Bachelor Research Project: Graph Machine Learning

Jannick Stuby
University of Stuttgart
st160888@stud.uni-stuttgart.de

Christian Staib
University of Stuttgart
staib.christian@hotmail.de

Lukas Zeil
University of Stuttgart
st161467@stud.uni-stuttgart.de

Abstract

We present our research on graph machine learning that we have made as part of the bachelor research project. We started our project examining DeepChem and MoleculeNet but later decided to focus on graph machine learning in general. During our project we had two main goals: to learn about graph machine learning in general and implement some graph machine learning algorithms ourselves. Therefore our report is structured in two parts: first we give an overview of graph machine learning and then we present our own implementations of some graph machine learning algorithms.

1 Introduction

1.1 Problem Domain

Neural Networks are thriving in the modern society with software ranging from facial recognition to language translation augmenting humans day-to-day. In this aspect, Machine Learning (ML) for chemical sciences gains more and more value and therefore is the focus of an increasing amount of research. In consequence, an increasing amount of data regarding molecules are gathered and labeled, multiplying the amount of information that can be used to train ML models to help chemical discovery and molecule analysis to aid chemical discovery for drug development and molecule investigation.

To aid in the improvement of ML models, benchmarks have been developed. A benchmark in a ML setting, is a site where models are compared against an accumulation of datasets, where the datasets are usually split into fixed training- and testing sets. The benchmark around which this project will be largely revolving is MoleculeNet. They tried a standardized approach for splitting their data, by creating more ways in which these datasets can be split, that can be applied to every dataset they have.

As the MoleculeNet paper was published in 2018, we try to find and use methods already used in other context but not in the paper. We will set up a dataset-to-metric model for each benchmark using new tools in four areas: splitters, featurizers, deep learning, and transfer learning. These results will be compared against the best-known results from MoleculeNet. While we may not be able to enhance the best-known results, we will formulate our accumulated knowledge for future exploration.

1.2 Definitions

A graph is a mathematical structure that consists of a set of vertices V and a set of edges E. Similarly a attributed graph is a mathematical structure that consists of a set of vertices and a set of edges, where each vertex and edge has an attribute, e.g. $A(V \bigcup E) \rightarrow \Sigma^*$. While this basic definition does

Preprint. Under review.

Table 1: Node feature vector

Index	Feature	Value Range
0	atom number (H excluded, C=5)	1 - 119
1	chirality	unspecified, Tetrahedral CW, Tetrahedral CCW, other and misc
2	degree	0 - 10, misc
3	formal charge	-5 - 5, misc
4	number of H-atoms	0 - 8, misc
5	number of radicals	0 - 4, misc
6	hybridization	SP, SP2, SP3, SP3D, SP3D2, misc
7	is aromatic	true, false
8	is in ring	true, false

Table 2: Edge feature vector

Index	Feature	Value Range
0	bond type	single, double, triple, aromatic, misc
1	bond stereo	none, z, e, cis, trans, any
2	is conjugated	true, false

not specify any conditions on the attributes, it is possible to limit the attributes to a certain set of values. Some graph machine learning methods require the attributes to be of certain types or to provide certain functionality.

1.3 MoleculeNet

MoleculeNet [10] was introduced as a comprehensive benchmark for comparing ML models in molecular sciences.

2 Preliminary

As we started our project examining DeepChem, we first looked into handcrafted graph to vector methods.

Later we looked into graph machine learning.

2.1 Open Graph Benchmark

Open Graph Benchmark(OGB) [3] is a framework for benchmarking machine learning on graphs, providing large-scale, real world datasets as well as an easy to use software framework for loading datasets and evaluating model performance. For comparing model performances, OGB provides leaderboards for different datasets and challenges, split into the sections Node Property Prediction, Link Property Prediction and Graph Property Prediction, as well as a large-scale challenge [2] leaderboard, which aids comparison of performance on very large graphs.

