Getting started with Graph Neural Networks

Antoine Moulin
Mathematics, Vision, Learning (MVA)
ENS Paris-Saclay
antoine.moulin@ens-paris-saclay.fr

Claire Vernade
DeepMind, UK
claire.vernade@gmail.com

Rémy Degenne
CWI Amsterdam
remydegenne@gmail.com

Abstract

Graph Neural Networks (GNNs) are models that operate on graph-structured data. In recent years, it became a very active area of research: many models have been developed and it has been used in different fields, such as computer vision, chemistry, and others. Here, we focus on the paper [1, Battaglia et al. 2018], which aims to gather many different approaches under a common framework. The authors introduce the so-called *graph network* block, designed to be the building block for learning from relational data in deep learning architectures. It is based on three principles: flexible representations of the data, a configurable within-block structure to control the information made available to the different functions, and composable multi-block architectures so that different blocks can be combined to construct complex models. We start by stating some definitions and challenges any graph neural network model faces. Then, we highlight the main differences with the previous frameworks and finally, we focus on an example and applications.

1 Introduction

A graph is a data structure used to represent *relations* between different *entities*. For instance, a mass-spring system can be seen as a graph where nodes are the different masses and edges are the springs that connect pairs of nodes. For a social network, nodes correspond to the users, and edges represent friendships (in which case the graph is undirected, such as Facebook) or follower relationships (in which case the graph is directed, such as Twitter). Another example is given by molecules - which can be naturally seen as graphs - where nodes are the atoms and edges are the interactions between atoms.

There is an increasing amount of data generated from complex processes that cannot fit into a Euclidean representation, and for which graphs are well suited. From scene understanding to molecules classification problems, graph-related techniques are being broadly applied. After the success of deep learning in computer vision and other fields, deep learning architectures for graphs are now being developed. Besides, the previous examples show that one can face different kinds of problems. When dealing with a mass-spring system, one could be interested in predicting future positions of the masses, i.e. solving a node regression task. With a social network, link prediction can be of great interest, which corresponds to an edge classification task. Finally, being able to tell whether or not a molecule is likely to be a drug corresponds to a graph classification task.

GNNs have been first introduced with [2, Gori et al. 2005]. In these original works, the authors propose an extension of recursive neural networks that can be applied to any graph. Based on a dynamical system, they update the node states until it converges (which is theoretically guaranteed because a contracting mapping is used). The constraints induced by the contracting mapping raised

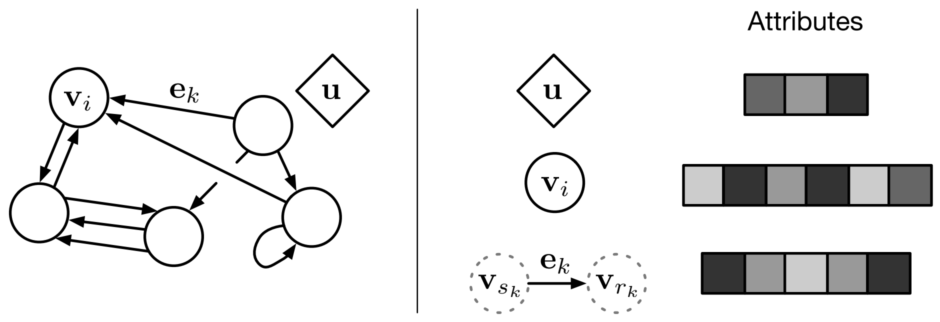

Figure 1: An example of graph as defined in section 2.1. (Illustration from [1].)

some issues, but the main idea was here: develop a model that directly process a graph and learn some function of the nodes by iteratively propagating and aggregating information through the graph. Since then, this area of research has been growing quickly: GNNs have been used in many different contexts (see section 5) and plenty of versions have been developed, often with impressive results.

Here, we focus on [1, Battaglia et al. 2018] (the notations used are those in the paper) and some of the ideas proposed afterward. In section 2, we give some definitions and the main challenges one has to deal with when building a GNN. In section 3, we present the background on which [1, Battaglia et al. 2018] is based and some other works. In section 4 is exposed the framework from [1, Battaglia et al. 2018] with the *graph network* block and its different components. Besides, it will be linked to the previous section. Finally, in section 5 are presented some examples and applications.

