table for information of the most relevant libraries used and their version.

Name	Version
torch-geometric	1.6.3
scipy	1.5.4
scikit-learn	0.24.1
python	3.6.13
numpy	1.19.2
networkxx	2.5.1

Flags inside the code:
The network library is used for computing the shortest path for the affinity matrix C in the "node-to-clustering mapping" stage of pooling. This computation can represent a burden to the device, for this reason, the user has the option of disabling the computation of β_i by setting the flag **use_shortestPath = False** when initializing the pooling layer inside the model instantiation function.
E.g: self.pool1 = LaPool(use_shortestPath = False).
By doing this, β_i will not be computed and it will be assumed to be always equal to 1.

Private methods:
__AdjToGraph(self,A): use networkx to convert a given adjacency matrix "A" into a graph "G"
__GraphToTorch(self, G): Convert a given networkx graph G into pytorch geometric data type.
__AdjToEdge(self, A): Convert a given adjacency matrix "A" into a scipy sparse matrix.
__Get_shortest_path_length(self,G, src, trgt): Compute the shortest path between two given nodes (src and trgt) on a given networkx graph G.
__Sparsemax(self, z): implement the aforementioned sparsemax function. This implementation was taken from the article written by Larionov, 2020 listed in the references section of this document.
__Pool(self): The main LaPool implementation

Inputs:
- For instantiating a LaPool object the only needed parameter is the aforementioned **"use_shortestPath"** which is optional and True by default.
  self.pool1 = LaPool(use_shortestPath = False)
- For using the main method of pooling one must call the **"apply(x, edge_index, edge_attr, batch)"** the inputs are all attributes of an object of type pytorch Data.

x, edge_index, edge_attr, batch = self.pool1.apply(x, edge_index, edge_attr, batch)

Outputs:
The apply method mentioned before returns the coarsened version of the input parameters.

Plotting the graphs:
The networkx library is also used for plotting and showing the current Graph. This function can be used by calling the public method **"demo_init(data, show_labels = False)"** for visualizing the initial structure of the data.

E.g:
g = LaPool(False)
g.demo_init(data, show_labels = False)

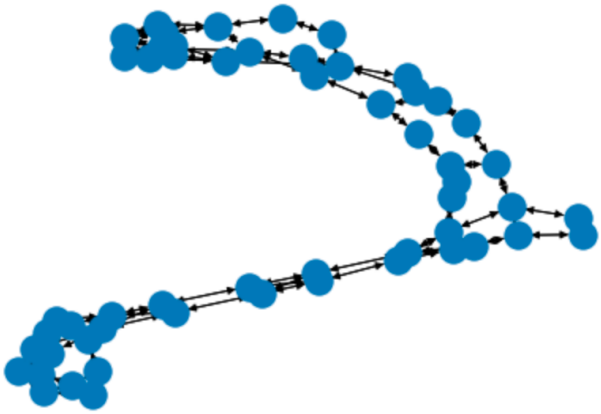

Fig. 3: Initial graph of a small dataset

After pooling, we can simply call the public method **"ShowGraph(self, show_labels = True)"** to visually show results of the pooling layer.

x, e, w, b = g.apply(data.x, data.edge_index, data.edge_attr, None)
g.ShowGraph(False)

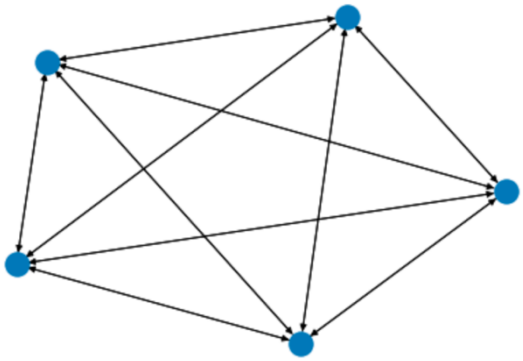

Fig. 4: Graph after pooling

Conclusion

Deep learning methods can also be applied for graph structures using some analogies with the image deep learning we all know and considering some other differences. One of these differences lies in the nature of the structure of the data. Meanwhile in an image matrix, we have a notion of space in every pixel, in a graph there is no such notion since any node can belong to any space and connected nodes cannot directly depict the distance between them. For this reason, pooling layers had to find different methods for graph coarsening, some of these methods rely on filtering operations of the graph's signals (spectral methods) while others use a message passing approach (spatial methods).
Pooling layers are a critical part of a GCN since they allow us to represent graph information in a more compact way that is suitable for processing. Different pooling methods can suit better for different situations. One of the advantages that LaPool layer has is the ability to consider both, the node's features and the graph structure, making it perfect for tasks such as enzyme analysis.

References:

Jepsen, Tobias Skovgaard. "How to Do Deep Learning on Graphs with Graph Convolutional Networks." *Medium*, Towards Data Science, 7 May 2019, towardsdatascience.com/how-to-do-deep-learning-on-graphs-with-graph-convolutional-networks-7d2250723780.

Larionov, Michael. "What Is Sparsemax?" *Medium*, Towards Data Science, 16 Feb. 2020, towardsdatascience.com/what-is-sparsemax-f84c136624e4.

Noutahi, Emmanuel, et al. "Towards Interpretable Molecular Graph Representation Learning." *Research Gate*, Jan. 2020, www.researchgate.net/publication/339642828_Towards_interpretable_molecular_graph_representation_learning.

Qimai Li, Zhichao, and Wu Xiao-Ming. "Deeper Insights into Graph Convolutional Networks for Semi-Supervised Learning." *ArXiv*, 22 Jan. 2018, arxiv.org/pdf/1801.07606.pdf.

Stern, Fabian. "Program a Simple Graph Net in PyTorch." *Medium*, Towards Data Science, 22 May 2020, towardsdatascience.com/program-a-simple-graph-net-in-pytorch-e00b500a642d.

Trudeau, Richard J. "A Beginner's Guide to Graph Analytics and Deep Learning." *Pathmind*, 2019, wiki.pathmind.com/graph-analysis.