Our main focus will be Graph Property Prediction, mainly on the datasets ogb-molhiv and ogb-molpcba. Both these datasets are derived from MoleculeNet and respectively provide a large amount of molecules for molecular property prediction. Molecules are represented by graphs, where atoms are the nodes and corresponding chemical bonds are represented by the edges. Node features are represented as a 9-dimensional vector, edge feature vectors are 3-dimensional, the respective value ranges can be seen in Table 3 and Table 2. For ogb-molhiv the model is evaluated using ROC-AUC, for ogb-molpcba Average Precision is used.

We additionally used ogb-molfreesolv as a smaller dataset which is also derived from MoleculeNet. FreeSolv contains SMILES strings of molecules from the Free Solvation Database and consists of "experimental and calculated hydration free energy of small molecules in water" . The model is evaluated using Root Mean Square Error.

add citation see moleculeNet paper

We decided to use OGB as a benchmarking framework instead of the MoleculeNet benchmark results, because OGB submissions are much more versatile and active, containing results from many research teams with the latest submission for ogb-molhiv dating to May 2022, contrary to the MoleculeNet results which were provided by the authors of the paper only and are dated to January 2018.

2.2 Graph Embedding

TODO: Add information on graph embedding relevant to problem domain

2.3 node2vec

TODO: Add information about node2vec

2.3.1 doc2vec

TODO: explain doc2vec according to [4] approach with respect to problem domain, possibly move to another section (paragraph/word embedding)

2.3.2 graph2vec

TODO: explain graph2vec [5] approach with respect to problem domain

2.4 Transformers

TODO: add Transformer [7] section: add explanation, formulas, visualization

In 2017 Vaswani et al. proposed a new network architecture for NLP problems, called the Transformer [7]. The general model structure, which was used to describe the idea, can be seen in.

Transformers use Self-Attention to increase model performance.

2.4.1 Graphormer

As an adaption of Transformers [7] for graph representation learning, Graphormer [11] was presented by Ying et al., providing a performant Transformer model, excelling in various graph representation learning tasks.

Graphormer achieves such good results by introducing three types of encodings to incorporate structural information of graphs in the feature vectors of nodes and edges.

To capture information about the importance of a node in a given network, Graphormer uses a measure called node centrality by utilizing degree centrality of a given node. This is done by introducing two new vectors for each node, one for the degree of ingoing edges and one for the outgoing ones Figure 1. Those vectors are learnable scalars which are added to the node features, so node importance and semantic correlation are respected during self attention.

add more information about learnable scalars, add general transforer structure to explain queries and keys

The second encoding, called spatial encoding, is used to capture positional information for each node with respect to the network. This is achieved by using the distance of the shortest path between connected nodes and, once again, using a learnable scalar which is added as bias in the self-attention module. By doing that, the model can attend to a more relevant subset of the network according to the task. An example given in the paper is the possibility to increase attention to nodes surrounding a given node instead of nodes which lay further away.

Finally Graphormer introduces a method to include edge information in the model by adding an additional bias term to the attention module. This bias is created by averaging the dot-products of edge features and a learnable scalar of the edges describing the shortest path between each two connected nodes.

perhaps add equations according to Transformer and Graphormer paper

2.4.2 GraphGPS

L. Rampásek et al. introduced GraphGPS in their paper 'Recipe for a General, Powerful, Scalable Graph Transformer.' [6] The objective of this model recipe is to provide a foundation for incorporating