2 Definitions and challenges

2.1 Graph

Before diving into the formalism of the GN block, let's give the definition of a graph that we will be using here (which can be found in [1]). By *attribute*, we mean "properties that can be encoded as a vector, set, or even another graph". A *graph* is a tuple $G = (\mathbf{u}, V, E)$ such that:

- G has a global attribute $\mathbf{u}$.
- G has attributed vertices $V = \{\mathbf{v}_k\}_{k=1:N^v}$.
- G has directed, attributed edges $E = \{(\mathbf{e}_k, r_k, s_k)\}_{k=1:N^e}$ where r_k, s_k are the indices of the receiver and the sender, respectively.
- G is a multi-graph, i.e. there can be more than one edge between vertices, including self-edges.

In the following, unless it is specified, a graph is defined like that. An illustration is given in figure 1. Besides the examples given in section 1, a graph could also represent a road network between different towns. The nodes would be the towns across a country, and the edges would be the roads between them. Then, the task could be to find the shortest path between two towns. In this case, the node attributes could be for instance a position or a boolean indicating if a node is the beginning or the end of the path to find. The edge attributes could be the distances between two nodes. This example is detailed in section 5.

2.2 Permutation invariance and isomorphism

Some deep learning architectures can encode spatial translation invariance (such as Convolutional Neural Networks that use the same filters on all the pixels of an image) or temporal translation invariance (such as Recurrent Neural Networks whose recurrent unit is shared over time). One of the motivations of graph neural networks is that in many cases, the objects involved in a learning task do not have a natural order (how could we order the towns from the previous example?) and that there is no deep learning model that has some kind of permutation invariance. If one trains a model using

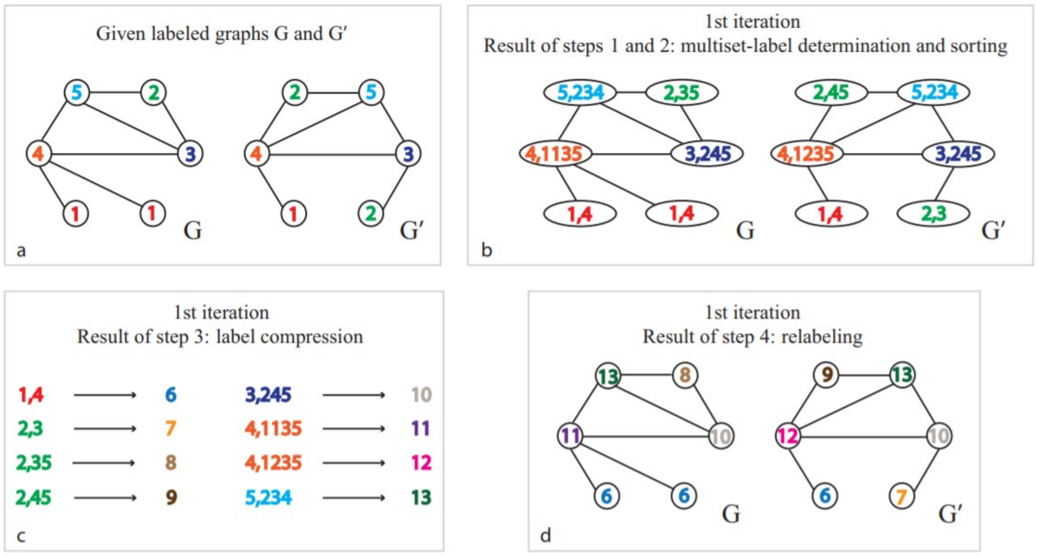
Given labeled graphs G and G′
a
G
G′
1st iteration
Result of steps 1 and 2: multiset-label determination and sorting
b
1st iteration
Result of step 3: label compression
1,4 → 6
2,3 → 7
2,35 → 8
2,45 → 9
3,245 → 10
4,1135 → 11
4,1235 → 12
5,234 → 13
c
1st iteration
Result of step 4: relabeling
d

Figure 2: One iteration of the Weisfeiler-Lehman graph isomorphism test. Note that here, one iteration is enough to see that G and G' are not isomorphic. (Illustration from [4].)

some features $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ and that a new input has an ordering $(\mathbf{x}_n, \mathbf{x}_2, \ldots, \mathbf{x}_{n-1})$, the model may completely fail. Similarly, if one indexes the nodes of a graph in two different ways, it will yield two different adjacency matrices, which can be an issue. Hence, we introduce the notion of equivariance and invariance functions.

