GNN ISP specification

Artem Kopyl

December 10, 2024

1 Introduction

There may be countless examples, where we need to describe data containing relational information between entities (chemical compounds, social networks, dependency schemes, etc.). Moreover, we would like to make an inference and gain insight into potential dependencies or trends. There are many possible approaches to this task, the most popular one being graphs. A graph is a mathematical structure that models pairwise relations between objects. Mathematics has a powerful framework for working with and using these structures for analysis. Further, we will employ this useful concept to solve our task.

2 Problem definition

Let's assume that we have a dataset of arbitrary domains, where each record describes some entity. Those data entries can contain both numerical and categorical values. The type of target problem is not relevant right now, but we can assume regression/classification for example. We suppose that such a dataset consists of:

- **Descriptive features** - data describing the properties of the entity. These are often represented as predefined metrics, characterizing each entity in some way (height, color, age, etc.).
- **Relational features** - metrics that denote the property of a subset of entities. Such data can vary in "form" for each specific domain, like interactions between users, the geographical position of an object, or a chemical bond between atoms in a molecule.

This issue can be solved by employing a graph as a data representation structure. As mentioned above, each node will represent a single entity/record in our dataset. Then we can enhance this structure by creating edges between nodes if their relationship satisfies some custom constraint. Being consistent with the mentioned domains, here are possible options:

- Create an edge, between two properties, if they are close enough (distance between them does not exceed some threshold)

- Connect two nodes, in case we have a record, that an entity interacts in some way with another one
- Add a link between atoms in a molecule if their chemical bond is of a certain strength/polarity

One can name many more possible scenarios for using this mechanism to *enhance tabular data* with relational data. With this representation of data, we can apply **Graph Neural Networks (GNNs)** [2] on it to employ all available information.

3 Theory behind GNNs

Graph Neural Networks is a deep learning approach for graph and relational data analysis. Its main difference from regular MLPs/NNs is the usage of graph convolution operators. In a nutshell, each such model consists of:

- Encoder - sequential *graph convolution* layers, that enable us to learn meaningful entity representation.
- Predictor - an MLP that will make actual predictions, according to the chosen problem type.

Graph convolution is the main trick to solve our problems. At this step, relational features are used to derive embeddings that incorporate available relationships in our data. This leads us to employ all the information for further inference. Additionally, these convolution operators use only order-invariant aggregation functions. This deals with the "order dependence" issue, which was mentioned previously. Let's explore this mechanism in depth.

3.1 Graph convolution

Graph convolution is an operation of updating the state of a node using available data about its neighbors and the state of the current entity. Talking about state embeddings, we consider them to be numerical vectors in the space of arbitrary dimensionality. Each node has its representation in the chosen embedding space, which is constructed on top of its descriptive features in some way. It follows, that after conducting a convolution, the node's state will capture information about its "surroundings" in the radius of one *node hop*. We can extend this to a wider range by adding several sequential convolution layers in our network. That will ensure that each node (after it was processed by *Encoder*) holds information about other entities as far as the number of convolution layers. This mechanism creates an *influence region* around each entity, allowing us to employ available relational information fully.

Additionally, we can elaborate on this and add weighting operators, different aggregating functions, or other mechanisms for combining information to our taste. Also, we can take into account different types of relations between objects

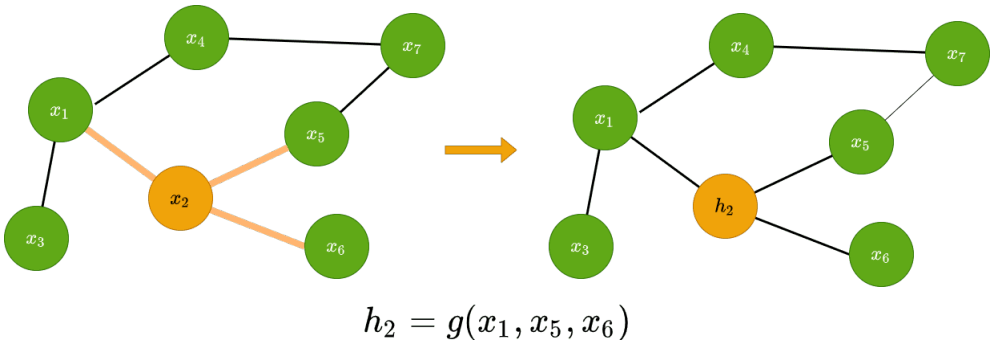

Figure 1: Message passing concept
Graph convolution illustration [6]. Assuming node x_2 as our target, we change its embedding using neighbors around it (denoted with yellow edges) and some *order invariant* aggregation function g(). Other nodes are not affected by this operation.