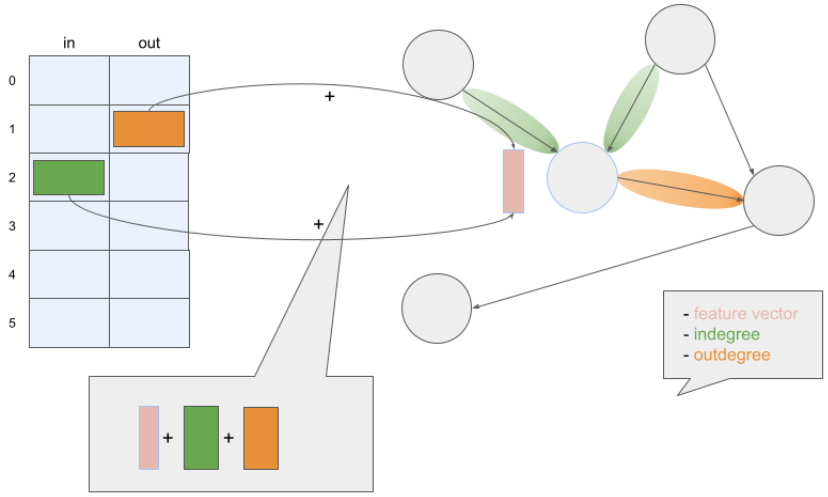

Figure 1: Centrality Encoding: learnable embedding vectors are added to feature vectors of nodes

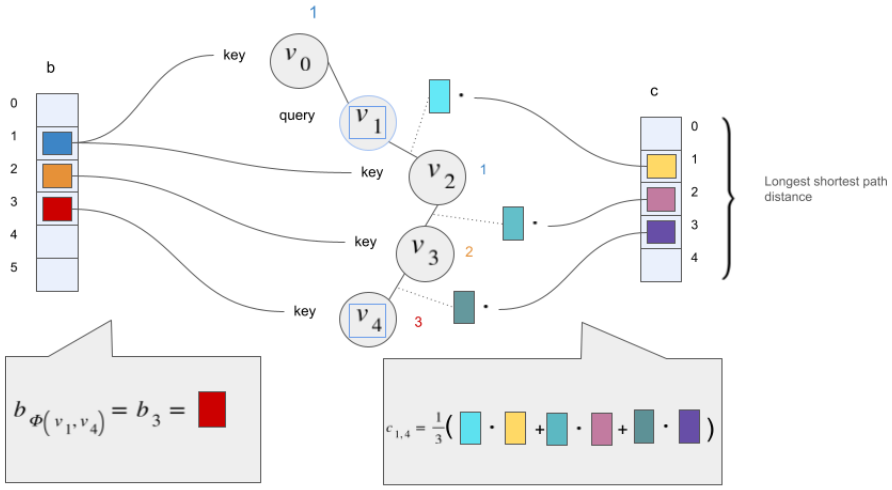

Figure 2: Spatial Encoding (left) and Edge Encoding (right) added as bias to Attention Module

structural and positional information into node and edge features, while maintaining the flexibility of the model that implements these features.

GraphGPS serves as a preprocessor for the actual model learning on the dataset. The authors tested GraphGPS with several models, such as Transformer, Performer, BigBird, Gine, and more. They designed it to allow models that permit only Node Features, known as the Global Attention layer, as well as models that permit both Node and Edge Features, known as the MPNN layer. Both layers can also be used in combination to integrate the distinct advantages of two models.

The authors categorized structural and positional information into two types since the distance between nodes, although indicating the graph's structure, is insufficient to capture structural similarities. They incorporated this information by utilizing structural and positional encodings (SE and PE), which are divided into Local, Global, and Relative categories. Local and Global encodings are implemented as node features, while Relative encodings are implemented as edge features.

2.4.3 Heterogeneous Interpolation on Graphs

A winning entry for the dataset ogb-molpcba, called HIG-GraphClassification [8, 9] was submitted to the OGB leaderboard in 2021 by Wang et al., providing an implementation and a brief technical report describing the research. By using their method combined with Graphormer, they were able to achieve better average precision than all previous submissions.

The report introduces heterogeneous interpolation, which is done by dropping the feature vectors of several randomly selected nodes and replacing them by the interpolated feature mix of all neighboring nodes. By using a mixing ratio, the influence of each neighbors features can be adapted. To account for possible information loss, KL-Divergence constraint loss is added. By doing that, the distributions of two identical graphs remain similar after interpolating some feature vectors. The general idea was visualized by Wang et al. in Figure 3.

