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

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.

1.1 Problem Domain

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,

Preprint. Under review.

where each vertex and edge has an attribute, e.g. $A(V \bigcup E) \rightarrow \Sigma^*$. While this basic definition does 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 [20] was introduced as a comprehensive benchmark for comparing ML models in molecular sciences, containing a multitude of tasks for molecular property prediction, among others including solubility, toxicity and bioactivity. Previously, there didn't exist a standardized evaluation platform for researchers to compare new machine learning algorithms and datasets were often small, as gathering molecular properties is an expensive task. MoleculeNet therefore aims to create a collection of datasets and evaluation tasks, enabling research teams to create meaningful and easily comparable performance measures for proposed ideas and algorithms. Furthermore MoleculeNet comes with a suite of tools to create and adapt ML models, containing implementation of many existing featurizers and algorithms for molecular machine learning.

To make those tools available at one source, Wu et al. the package DeepChem [1] was created.

2 Methodology

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) [7] 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 [6] 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 2 and Table 3. For ogb-molhiv the model is evaluated using ROC-AUC, for ogb-molpcba Average Precision is used.

name	graphs	tasks	task type	max fraction positive
ogbg-molhiv	41127	1	binary classification	0.035086
ogbg-molpcba	437929	128	binary classification	0.143402
ogbg-moltox21	7831	12	binary classification	0.120291
ogbg-molbace	1513	1	binary classification	0.456709
ogbg-molbbbp	2039	1	binary classification	0.765081
ogbg-molclintox	1477	2	binary classification	0.936357
ogbg-molmuv	93087	17	binary classification	0.000322
ogbg-molsider	1427	27	binary classification	0.923616
ogbg-moltoxcast	8576	617	binary classification	0.205340
ogbg-molesol	1128	1	regression	na
ogbg-molfreesolv	642	1	regression	na
ogbg-mollipo	4200	1	regression	na

Table 1: the molecule datasets from ogb

index	feature	feature type
0	atom number	multi-class
1	chirality	multi-class
2	degree	multi-class
3	formal charge	multi-class
4	number of H-atoms	multi-class
5	number of radicals	multi-class
6	hybridization	multi-class
7	is aromatic	binary-class
8	is in ring	binary-class

Table 2: Node feature vector

index	feature	feature type
0	bond type	multi-class
1	bond stereo	multi-class
2	is conjugated	binary-class

Table 3: Edge feature vector

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" [20]. The model is evaluated using Root Mean Square Error.

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.

2.1.1 Doc2vec

Doc2vec is a widely used method for obtaining distributed representations of documents as continuous vectors in a low-dimensional space, also known as paragraph embeddings. Doc2vec, introduced by Le and Mikolov [10], was built as an extension of the popular Word2vec algorithm [12] to the domain of entire documents and together with Word2vec builds the base for many ideas regarding graph embedding.

Doc2vec learns a vector representation for each document in a corpus by utilizing a so called Paragraph Vectors. The architecture is similar to that of Word2vec but includes an additional vector representation for each document. There are two ways to obtain document vectors: Distributed Memory version of Paragraph Vector (PV-DM) and Distributed Bag of Words version of Paragraph Vector (PV-DBOW).

In the PV-DM model, the document vector is trained to predict the next word in a randomly sampled sequence of words, with the document vector and a window of surrounding words as input. The document vector is then updated alongside the word vectors during backpropagation. This model captures the overall topic and meaning of the document, as it has access to all words in the document.

On the contrary, the PV-DBOW model doesn't pay attention to the order of words and only utilizes the occurrence information of words in the document. The input to this model is a bag of words from a randomly sampled window with the task to predict the words in the window. The document vector is trained alongside the word vectors, and aims to represent the document's overall semantic content.

In both models, the document vector is learned by optimizing a loss function, typically the negative log-likelihood of predicting the target words. The resulting document vectors are continuous, dense, and low-dimensional, making them suitable for various downstream tasks, such as document classification, clustering, and information retrieval.