For an order-k tensor $T \in \mathbb{R}^{n^k}$ and $\sigma \in \mathcal{S}_n$, we define the operation: $(\sigma \star T)_{\sigma(i_1),\ldots,\sigma(i_k)} = T_{i_1,\ldots,i_k}$, which permutes the indices along each axis of the tensor T with the same permutation σ.

- A function $f : \mathbb{R}^{n^k} \to \mathbb{R}^l$ is said to be *invariant* if $f(\sigma \star T) = f(T)$, i.e. if the output does not depend on the input's ordering.

- A function $f : \mathbb{R}^{n^k} \to \mathbb{R}^{n^l}$ is said to be *equivariant* if $f(\sigma \star T) = \sigma \star f(T)$, i.e. if the output's ordering remains the same than the input's ordering, even in case of a permutation.

A graph G can be seen as a 2-tensor using the adjacency matrix $A \in \mathbb{R}^{N^v \times N^v}$ that contains N^e nonzero elements. Denote by $X \in \mathbb{R}^{N^v \times d}$ the vertices' attributes. Let P be a permutation matrix. In this case, the definitions can be reformulated as: f is invariant if $f\left(PAP^T, PX\right) = f(A, X)$, and f is equivariant if $f\left(PAP^T, PX\right) = Pf(A, X)$. With these definitions in mind, we notice that when performing graph-level tasks, a GNN has to be invariant. When performing node-level tasks, a GNN has to be equivariant.

Besides, we say that two graphs G_1 and G_2 are *isomorphic* if there exists σ such that $G_1 = \sigma \star G_2$. An important question is which graphs can GNNs distinguish, and what does this depend on? While the performance of a GNN depends on its update and aggregation functions (see section 4), it has been shown in [3, Xu et al. 2019] that an upper bound is given by the Weisfeiler-Lehman graph isomorphism test, which we quickly define here.

Given two graphs G and G' (for which we assume to have generated some labels for the nodes, if not available), we would like to know whether or not G and G' are isomorphic. The key idea of the Weisfeiler-Lehman test is to iteratively (i) update the node labels using the sorted set of neighboring node labels, (ii) naming it as a new label. We iterate this process until the label sets of G and G' differ or the number of iterations reaches the number of nodes, in which case either the graphs are isomorphic, either the test failed to discriminate them. An iteration is showed in figure 2.

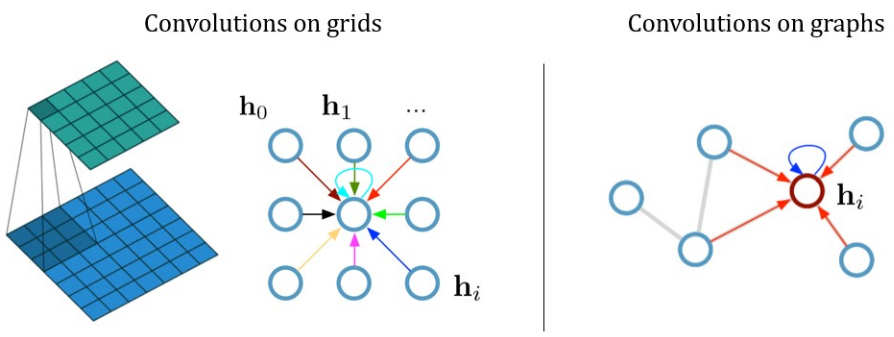

Figure 3: Convolution on a grid and its possible extension on a graph. Note that while the weights corresponding to the neighbors on a grid are all different, they are shared across neighbors on a graph. $\mathbf{h}_i \in \mathbb{R}^d$ are the hidden layer activations of a pixel or a node. (Illustration from Thomas Kipf.)

