A review of Graph Neural Networks

Sofia Larsson
Modulai AB, Stockholm, Sweden

October 2020

Introduction

Graphs, commonly referred to as networks, are ubiquitous structure as a wide range of domains employ them to capture relationships (edges) between different entities (nodes). For example in e-commerce, the network of items and users are modelled as a graph to predict new links representing a purchase or exploit interactions between users and products to produce high-quality recommendations. In chemistry, molecules are modelled as graphs; with atoms as nodes and edges as their bonds, to predict their bioactivity for drug discovery. Thus, as graphs are the founding structure in a wide variety of systems, providing efficient and reliable algorithms to model graph networks is of great importance.

Graph Neural Networks (GNNs) has emerged as a generalisation of neural networks that learn on graph structured data by exploiting and utilizing the relationship between data points to produce an output. These architectures aim to solve (i) node representation, (ii) link prediction or (iii) graph classification related tasks. Contrary to the Euclidean space, operations that we take for granted such as convolution becomes increasingly difficult in the non-Euclidean setting due to the lack of grid structure. Furthermore, the basic assumption that all data points are independent no longer holds true as all data points are related to each other by edges.

As tremendous effort has gone towards generalising definitions and operations to the graph setting, GNNs have expanded explosively with novel architectures and been introduced to a broader range of applications. Today GNNs can be divided into multiple categories: Graph Convolutional Neural Networks (GCNs), Recurrent Graph Neural Networks (RGNNs), Graph Attention Networks (GANs), Graph Auto-encoders (GAEs), Graph Spatial-temporal Networks (GSTNs) and Graph Reinforcement Learning, all of which continue to grow with new methods and models.

How do you adapt a neural network to graph structures?

One of the first models proposed within GNNs are RGNNs. They were originally proposed to extend Recursive Neural Networks in 2009, which also operates on graphs but require them to be directed and acyclic. RGNNs loosen this restriction by handling any type of graph.

To each node i in the graph, a state $\boldsymbol{h}_i \in \mathbb{R}^d$ is attached that depends on the nodes' features and the features of the nodes in its neighbourhood. An output if subsequently produced based on these states. Given a graph $\mathbb{G}$ with features $\boldsymbol{X} \in \mathbb{R}^{dxn}$, the states $\boldsymbol{h}_i$ and output $\boldsymbol{o}_i$ are defined for all pairs $(i, j) \in \mathbb{E}$ by

$$\begin{aligned} \boldsymbol{h}_i &= f(\boldsymbol{h}_j, \boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{w}_{ij}) \\ \boldsymbol{o_i} &= g(\boldsymbol{h}_i, \boldsymbol{x}_i), \end{aligned} \tag{1}$$

Figure 1: Graph visualizing the Cora dataset. The colors display classes.

where $\boldsymbol{x}_i$, $\boldsymbol{h}_j$, $\boldsymbol{x}_j$ and $\boldsymbol{w}_{ij}$ denote the feature of node i, the states and features of the neighbors to node i and the edge weights. The local transition function, f is a parametric function describing the dependence of node i on its neighborhood and g is the *local output function*. This can be written in a more compact form

$$\begin{aligned} \boldsymbol{h} &= F_\theta(\boldsymbol{h}, \boldsymbol{x}, \boldsymbol{w}), \\ \boldsymbol{o} &= G_\theta(\boldsymbol{h}, \boldsymbol{x}), \end{aligned} \tag{2}$$

and $\boldsymbol{\Theta} = [\boldsymbol{\theta}_F, \boldsymbol{\theta}_G]$ with $\boldsymbol{\theta}_F$ being the parameters of F and $\boldsymbol{\theta}_G$ being the parameters of G, $\boldsymbol{x}$ is the features of all the nodes in the graph and $\boldsymbol{w}$ is all the edge weights in the graph. The functions F and G are referred to as the *global transition function* and *global output function*.

The system in 2 have an unique solution if F is a contraction map with respect to the state, i.e $\exists \mu \in [0, 1) : ||F(\boldsymbol{h_i}, \boldsymbol{x}, \boldsymbol{w}) - F(\boldsymbol{h_j}, \boldsymbol{x}, \boldsymbol{w})|| \leq \mu ||\boldsymbol{h_i} - \boldsymbol{h_j}||$ holds for any $\boldsymbol{h_i}, \boldsymbol{h_j}$, according to Banach's fixed point theorem. Further, the theorem suggest the following recursive update scheme for the states

$$\boldsymbol{h}(t+1) = F_\theta(\boldsymbol{h}(t), \boldsymbol{x}, \boldsymbol{w}), \tag{3}$$

where $\boldsymbol{h}(t)$ denotes the t^{th} update of $\boldsymbol{h}$, where we let $\boldsymbol{h}(0)$ be set to an arbitrary initial value. Thus, by applying this operation to the states repeatedly, this update scheme will converge exponentially fast to the fixed point solution. This stable state is subsequently used to produce the output $\boldsymbol{o}$, a decision regarding the node.