and, respectively, specially treat them. As you can see, the process of combining data is highly customizable, which leads to its big potential. There exists a general technique, which encapsulates this whole process. It is called **"Message Passing"** and can be described with the formula [1]:

$$h_t^n = q\left(h_{t-1}^n, \bigcup_{\forall n_j : n \to n_j} f_t(h_{t-1}^n, k, h_{t-1}^{n_j})\right)$$

where the notation is:

- h_n^t - embedding of node n in the *t-th* passthrough
- $q()$ - update function for combining current node state with "received messages" from the neighboring nodes
- $\bigcup_{\forall n_j : n \to n_j}$ - *order invariant* aggregation function, that combines all adjacent edges to node n
- $f_t()$ - generating *edge embedding*, based on properties of end-nodes
- k - indicator, which denotes type of the connection $n \to n_j$. With the help of this parameter, we can customize the process of edge embedding generation.

3.2 Convolution operators

As previously mentioned, it is possible to modify the data combining step in this mechanism according to different requirements. There exists a huge variety of convolution operators to effectively capture all kinds of entity relationships and properties. Here are some examples to name a few:

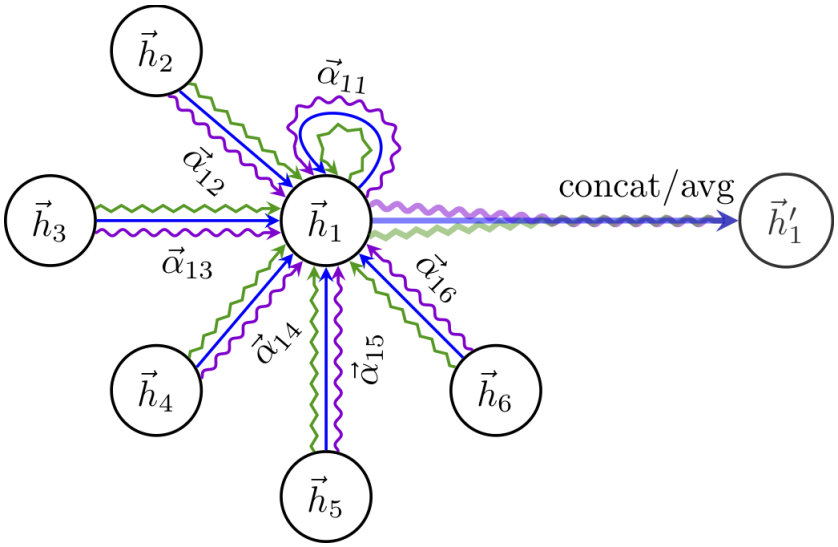

Figure 2: GAT convolution

The illustration shows multi-head attention (with K=3 heads) applied to node h_1 and its neighbors. Various arrow styles and colors represent separate attention processes. The attention coefficients for each head are denoted by vectors $\alpha_{i,j}$. The combined features from the heads are either concatenated or averaged to form h'_1

- "SAGEConv" operator [5]

$$h_i^{t+1} = W_1 h_i^t + W_2 * mean_{j \in N(i)}(h_j^t)$$

- "GraphConv" operator [8]

$$h_i^{t+1} = W_1 h_i^t + W_2 \sum_{j \in N(i)} e_{j,i} h_j^t$$

- "GATConv" operator [10] (Figure 2)

$$h_i^{t+1} = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j \in N(i)} \alpha_{i,j}^k W_k h_j\right)$$

Additional notation here is:

- W_n - weight matrix (usually a learnable parameter for the neural net)
- $N(i)$ - set, containing all neighbours of node i
- $mean_{j \in N(i)}$ - aggregating all presented values with an average
- $e_{i,j}$ - weighting factor for the edge $i \to j$

- K - number of attention heads
- $\alpha_{i,j}^k$ - attention coefficient (k-th head) for the nodes i and j
- $\sigma()$ - nonlinear function (e.g., sigmoid)

This variety in existing operators makes GNNs a powerful tool for data analysis and Deep Learning in general. Due to this, we can apply the mechanism to different domains and types of analysis tasks.