2.3 Convolution on a graph

Given the success of CNNs, there have been some researches about how to apply a convolution on a graph. The goal for GNNs is to find a way to efficiently use the local information available in a node's neighborhood. Unlike an image that can be represented as a 2D-grid, there is no notion of space in a graph, and the neighborhood's size can vary. Different ideas were proposed. Some are designed as "spectral" methods as they rely on a computation that involves the Fourier transform of a graph signal (and thus the eigendecomposition of the Laplacian matrix). Those methods require to compute the eigendecomposition of the Laplacian matrix for each graph and thus are computationally expensive. Hence, we do not give more details about them and instead focus on "spatial" methods.

As a reminder and using notations from figure 3, a convolution on the node $\mathbf{h}_4 \in \mathbb{R}^d$ from a grid can be performed as:

$$\mathbf{h}'_4 = \sigma\left(\mathbf{W}_0\mathbf{h}_0 + \mathbf{W}_1\mathbf{h}_1 + \cdots + \mathbf{W}_9\mathbf{h}_9\right)$$

A convolution on a graph, can be performed as:

$$\mathbf{h}'_i = \sigma\left(\mathbf{W}_0\mathbf{h}_0 + \mathbf{W}_1 \sum_{j\in\mathcal{N}_i} \mathbf{h}_j\right)$$

where σ is a nonlinear function. Note that because of the sum, this convolution does not depend on the neighbors' order. There are many variants (e.g. using an attention mechanism) but the idea remains the same: update the node embeddings, aggregate it (here with the sum), and so on.

3 Background and other works

3.1 Background

Many papers proposed different approaches for GNNs before [1, Battaglia et al. 2018]. Here, we only present the one from [5, Gilmer et al. 2017], which was already an attempt of unification and which will appear as a particular case of the framework mentioned here in the section 4.

Message-Passing Neural Networks (MPNNs).

MPNNs were defined in [5, Gilmer et al. 2017] as a class of models that learn a message-passing algorithm and an aggregation procedure to compute a function of an entire input graph. The goal was to predict properties of molecules much faster than Density Functional Theory (DFT), which is usually used in chemistry for approximations. This framework unifies different approaches, such as [6, Kipf et al. 2016]'s one.

Let G be a graph. The forward pass has two phases, a *message-passing* phase and a *readout* phase:

- The message-passing phase runs for T time steps and uses message functions ϕ_t^e and vertex update functions ϕ_t^v, where t stands for the time step. During this phase, the vertices' features $\mathbf{v}_i$ are updated ($\mathbf{v}_i^{(0)} \triangleq \mathbf{v}_i$), based on messages coming from the neighborhoods and through the edges as:

$$m_i^{t+1} = \sum_{k:r_k=i} \phi_t^e \left(\mathbf{e}_k, \mathbf{v}_{r_k}^{(t)}, \mathbf{v}_{s_k}^{(t)}\right) \triangleq \rho^{e\to v}\left(E_i^{(t)}\right) \quad \text{where } E_i^{(t)} = \{(\mathbf{e}_k, r_k, s_k)\}_{r_k=i}$$

$$\mathbf{v}_i^{(t+1)} = \phi_t^v \left(\mathbf{v}_i^{(t)}, m_i^{(t+1)}\right)$$

In other words, here, the edges are just a way for the vertices to communicate information, and the information is aggregated with the summation operator.

- The readout phase computes a feature vector for the whole graph using some readout function ϕ^u according to

$$\mathbf{u} = \phi^u \left(\left\{\mathbf{v}_k^{(T)}\right\}_{k=1:N^v}\right)$$

The functions ϕ are all learned and differentiable. As ϕ^u operates on the set of vertices' features, it must be invariant to permutations in order for the MPNN to be invariant to graph isomorphism (see section 2).