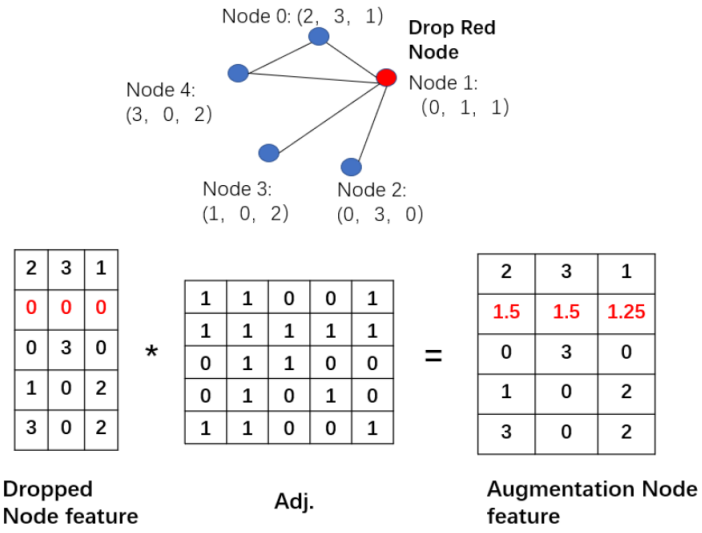

Figure 3: Visualization of HIG-GraphClassification by Wang et al., source: [9]

We combine the idea of interpolating feature vectors of neighboring nodes with GraphGPS to test performance with different embedding methods.

3 Our approach

In order to further strengthen our understanding, we decided to try to implement a graph machine learning method ourselves. As we knew we wouldn't be able to invent a game-changing method, we decided to try the simplest method we could imagine: A walk based embedding for whole graphs.

3.1 Walk based graph embeddings

One of the primary challenges in graph machine learning, as opposed to text-based machine learning, is the multidimensionality of graphs. Each node can have varying numbers of neighbors and different features and feature types. Walk based embeddings reduce this multidimensionality.

For our walk based embedding we focused on the embedding of whole graphs, as a similar method to embed nodes already exists (see [1]). The downstream task after the embedding of the graph is either regression or classification. For this we use sklearn.

3.1.1 walks to vector

A walk on graph is a unambiguously sequence of vertices and edges, where each vertex is connected to the next vertex by an edge. Given a graph g, we can write a walk as list of nodes (denoted by their id), e.g. $(0, 2, 3)$ is a walk that start from the node with the id 0, goes to 2 and end at 3. For a non-trivial graph there a many different possible walks, therefore we can create a set of walks w that all walk on the same graph. At this moment the walk doesn't contain any information relevant for our task, for this we need to substitute the node ids with the corresponding features. We substitute each id with the feature value prefixed with index of the feature separated by an otherwise unused character. For this we require that the features of a node are ordered. Therefore our schema for a feature in the walk is FeatureIndex_FeatureValue. We use this prefix in order to differentiate between same values of different feature types.

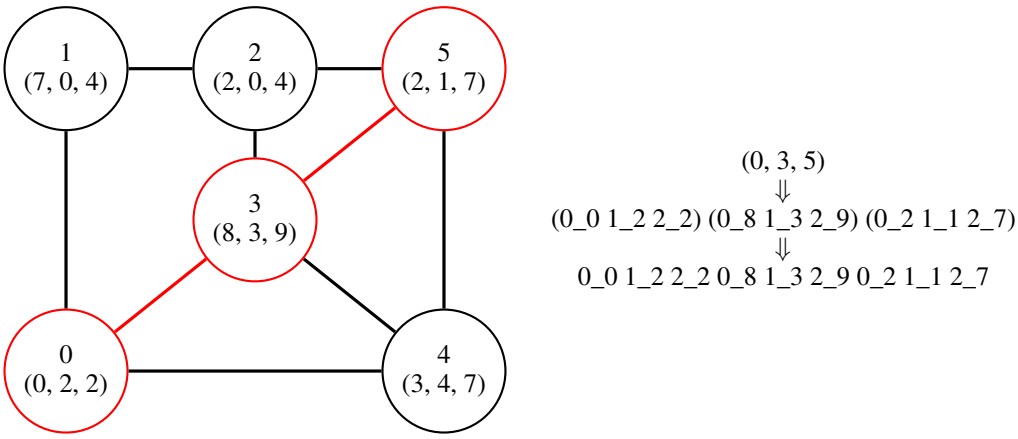

Figure 4: Example of graph on which a walk is done that is converted into a sentence

Lets talk about a concrete example. In the left side of figure 4 is an undirected graph with six vertices given. Each vertex has three numerical features. These features are given as list of integers below the node id. The red marked nodes form a walk from node 0 to node 5. On the right side of the figure, we listed the steps needed for each walk to generate the document needed for the downstream training: First we generate a walk, then we substitute the node id with the prefixed features and at last we concatenate the features into a sentence.