3.3 Possible tasks for GNNs

Speaking of graph structures, we can find a lot of interesting approaches on how to analyze our relational data. It is possible to analyze the whole network as one entity, deriving some of its properties, or we could focus on its parts (*subgraphs*) and infer relationships between node clusters. Here are some example tasks which GNNs can solve:

- *Link prediction* - GNN aims to uncover the connections among entities within graphs while also attempting to forecast the presence of links between two entities. For example, in social networks, it is crucial for inferring social interactions or recommending potential friends to users.
- *Graph clustering* - graph data clustering involves organizing data into groups represented by graphs. There are two primary methods of clustering for graph data. Vertex clustering focuses on grouping the nodes of the graph into clusters of closely interconnected regions, utilizing either edge weights or distances. The other method treats entire graphs as objects to be clustered, grouping them based on their similarity.
- *Node classification* - the objective is to assign labels to nodes by examining the labels of their "neighbors". Typically, such problems are trained using a semi-supervised approach, where only a portion of the graph is labeled.

4 Data Enhancement

Considering such powerful tool as GNNs at hand, we can try to apply relational information insights in practice. As a task for this project, I have chosen to work with geo-positional data. Suppose that we have a dataset of housing prices for some region. Each record corresponds to some property, denoting its main *descriptive features* as Lot Size, Total Area, Number of Bedrooms, etc.

Firstly, we need to decide on how to create edges/links between entities in our enhanced data model. Assuming that relational data is positional in some Euclidean space, we can come up with a **constraint** for **distance metric**, that will indicate for us the presence of a relationship between two entities. Thus the whole enhancement process can be divided into the following steps:

1. Distance metric and Metric constraint selection.

2. Calculate the metric's value for each pair of nodes.

3. Populate graph with edges, according to metric constraint fulfillment.

It is important to mention that this algorithm can be modified to decrease its time complexity. For example, if the selected metric constraint does not require any relationship comparison, we could create edges on the fly, by checking on constraint fulfillment right after the metric value is calculated. This trick could save us a lot of time, although it is not always applicable. According to this principle, we can create a graph, representing our data model in the dataset.

4.1 Distance metric

I have mentioned that we are going to employ *distance metric* as a parameter to our constraint mechanism. The most obvious choice (since we are using geo-positional relational data) is to calculate the distance between two positions given by the object's coordinates, using *Haversine formula*. But it is not the only option. We can introduce another metric space by changing the subject feature of the relationship. For example, we could consider the Number of Bedrooms as relational information and, respectively, change the distance metric to the absolute difference between two values. As you can see, the choices of relational feature and distance metric are bound and should work well together to encapsulate the targeted relational insight. This tuning is rather domain-specific, and one should seek the most suitable strategy for each use case independently.

4.2 Metric constraint

Having at hand a distance metric, we need to decide which edge-creating condition to choose. This indicator will create the structure of our data model, connecting entities with relationships. Here are some examples that are possible for geo-positional case:

- *Radius threshold*

  One possible option is to choose the neighborhood radius R. This means that a node A will be connected to all other nodes at a distance of at most R from A. A strategy like this preserves meaningful connections in our graph (edges are based on an absolute distance measure, ensuring that connected nodes are actually close in terms of the chosen distance metric) and guarantees that the created graph will be undirected.

- *K Nearest Neighbours*

  Edges are created between a target node A and at most K nearest entities (according to the distance metric). This mechanism can be useful for cases where it is important to maintain control over *nodes' degree* and *graph density*.

5 Enhancer system

5.1 Targeted task

Main task for the implemented system will be to search for the best positional encoders, that can be used in combination with GNNs. This will be achieved by iterative GNN training on different configurations and comparing the evaluation results for each option. After this analysis, the user will be provided with the resulting report on GNN's performance and enhanced dataset for further usage. The solution will have the following inputs/outputs:

- Input:
  - training dataset,
  - GNN implementation (or a configuration for GNN creation), and
  - Distance metric and constraint configuration / implementation.
- Output:
  - Enhanced data in the form of a graph structure, and
  - a performance report for the tested enhancers.