The optimal weights of $\boldsymbol{F}$ and $\boldsymbol{G}$ are found by minimizing a given loss function. Note that to update the weights, the derivatives have to be calculated back through the contractions of F. Thus, to compute the gradients efficiently, an algorithm based on the Almeida-Pineda algorithm was proposed. The definition of this algorithm can be found in [4].
Even tough this architectures was proposed already back in 2009, it took some time before GNNs gained attention. One of the major breakthroughs in this area was when GCNNs were introduced and especially when spatial-based GCNNs were proposed. But let's not jump ahead, but start by explaining GCNNs.

Graph Convolutional Neural Networks

In the Euclidean space, the convolution between two functions g and h

$$(g * h)(\boldsymbol{x}) = \int_{\Omega} g(\boldsymbol{x} - \boldsymbol{x'})h(\boldsymbol{x'})d\boldsymbol{x'}. \tag{4}$$

For graphs, the operation $\boldsymbol{x} - \boldsymbol{x'}$ is not defined. Thus, we need to find another way to define the convolution operation in the non-Euclidean setting. According to the Convolution theorem, it is possible to express the convolution between to functions g and h as the element-wise product of their Fourier transform. But what is the Fourier transform of a graph?

In classical Fourier analysis, the eigenfunctions to the one-dimensional Laplacian that solves the eigenvalue equation

$$\Delta \boldsymbol{u}_i = \lambda_i \boldsymbol{u}_i, i = 0, 1, .. \tag{5}$$

are the complex exponentials, i.e the Fourier basis. This can be extended to a non-Euclidean space where the eigenvectors of the graph Laplacian can be interpreted as a generalized Fourier basis in the graph setting. The graph Laplacian is defined as $\Delta = \boldsymbol{X}^{-1}(\boldsymbol{D}\boldsymbol{A})$, where $\boldsymbol{D}$ is the degree matrix and $\boldsymbol{A}$ is the adjacency matrix.

By combining the convolution theorem with the generalised Fourier basis, the convolution of two function on the nodes of the graph be defined as

$$(f * g)(\boldsymbol{x}) = \sum_{i \geq 0} \langle f, \boldsymbol{u}_i \rangle \langle g, \boldsymbol{u}_i \rangle \boldsymbol{u}_i(x), \tag{6}$$

.
where $\boldsymbol{u}_i$ is the i^{th} eigenvector of the graph Laplacian. In matrix notion, it can be written in a more compact form: $\boldsymbol{g} * \boldsymbol{f} = \boldsymbol{u}\text{diag}(\hat{\boldsymbol{g}})\boldsymbol{u}^T\boldsymbol{f}$ where $\hat{\boldsymbol{g}}$ is the spectral representation of the filter $\boldsymbol{g}$.

The use of the spectral representation of g is the reason for this category of GCNN being named spectral-based GCNN. The spectral approach does however suffer from limitations. By defining convolution in this fashion, the filters are no longer shift-invariant but dependent upon position. Small perturbations of the graph will result in a change of eigenbasis, so the model will not generalise to new, unseen graphs. Lastly, the eigendecomposition of the Laplacian is of order $\mathcal{O}(n^3)$ and this not computationally efficient when the graph becomes lager than a few thousand nodes.

To tackle these drawbacks, some simplifications and approximations have to be made. Hammond et al. [2] proposed that the convolution between $\boldsymbol{g}$ and $\boldsymbol{f}$ can be approximated by low-ordered truncated Chebyshev polynomials of the k^{th} order. Thus, we can approximate the convolution as an expansion of Chebyshev polynomials as

$$\boldsymbol{g} * \boldsymbol{f} \approx \sum_{k=0}^{K} \boldsymbol{g}_k T_k(\tilde{\boldsymbol{\Delta}})\boldsymbol{f} \tag{7}$$

,
where $\tilde{\boldsymbol{\Delta}}$ is the re-scaled normalized Laplacian, $\tilde{\boldsymbol{\Delta}} = \frac{2}{\lambda_{max}}\boldsymbol{\Delta} - \mathbb{I}_n$ and the T_k's are the Chebyshev polynomials recursively defined as $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$ with $T_0(x) = 1$ and $T_1(x) = x$. Hammond et al. chose these polynomials because they have shown to approximate minimax polynomials, which are polynomials that minimize the maximum error given by $\max_{a<x<b}|f(x) - p(x)|$ for a function f defined on the interval $[a, b]$. One could consider other polynomials for the approximation, but this will require some further investigation in the matter.