After this overview, we can formalize this method. Given a set of graphs $G = \{g_i \mid i \in 0 \dots n\}$, where n is the number of graphs, we can generate a set of sets of walks $W = \{ w_i \mid i \in 0 \dots n\}$. Now we create a set W' where for each walk we substitute the node id with the features of the node. If we view each walk as sentence we get a set of documents (multiple sentences). These documents are feed into a text embedding method.

Algorithm 1: basic idea of our walk based embedding

```
def get_vectors(graphs)
    documents = [ ]
    forall graph in graphs do
        walks = generate_walks(graph)
        document = walks_to_document(walks)
        documents.append(document)
    end
    model = Text_Embedding_Model()
    model.fit(documents)
    return model.get_document_vectors()
```

Pseudocode for our first implementation of our graph embedding method is given in algorithm 1. We identified two ways to influence the embedding in a meaningful way:

1. change the way walks are generated
2. change the text embedding model

3.1.2 change the way walks are generated

To accurately depict the neighborhood of a vertex, it is essential to represent the neighborhood within the generated document. This can be achieved when every combination of adjacent vertices is represented in the document multiple times.

We have considered two approaches for generating walks: All Pair Shortest Paths (APSP) and Random Walks. Each method possesses distinct advantages and drawbacks. APSP tends to have more hotspots of frequently visited graphs, which may introduce bias in the document; however, it generates similar walks for structurally analogous graphs. On the other hand, Random Walks can be set to a specific length and do not exhibit the same issue with frequently visited graphs. Nevertheless, they do not guarantee the generation of similar walks for structurally similar graphs.

One reason to consider using graph walks for analysis is that, given a sufficient number of walks over a graph, it becomes possible to reconstruct the original graph to some degree. These walks therefore provide insight into the graph's structure, capturing information about the relationships between vertices and the paths connecting them. By aggregating data from numerous walks, a comprehensive representation of the graph can be formed, enabling its reconstruction for further evaluation.

Reconstructing a graph from walks With an adequate quantity of walks over a graph, one can partially reconstruct the graph. However, there are limitations: For example, distinguishing between 'line' and 'circle' subgraphs is not always possible without any specific preparation of the graph. To illustrate this, refer to Figure 5. It is evident that each potential walkover one two graphs can be accommodated within the other graph as well.

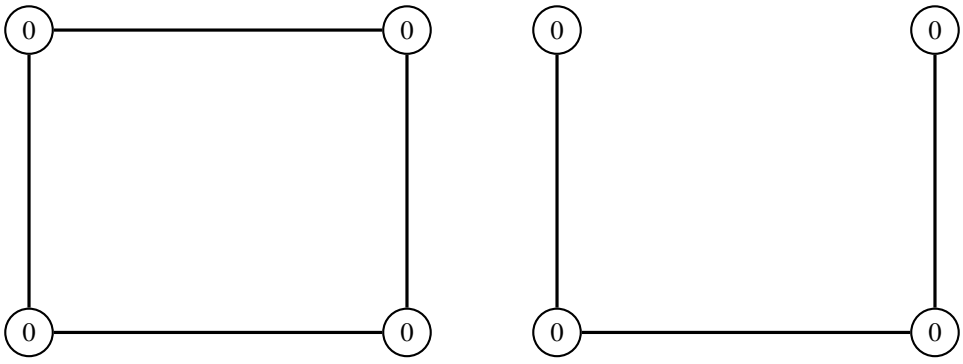

Figure 5: Example of two graphs whose walks can not be be differentiated

Similar, it os

Table 3: Node feature vector

Feature	Feature-Specialization	Test-AUC
Normal HiG		
Loss	0.5	0.77894
	0.1	0.77033
	0.01	
Nodes Interpolated	2	0.73967
	3	0.73882
	5	0.71498
min. graph size	5 nodes	
	10 nodes	0.76795
	20 nodes	0.77084
	50 nodes	0.7566