5.2 Implementation description

Let us now describe how the user will interact with the proposed solution. Firstly, we pass the dataset as an input to the Enhancer and denote target data features, which will be employed as the relational information (in our example – Longitude/Latitude). After that, the system will facilitate the comparison of different techniques for enhancing the data, similar to the role of a positional encoder in Kerlmer's article [7]. Its goal is to recommend the most effective encoding strategy that improves the prediction or classification performance of an employed GNN.

A GNN will be trained and tested (on respective data subsets) to compare enhancement mechanisms. In order to customize specific submodules, the system also accepts a configuration file with the description of the GNN's architecture. It should contain the following details:

- *Encoder*
  1. The type of convolution operators.
  2. Number and size of Encoder layers.
- *Predictor*
  1. Number and size of hidden MLP layers.
  2. Regularization dropout rate.
  3. Activation function.

Although the system will contain a predefined set of distance metrics and constraints to choose from, the user can pass a custom choice of those to be evaluated on the provided data. This configuration should be provided as two separate Python files with respective function implementations of:

1. Metric value calculation between two entities (conforming to MetricProtocol)
2. Constraint satisfaction calculation, based on a metric value (conforming to ConstraintProtocol)

Mentioned *Metric* and *Constraint* protocols will be a predefined Python interfaces in my solution. The user can rely on them while creating his own implementations, to make them compatible with the Enhancer system.

As an output, the user can expect to receive an enhanced data model in the form of a graph structure, with populated edges. Thanks to the usage of the best-performing enhancement strategy, this graph will ensure maximal informational gain. Additionally, a report will be generated, describing the performance of compared "edge-creating" mechanisms (GNN's train loss progression and test evaluation, graph features as density/average node degree, etc.). The state of the GNN with the highest evaluation will also be available for further usage.

6 Existing tools

Now we can discover existing frameworks and tooling in general for work with Graph Neural Networks. One can find many different options but I want to describe the most popular ones. They differ in various aspects, and that is why we cannot outline one of them as the best one. I would rather choose one out of them, that suits my needs the most in some specific scenario. Now, let us have a closer look at them:

1. **Deep Graph Library (DGL)**

   Deep Graph Library [3] is an open-source library that enables developers to efficiently create, train, and deploy GNNs across different deep learning frameworks such as PyTorch, TensorFlow, and MXNet. It is designed to strongly emphasize scalability and performance, particularly for large-scale graph data.

   Advantages:

   - Compatibility: a unique feature of DGL is its framework-agnostic approach, allowing users to build GNN models using the backend of their choice, whether it be PyTorch or TensorFlow. This flexibility is particularly useful for teams that work across different deep learning frameworks or who want to transition between frameworks with minimal changes to their graph-related code.

- Optimization: DGL is built to support efficient message-passing mechanisms, which are crucial for GNNs. It offers a flexible framework for defining custom GNN layers, as well as optimized implementations for common models like Graph Convolutional Networks (GCN), Graph Attention Networks (GAT), and GraphSAGE. One of DGL's key strengths is its high scalability, which is achieved through techniques like graph sampling, mini-batching, and parallelization, allowing it to handle very large graphs that may not fit into memory.

Disadvantages:

- Small model variety: compared to PyTorch Geometric, DGL offers fewer pre-built GNN models and layers, although the available models cover the basic needs.
- Small community: while DGL has grown in popularity, the community is still smaller compared to Graph Nets/PyG, leading to fewer tutorials, code examples, and discussions online.

2. **PyTorch Geometric (PyG)**

PyTorch Geometric [9] is a deep learning library built on top of PyTorch, specifically designed for graph-structured data. It has emerged as one of the most popular frameworks for building and deploying GNNs due to its ease of use, modular design, and integration with PyTorch. It provides a comprehensive set of tools for creating various GNN architectures, training them efficiently, and evaluating their performance across multiple domains.

Advantages:

- Ease of use: intuitive API and high extensibility, enable researchers and developers to easily prototype novel GNN architectures by combining various pre-built GNN layers. This feature also makes this framework beginner-friendly, allowing one to start implementing any idea with almost no time spent getting acquainted with the tool.
- Big community: over time, PyG has built a robust ecosystem and a strong community, which contributes to its rapid development and the availability of tutorials, examples, and support materials.
- Strong GPU acceleration: PyG leverages PyTorch's dynamic computation graph and automatic differentiation capabilities, making it simple to create custom GNN layers and loss functions while taking full advantage of the GPU acceleration PyTorch provides.