Note that the expression in Eq. (7) only depends on nodes that are k-steps away in the graph. The complexity in Eq. (7) is $\mathcal{O}(|\mathcal{E}|)$, linear in the number of edges.

Kipf et al. [3] simplified this further by only including Chebyshev polynomials of the first order, motivating that higher order connections in the graph can be accessed by performing the convolution multiple times. Further, they approximate $\lambda_{max} \approx 2$, expecting that the neural network will adapt to this during training. Thus, Eq. (7) simplifies to

$$\begin{aligned} \boldsymbol{g} * \boldsymbol{f} \approx \sum_{k=0}^{1} \boldsymbol{g}_k T_k(\tilde{\boldsymbol{\Delta}}) \boldsymbol{f} &= \boldsymbol{g}_0 \boldsymbol{f} + \boldsymbol{g}_1 \tilde{\boldsymbol{\Delta}} \boldsymbol{f} \\ &= \boldsymbol{g}_0 \boldsymbol{f} + \boldsymbol{g}_1 (\boldsymbol{\Delta} - \mathbb{I}_n) \boldsymbol{f} \\ &= \boldsymbol{g}_0 \boldsymbol{f} - \boldsymbol{g}_1 (\boldsymbol{D}^{-1/2} \boldsymbol{A} \boldsymbol{D}^{-1/2}) f. \end{aligned} \tag{8}$$

Furthermore, they constrain the expression by letting $\boldsymbol{g} = \boldsymbol{g}_0 = -\boldsymbol{g}_1$, resulting in the following approximation

$$\boldsymbol{g} * \boldsymbol{f} \approx \boldsymbol{g}(\mathbb{I}_n - \boldsymbol{D}^{-1/2} \boldsymbol{A} \boldsymbol{D}^{-1/2}) \boldsymbol{f}. \tag{9}$$

By using a re-normalization trick $\mathbb{I}_n - \boldsymbol{D}^{-1/2} \boldsymbol{A} \boldsymbol{D}^{-1/2} \rightarrow \tilde{\boldsymbol{D}}^{-1/2} \tilde{\boldsymbol{A}} \tilde{\boldsymbol{D}}^{-1/2}$ with $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \mathbb{I}_n$ and $\tilde{\boldsymbol{D}}_{ii} = \sum_j \tilde{\boldsymbol{A}}_{ij}$, they rewrite this as

$$\boldsymbol{g} * \boldsymbol{f} \approx \boldsymbol{g}(\tilde{\boldsymbol{D}}^{-1/2} \tilde{\boldsymbol{A}} \tilde{\boldsymbol{D}}^{-1/2}) \boldsymbol{f}. \tag{10}$$

The update in Eq. (10) is then applied for GCNs and simultaneously extended by letting each node exhibit a feature vector of dimension d. Thus by replacing the signal $\boldsymbol{f}$ by the features $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, and using y filter maps we have

$$\boldsymbol{H} = \tilde{\boldsymbol{D}}^{-1/2} \tilde{\boldsymbol{A}} \tilde{\boldsymbol{D}}^{-1/2} \boldsymbol{X} \boldsymbol{\Theta}, \tag{11}$$

where $\boldsymbol{\Theta} \in \mathbb{R}^{d \times y}$ is the matrix of filter parameters and $\boldsymbol{H} \in \mathbb{R}^{n \times y}$ is the convolved signal matrix. This filtering operation has the time complexity $\mathbb{O}(|\mathcal{E}|dy)$. Thus, for the neural network model, we get the following update scheme for a two-layer GCN

$$\boldsymbol{o} = softmax(\tilde{\boldsymbol{D}}^{-1/2} \tilde{\boldsymbol{A}} \tilde{\boldsymbol{D}}^{-1/2} \sigma(\tilde{\boldsymbol{D}}^{-1/2} \tilde{\boldsymbol{A}} \tilde{\boldsymbol{D}}^{-1/2} \boldsymbol{X} \boldsymbol{\Theta}^{(0)}) \boldsymbol{\Theta}^{(1)}), \tag{12}$$

where $\boldsymbol{o}$ is the output of the model.