data

ogbg-molfreesolv all pair shortest path:6.528969941013406 [6.609125555671113, 6.57278283862703, 6.456680877236287, 6.539184198294062, 6.593562123437148, 6.417587074655182, 6.570877169599866, 6.478911030725958, 6.496761022461019, 6.5542275194263935]

3.1.3 change the text embedding model

3.2 GraphGPS with HIG

GraphGPS is by design a highly malleable model, which allows for easy additions to the code. Thus we implemented HiG in a section of GraphGPS.

3.2.1 Implementation

Listing 1: HiG-Code in GraphGPS

```
    if 'HIG' in pe_types:
        interpolation_chance = cfg.posenc_HIG.loss
        is_interpolating = np.random.choice(2, 1, p=[1-interpolation_chance, interpolatio
        minimum_node_size = cfg.posenc_HIG.minimum_node_size
        nodes_interpolated = cfg.posenc_HIG.nodes_interpolated
        minimum_node_size = max(minimum_node_size, nodes_interpolated)
        if data.num_nodes > minimum_node_size and is_interpolating == 1:
            rand_ints = np.random.choice(data.num_nodes, nodes_interpolated, replace=Fals
            sum = {}
            for rand_int in rand_ints:
                data.x[rand_int] = 0
                sum[rand_int] = 0
            for i in data.edge_index[0]:
                if i in rand_ints:
                    relevant_node = data.edge_index[1][i]
                    data.x[i] = torch.add(data.x[i], data.x[relevant_node])
                    sum[i] += 1
            for rand_int in rand_ints:
                data.x[rand_int] = data.x[rand_int] / sum[rand_int]
```

3.2.2 Results

References

[1] Aditya Grover and Jure Leskovec. *node2vec: Scalable Feature Learning for Networks*. 2016. DOI: 10.48550/arXiv.1607.00653.

[2] Weihua Hu et al. *OGB-LSC: A Large-Scale Challenge for Machine Learning on Graphs*. 2021. DOI: 10.48550/arXiv.2103.09430.

[3] Weihua Hu et al. *Open Graph Benchmark: Datasets for Machine Learning on Graphs*. 2021. DOI: 10.48550/arXiv.2005.00687.

[4] Quoc V. Le and Tomas Mikolov. *Distributed Representations of Sentences and Documents*. 2014. DOI: 10.48550/arXiv.1405.4053.

[5] Annamalai Narayanan et al. *graph2vec: Learning Distributed Representations of Graphs*. 2017. DOI: 10.48550/arXiv.1707.05005.

[6] Ladislav Rampášek et al. *Recipe for a General, Powerful, Scalable Graph Transformer*. 2023. DOI: 10.48550/arXiv.2205.12454.

[7] Ashish Vaswani et al. *Attention Is All You Need*. 2017. DOI: 10.48550/arXiv.1706.03762.

[8] Yan Wang et al. *HIG-GraphClassification*. https://github.com/TencentYoutuResearch/HIG-GraphClassification.git. 2021.

[9] Yan Wang et al. *Technical Report for OGB Graph Property Prediction*. Tech. rep. Shanghai, China: Tencent Youtu Lab, Dec. 2021. URL: https://github.com/TencentYoutuResearch/HIG-GraphClassification/blob/main/report/Report_HIG_OGB.pdf.

[10] Zhenqin Wu et al. *MoleculeNet: A Benchmark for Molecular Machine Learning*. 2018. DOI: 10.48550/arXiv.1703.00564.

[11] Chengxuan Ying et al. *Do Transformers Really Perform Bad for Graph Representation?* 2021. DOI: 10.48550/arXiv.2106.05234.