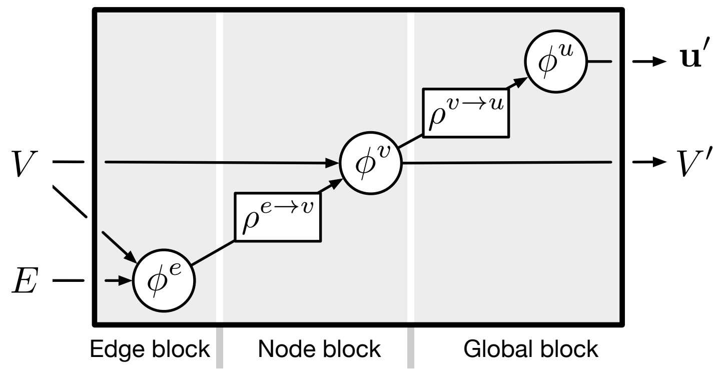

Figure 4: MPNN block. (Illustration from [1].)

3.2 Other works

Besides MPNNs, many different approaches have been proposed. For instance, [6, Kipf et al. 2016] introduced a Laplacian based method, which falls in the "spectral" methods category mentioned before. [7, Veličković et al. 2017] presents a model in which the update uses an attention mechanism. Non-Local Neural Networks from [8, Wang et al. 2017] take inspiration from the non-local mean operation.

For a more complete list, one can check the recent review from [9, Wu et al. 2019].

4 The framework: Graph Network (GN)

As said previously, [1, Battaglia et al. 2018] introduces a general framework, the *graph network* (GN) block, unifying several approaches (such as MPNNs or NLNNs). It leverages every step of the computation by formulating it in a general form, enabling anyone to custom and synthesize new architectures that express a strong relational inductive bias. In other words, this paper does not introduce a new graph-related technique strictly speaking but highlights all possible degrees of freedom in an architecture using GN blocks.

The GN block is a graph-to-graph module, based on three design principles:

- **Flexible representations**. The global, node, and edge attributes of a GN block can be represented with arbitrary formats, such as sets, sequences or graphs. Most of the time however, vectors or tensors are used (one per edge or node, and one for the global attribute) so a GN can be used with other blocks such as a CNN or a RNN. Besides, when representing the input data as a graph, there are two possibilities: (i) the input explicitly specifies the structure, (ii) it must be inferred or assumed. In the second case, if the entities (i.e. the nodes) are not explicit, they can be for example inferred by a neural network. If the relations (i.e. the edges) are not explicit, one can use a fully-connected graph or try inferring sparse relations between the nodes (sparsity is often desirable, especially with graphs where it can speed up the computations).

- **Configurable within-block structure**. The structure and functions within a GN block can be configured in different ways, which offers flexibility in what information is made available as inputs to its functions, as well as how output edge, node, and global updates are produced. The different functions within a GN block are explained in this section. An illustration of the general GN block is given in figure 5, and an illustration of a MPNN, which can be seen as a particular case, is given in figure 4.
- **Composable multi-block architectures**. The third principle of GNs is that it must be composable so that one can build complex architectures. The graph-to-graph interface ensures that the output of one GN block can be passed as input to another, even if their internal configurations are different. This allows arbitrary numbers of GN blocks to be composed. They can also be used within a CNN or RNN architecture. Besides, the blocks can be shared (as in unrolled RNN) or not.

Now, let's explain the structure of a GN block. It contains three "update" functions, ϕ, and three "aggregation" functions, ρ. ϕ^e is used for the per-edge updates, ϕ^v for the per-node updates and ϕ^u for the global attribute update. $\rho^{e\to v}$ is used to aggregate information from the edges and sends it to the nodes, $\rho^{e\to u}$ serves a similar purpose but sends information to the global attribute, and $\rho^{v\to u}$ aggregates information from the nodes and sends it to the global attribute:

$$\begin{aligned}
\mathbf{e}'_k &= \phi^e\left(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}\right) & \bar{\mathbf{e}}'_i &= \rho^{e\to v}\left(E'_i\right) & \bar{\mathbf{e}}' &= \rho^{e\to u}\left(E'\right)\\
\mathbf{v}'_i &= \phi^v\left(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}\right) & \bar{\mathbf{v}}' &= \rho^{v\to u}\left(V'\right) & &\\
\mathbf{u}' &= \phi^u\left(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}\right) & & & &
\end{aligned}$$

where $E'_i = \left\{(\mathbf{e}'_k, r_k, s_k)\right\}_{r_k=i,k=1:N^e}$, $E' = \left\{(\mathbf{e}'_k, r_k, s_k)\right\}_{k=1;N^e}$ and $V' = \left\{\mathbf{v}'_k\right\}_{k=1:N^v}$. Note that the functions ρ must be invariant to permutations. Under this form, the procedure for an update is as follows:

1. Compute the per-edge updates $\mathbf{e}'_k$ with the function ϕ^e, which could correspond to the estimated time to get from one town to another (e.g. depending on the traffic). Then aggregate the information for every receiver node with the function $\rho^{e\to v}$ (e.g. a driver looks at the possible directions and choose to take the quickest route) and the message for the global attribute with the function $\rho^{e\to u}$ (e.g. the total estimated time or distance between two towns).
2. Compute the per-node updates $\mathbf{v}'_i$ with the function ϕ^v, using both the current nodes' attributes and the edge messages (i.e. the $\bar{\mathbf{e}}'_i$). For instance, this update could give a piece of information about whether or not the driver will have to go through a certain town. Then aggregate the new attributes with the function $\rho^{v\to u}$ to send a message to the global attribute (e.g. the number of towns to go through).
3. Compute the global attribute update $\mathbf{u}'$ with the function ϕ^u, using the current global attribute, the message from the edges $\bar{\mathbf{e}}'$ and the message from the nodes $\bar{v}'$.

An illustration of such a procedure is given in figure 5. The given order is arbitrary and can be changed, i.e. one could have started to compute the per-node updates and used a function $\rho^{v\to e}$.

Other examples than MPNNs can be found in [1] and have different structures than showed in figure 4, but what we see with figures 4 and 5 is that [1] highlights all the possible configurations of a network that can process graphs and packs everything in an easy-to-use block, i.e. the authors present a model compatible with the current deep learning frameworks.

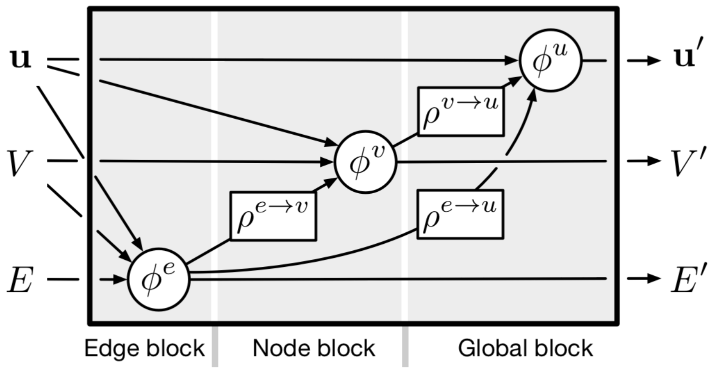

Figure 5: A full GN block. Every degree of freedom is illustrated in this block. The ρ functions must be invariant to permutations, and the ϕ functions can be neural networks, such as MLPs. (Illustration from [1].)

5 Example and applications

5.1 Shortest path problem

Let's consider the shortest path problem. The code is available at https://tinyurl.com/gn-shortest-path-demo and comes from the demonstrations of the `graph_nets` [1] library. Let's define the GNN, step by step.

5.1.1 The graph

The graphs are generated as follows: (i) N^v nodes are uniformly sampled in a 2D rectangle, (ii) N^v weights are sampled according to an exponential law and assigned to each node, (iii) edges are generated according the geographical threshold model (which just says that two nodes are connected if their weights verify a certain relation) and (iv) the graph is combined with a minimum spanning tree to ensure the graph is connected.

Here, the edges have two attributes represented in an array:

- `distance`, which gives the Euclidean distance between two nodes,
- `solution`, which is a boolean to be determined and indicates if the edge belongs to the predicted path. The second attribute is not necessary, as it will be in the nodes as well, but shows that edge classification can be done.

The nodes have five attributes represented in an array:

- `pos`, which gives the position of a node in $\mathbb{R}^2$,
- `weight`, which has been sampled from an exponential law,
- `start`, which indicates if the node is the source of the path,
- `end`, which indicates if the node is the target one wants to reach,
- `solution`, which indicates if the node belongs to the path (to be determined).