By using the adjacency matrix in the update instead on the Laplacian, we take advantage of the spatial relations in the graph, thus this is called the spatial based approach. From this, a more general update scheme has been developed called the message passing scheme, which just as the spatial approach utilizes the spatial relations in the graph and updates the representation of a node by sending information to its neighbors by edges [1].

The k^{th} layer of an arbitrary GNN is

$$\boldsymbol{m}_i^{(k)} = aggregate^{(k)}\big(\{\boldsymbol{h}_j^{(k-1)} : j \in \mathcal{N}(i)\}\big), \quad \boldsymbol{h}_i^{(k)} = combine^{(k)}\big(\boldsymbol{h}_i^{(k-1)}, \boldsymbol{m}_i^{(k)}\big), \tag{13}$$

where $\boldsymbol{h}_i^{(k)}$ is the feature vector of node i at the kth layer. The GNN models are distinguished by the choice of the *aggregate* and *combine* functions. The GCN model uses a mean-pooling function as aggregation function and ReLu as a combination function. For RGNN, the aggregation respective combination function are the function F and G, and are the same for each layer k. These networks are trained like any other neural network, using stochastic gradient descent or a variant thereof and

back-propagation to optimize the filter weights.

Kipf et al. simplifications and the derivation of the message-passing scheme have made GNNs more accessible and applicable. However, important questions still remains to be addressed. What is the best architecture for a specific task? Are the same optimization algorithms still suitable to optimize the network as in the Euclidean setting? Does the graph structure affect the models' ability to converge and in that case - to what extent? How much does the model rely on the structure of the graph and how much on the information provided by the features?

For graph classification and prediction tasks, it has been shown that the discriminate power of GCNNs Kipf. et al introduced is no better than the Weisfeiler-Lehman (WL) graph isomorphism test, which is a test known for distinguish a broad range of graphs [6]. The test asks if two graphs have identical topology and GCNNs have surprisingly difficulty in distinguish between simple structures. However, the models still performs well on node representation task.

Furthermore, on some standard datasets, it has been experimentally observed that the performance of the GNN model drops with increasing numbers of layers, which is counter-intuitive as it would suggest that aggregating more information from a greater neighborhood and thus accessing more information in the graph results in lower performance [3]. This has further been investigated and a possible explanation is that the information will spread differently in the graph depending on the sub-graph in the local neighborhood [7]. If the subgraph is dense, the information will spread quickly in the local neighborhood, while if the graph is sparser, the information needs to be aggregated multiple times in order for it to reach the same number of nodes as in the denser case. In the study, a graph containing tree-like substructures, only aggregating information from a 2-hop neighborhood resulted in miss-classifications while incorporating information from a 3-4 hop neighborhood classified more correctly. On the contrary, subgraphs were the information can spread explosively, 3-4 layer models tend to make the wrong classification while 2 layer GNNs are more accurate.

Multiple studies have also shown that by adding connections in the graph to access more global information improves performance than just aggregating information in the local neighborhood.

A simple experiment

A simple experiment to test how the overall structure of the graph affects the performance is by perturbing the structure and observe how the accuracy is affected. This was done on the three standard and publicly available dataset Cora, Citeseer and Pubmed. All three dataset are citation networks with the articles as nodes and the references as edges. The objective is to categories the articles into different subjects. In both the Cora and Citeseer datasat, each paper has a 0/1-valued vector indicating the presence/absence of words from a dictionary containing 1433 unique words. The words was collected by a stemming and stop word removal process. In Pubmed, each publication is described by an information retrieval weighted word vector, which indicates how important the word is in the publication, from a dictionary containing 500 unique words.

```python
import torch
import torch.nn as nn
from dgl.nn.pytorch import GraphConv

class GCN(nn.Module):
    def __init__(self,
                 g,
                 in_feats,
                 n_hidden,
                 n_classes,
                 n_layers,
                 activation,
                 dropout):
        super(GCN, self).__init__()
        self.g = g
        self.layers = nn.ModuleList()
        # input layer
        self.layers.append(GraphConv(in_feats, n_hidden, activation=activation))
        # hidden layers
        for i in range(n_layers - 1):
            self.layers.append(GraphConv(n_hidden, n_hidden, activation=activation))
        # output layer
        self.layers.append(GraphConv(n_hidden, n_classes))
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, features):
        h = features
        for i, layer in enumerate(self.layers):
            if i != 0:
                h = self.dropout(h)
            h = layer(self.g, h)
        return h
```

Listing 1: Simple implementation of a GCNN by using their Graph Convolution module. More examples are found at *https://github.com/dmlc/dgl*

A 2-layer GCNN is fitted to each of these datasets, with the graphs perturb by a percentage of the edges added or removed. The models are implemented using *Deep Graph Library* (dgl) [5]. An example implementation of a GCNN is shown in Listing 1. Each training was repeated 10 times and the result is shown in Figure 2. Not surprisingly, the graph structure has an impact on the optimization and generalisation capabilities of the models, indicating that the topological structure is exploited during training. However, it is quite surprising that the performance of the GCN models appear to be more sensitive to decreasing number of edges, than to increasing number of edges. A possible explanation could be that the models are able to adjust to the added noise by setting weights to zero but cannot compensate for the lost information with removed edges.