Disadvantages:

- Weak heterogeneous graphs support: While PyG supports heterogeneous graphs, it lacks the out-of-the-box, highly-optimized support frameworks like DGL offer for multi-relational and heterogeneous

graphs. This limitation can make working with complex, multi-relational data such as knowledge graphs harder.

- Sparse documentation: while the core functionalities are well documented, more advanced or custom usages may require a deeper dive into the source code, which can be time-consuming.

3. **Graph Nets by DeepMind**

   Graph Nets [4] is a research-oriented library developed by DeepMind and implemented in TensorFlow. It is part of a broader initiative to provide flexible tools for creating Graph Networks and models that operate on graph-structured data. Graph Nets is not as widely known or used as previously mentioned PyG/DGL, but it holds a unique place for those working closely with TensorFlow or who are focused on developing customized GNN architectures for research purposes.

   Advantages:

   - Modularity and flexibility: unlike PyG/DGL, Graph Nets does not come with a large collection of pre-built GNN layers or datasets. Instead, its core strength lies in its flexibility and the ability to customize each component of the graph network, from node embeddings to edge updates and global features. This flexibility makes Graph Nets an attractive option for researchers working on novel or experimental GNN architectures beyond standard models.
   - Research-Oriented: suited for theoretical research and novel model development, providing researchers with tools to create highly customized graph-based models. Offers full control over message-passing mechanisms, allowing for experimentation with non-standard graph networks.

   Disadvantages:

   - Tied to TensorFlow: Graph Nets is exclusively built for TensorFlow, limiting flexibility for users who prefer PyTorch or other frameworks. That also means transitioning from or integrating with non-TensorFlow workflows can be cumbersome.
   - Slow development and maintenance: While both PyG and DGL have thriving communities and regular contributions, Graph Nets has not experienced the same level of consistent updates or expansions in features, which can be a critical drawback, especially in a rapidly evolving field like graph-based deep learning. This slower development means Graph Nets may lag in adopting cutting-edge research innovations and optimizations.

In conclusion, the most suitable option for our use was the PyG framework. Several reasons came into play. Firstly, PyTorch support was crucial due to previous experience with this tool and its optimization advantages over other

popular Deep Learning libraries. Additionally, PyG's extensive collection of pre-built models lets us explore different approaches and try all possible combinations of mechanisms while solving our task.

References

[1] Miltos Allamanis. *MSR Cambridge Lecture Series: An Introduction to Graph Neural Networks: Models and Applications*. 2019. URL: https://www.microsoft.com/en-us/research/video/msr-cambridge-lecture-series-an-introduction-to-graph-neural-networks-models-and-applications/.

[2] Christopher M. Bishop and Hugh Bishop. "Graph Neural Networks". In: *Deep Learning: Foundations and Concepts*. Cham: Springer International Publishing, 2024, pp. 407–427. ISBN: 978-3-031-45468-4. DOI: 10.1007/978-3-031-45468-4_13. URL: https://doi.org/10.1007/978-3-031-45468-4_13.

[3] *Deep Graph Library official website*. https://www.dgl.ai/.

[4] *Graph Nets official GitHub repository*. https://github.com/google-deepmind/graph_nets.

[5] William L. Hamilton, Rex Ying, and Jure Leskovec. *Inductive Representation Learning on Large Graphs*. 2018. arXiv: 1706.02216 [cs.SI]. URL: https://arxiv.org/abs/1706.02216.

[6] Sergios Karagiannakos. *Best Graph Neural Network architectures: GCN, GAT, MPNN and more*. 2021. URL: https://theaisummer.com/gnn-architectures/.

[7] Konstantin Klemmer, Nathan Safir, and Daniel B. Neill. *Positional Encoder Graph Neural Networks for Geographic Data*. 2023. arXiv: 2111.10144 [cs.LG]. URL: https://arxiv.org/abs/2111.10144.

[8] Christopher Morris et al. *Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks*. 2021. arXiv: 1810.02244 [cs.LG]. URL: https://arxiv.org/abs/1810.02244.

[9] *PyTorch Geometric official website*. https://www.pyg.org/.

[10] Petar Veličković et al. *Graph Attention Networks*. 2018. arXiv: 1710.10903 [stat.ML]. URL: https://arxiv.org/abs/1710.10903.