And the global attribute can be defined as the length of the path, which is stored in an array as well.

5.1.2 The within-block structure

Within each block of the model used, the update functions ϕ^e, ϕ^v and ϕ^u are all defined as a Multi-Layer Perceptrons (MLPs) with two hidden layers that contain 16 hidden units each. They have to be trained.

The aggregate functions ρ^e, ρ^v and ρ^u are all defined as the sum operator.

[1]https://github.com/deepmind/graph_nets

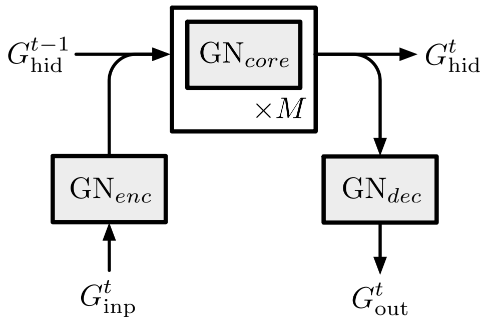

Figure 6: Configuration of the GNN used for solving the shortest path problem. (from [1].)

5.1.3 The configuration

The configuration of the GNN is an autoencoder formed by three components:

- an encoder, which learns an embedding for the nodes, edges and the global attribute,
- a core module that performs several message-passing phases with the encoder's output and a hidden state, in a recurrent fashion,
- a decoder, which decodes the output of the code module.

The core module is where the problem is solved. The configuration is showed in figure 6.

5.1.4 Analysis

After around 2000 iterations of training, the GNN reaches around 0.99 of accuracy. In figure 7, we can see some message-passing steps of the GNN on a graph, as well as the ground-truth for the shortest path between the white and black nodes. Red corresponds to a low probability of being in the path, and blue corresponds to a high probability. The bold arrows correspond to the ground-truth path.

Here is what we can say about the different steps.

- In the first step, the model tries the different edges but does not have a strong clue about the path to follow (all the nodes have more or less the same color).
- In the third step, it starts to form a path between the two nodes and highlights some nodes in blue (see the third picture of figure 7), whereas the others are left in red. If there were 3 message-passing steps, it is likely that the GNN would have come up with the wrong solution.
- In the fifth step, it notices another interesting path and until the last step, it will switch from the previous path to this new one which appears to be the ground-truth solution and will change the colors accordingly: the nodes that were blue in the third step become red, and the nodes from the new path become blue. The model succeeded to find the correct solution.

The resolution done by the GNN seems similar to the Bellman-Ford algorithm, which is usually used for solving shortest path problems. As a reminder, the algorithm is given by:

Algorithm 1: Bellman-Ford

```
Data: A graph $G = (V, E)$ and a source $s$.
Result: Distances from the source to the other nodes, $d$.
Initialization of $d$
for $k = 1, \dots |V| - 1$ do
    for $t \in V$ do
        $d' = \min_{u:(u,t)\in E} d[u, k-1] + w(u, t)$
        $d[t, k] = \min(d[t, k-1], d')$
    end
end
```