An interesting aspect to note is the percentage of edges required to be removed before the accuracy drops for each dataset. The GCNs trained on Citeseer and Pubmed experience a drop in test accuracy when the edges are reduced by 20% respective 70%, while the test accuracy of the model trained on Cora declines after 40% of the edges are removed, with a slower rate.
One possible explanation is that the features of Pubmed contains richer information than the features in Cora and Citeseer making it more robust against perturbation of the graph structure. Another possible explanation could be that Pubmed contains fewer classes, making it less sensitive to the number of neighbors the nodes are connected to. It could also possibly be explained by the distribution of edges in the graph; if Pubmed contains a higher share of nodes connected to a higher number of nodes, it will be less perceptive to eliminated edges.

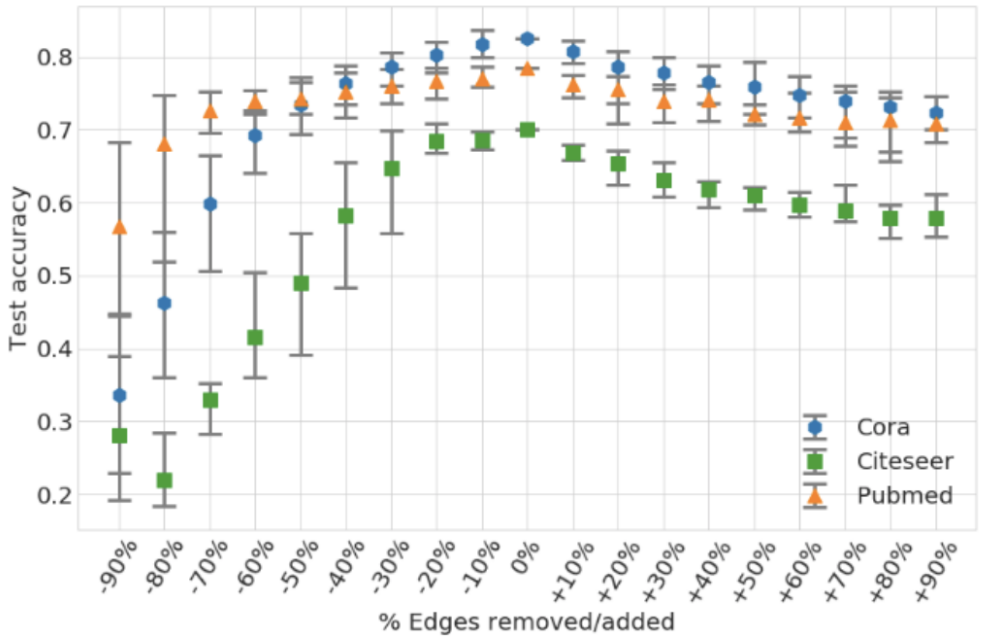

Figure 2: The test accuracy for Cora (blue), Pubmed (red) and Citeseer (green) is plotted against the percentage of edges removed respectively added to the network. All models were trained 10 times and an average was taken of the runs. The errorbars represent the maximum and minimum gained accuracy of the 10 runs.

Conclusion

In this article, we have briefly introduced GNNs and explained the idea behind RGNNs and GCNNs. We have discussed interesting questions that remain to be addressed for these exciting models, and dived deeper into an experiment that investigates the influence of the structure on the performance of a GCNN model. The accuracy incline differently for each datasets and we hypothesize its due to the edge distributions. The information propagation in the graph depends on the connectivity between nodes and the nodes' representation is updated by the messages passed from neighboring nodes. A more qualitative comparison between different graph types is needed to elucidate this hypothesis. Other factors can also affect the result, such as small datasets, the number of classes and the size of and information in the feature vectors, and a controlled analysis with several graphs could shed a light on each of these factors' influence.

References

[1] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry, 2017.

[2] David K Hammond, Pierre Vandergheynst, and Rémi Gribonval. Wavelets on graphs via spectral graph theory, 2009.

[3] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

[4] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[5] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks, 2020.

[6] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2019.

[7] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks, 2018.