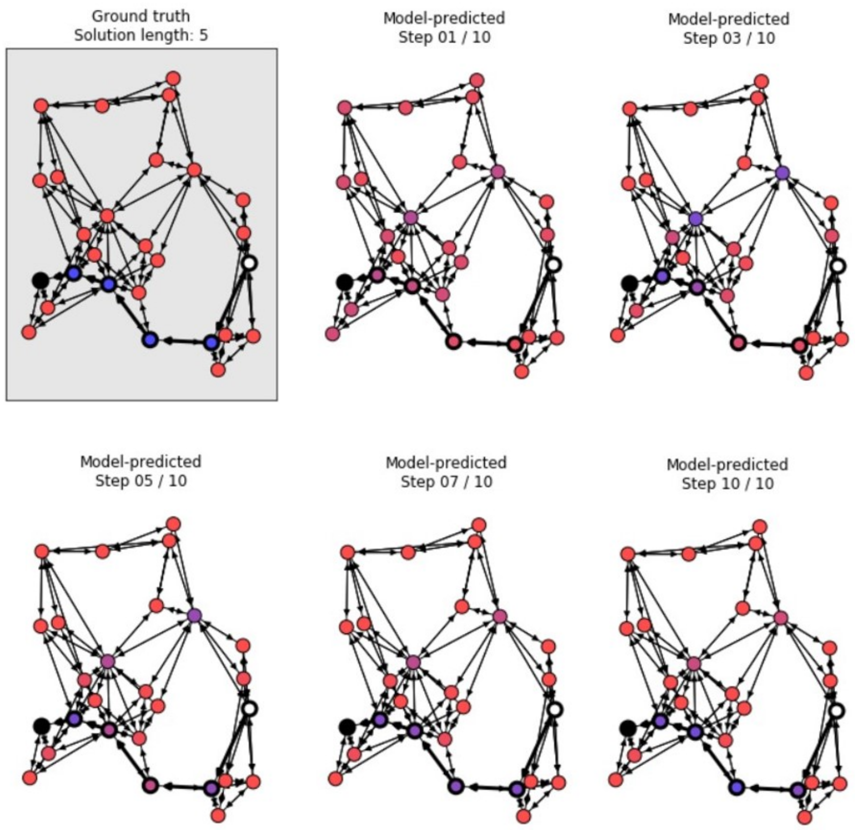

Figure 7: The ground truth for the shortest path between the white and the black nodes, and the different steps processed by the GNN. Blue corresponds to a high probability of being in the path, whereas red coresponds to a low probability.

It appears that the GNN actually mimics the Bellman-Ford algorithm, much better than a single MLP would. This is shown in [10], where the authors introduce a theoretical framework of *algorithmic alignment*. They show that under some assumptions, a GNN aligned with a reasoning function g can learn this function g. In our case, the GNN is "aligned" with the Bellman-Ford algorithm because the configuration of the model, that performs several message-passing steps, encode the "for" loops which allows the small MLPs to focus on the two lines of computations, whereas a single huge MLP would have to learn both the "for" loops and the computations of the distances.

5.2 Applications

In this section, we just give a quick overview of the possible applications of GNNs. References are not given but can be found either in the aforementioned review [9] or elsewhere.

- **Computer Vision**. As many images are provided with a textual description, GNNs can be used in semantic segmentation or visual reasoning. Indeed, images can contain many objects and understanding the relationships between them helps characterize their interactions and can be very useful. It can also be used as an embedding block in an autoencoder network, such as U-Net. Another application concerns action recognition, where the joints of a skeleton naturally form a graph that can be used.
- **Physics and chemistry**. An interesting application is to predict the dynamics of physical systems, such as the mass-spring system mentioned previously. GNNs appear to perform much better than other models such as LSTMs (probably because of an "alignment" between a GNN and the task to solve). In chemistry, we already mentioned the example of predicting molecule properties with [5].

- **Recommender systems**. A recommender system models relations between users or consumers and items. Its goal is to score the importance of an item to a user, which can be seen as an edge-level task.
- **Social network analysis**. Some papers have used GNNs in the context of influence maximization in a social network. It could also be used to study the contagion of nodes in a graph.
- **Programs correction**. A program coded in some language can also be seen as a graph with the different variables and functions. GNNs have been used for auto-completion or to correct code.

The range of applications is wide and this list is far from being complete, but it is here to give some examples.

Acknowledgments

This review was done during a course given by Michal Valko for the Master Mathematics, Vision, Learning (MVA) from ENS Paris-Saclay, and was supervised by Claire Vernade and Rémy Degenne (warm thanks to them for their precious advices).

References

[1] Peter W. Battaglia et al. Relational inductive biases, deep learning, and graph networks, 2018.

[2] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks.*, July.

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

[4] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *J. Mach. Learn. Res.*, 12:2539–2561, 2011.

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

[6] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2016.

[7] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2017.

[8] Xiaolong Wang, Ross Girshick, Abhinav Gupta, and Kaiming He. Non-local neural networks, 2017.

[9] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks, 2019.

[10] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S. Du, Ken ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about?, 2019.