Doc2vec has been shown to outperform other methods for document embeddings in several benchmark datasets and downstream tasks. This idea of capturing the semantics and meaning of the documents was the basis for many future implementations and also layed the path for good algorithms regarding graph embedding.

2.2 Graph Embedding

Graph embedding is a technique in graph machine learning that maps nodes and edges of a graph into a low-dimensional vector space while preserving the graph structure and semantic information. This technique has become increasingly popular due to its usefulness in various applications such as social network analysis, recommendation systems, and molecular sciences.

The goal for this technique is to produce a vector which can be used as input for various existing machine-learning models and techniques, elevating their usefulness to graphs. To achieve a good representation for those graphs, local and global structural information has to be encoded into such vectors. This was visualized very nice by Google Research [4] as seen in Figure 1. As seen here, overlapping communities are pulled apart while similar nodes are grouped together.

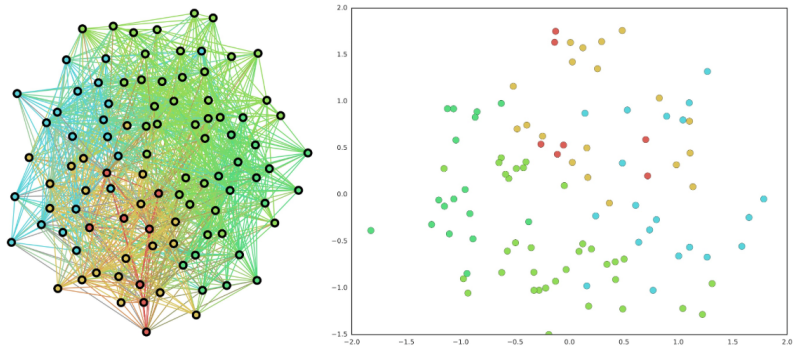

Figure 1: Embedding of a graph with overlapping communities using Node2vec source: [4]

There is a multitude of different ideas and methods to create graph embeddings, we will focus mainly on random walk embeddings and deep learning approaches, to be exact most methods are combinations of those ideas [13, 5, 21].

2.3 Node2vec

Node2vec, presented by Grover and Leskovec in 2016 [5], is a framework for learning low-dimensional representations of nodes in large-scale networks. The goal of Node2vec is to learn embeddings that capture the structural information of nodes, such as their connectivity patterns, in a way that is useful for downstream tasks, such as node classification, clustering, and link prediction, i.e. lowering the complexity of large network structures and encapsulating as much information as possible in a less complex structure, as seen in Figure 1.

Node2vec builds on the skip-gram model, which is popular in NLP problems and works by learning word embeddings through predicting the context of a given word. Correspondingly, Node2vec learns node embeddings by training a neural network to predict the context of a given node, where the context of a node is defined as the set of nodes that are likely to appear in random walks starting from that node.

Node2vec uses a biased random walk strategy, that balances the trade-off between exploring the whole, large network and staying in the local neighborhood of a node. Each walk starts at a given node choosing for each step to either revisit the previous node or move to the next one based on two parameters: the in-out degree bias p and the return bias q, where p controls the likelihood of staying in the same community, while q controls the likelihood of visiting a node that is further away from the current one. Node2vec can generate different types of random walks by changing these parameters, which are able to capture different types of structural information of the network.

After the random walks are generated, Node2vec uses the skip-gram model to learn node embeddings by maximizing the likelihood of predicting the correct context nodes given the current node. To train the skip-gram model, negative sampling is used, which samples negative nodes that are not in the context set and adjusts the embedding vectors to minimize the difference between the predicted and actual context sets.

Node2vec has several advantages over previous methods for network embedding. Although being scalable to large networks and being able to handle diverse networks with multiple types of nodes and edges, it can still capture both the local and global structure of the network by tuning the p and q parameters. Additionally, it can handle noisy or incomplete networks by incorporating information from the context nodes, alleviating missing data.

In conclusion, Node2vec is a performant and popular node embedding model that can be applied to a wide range of downstream tasks, such as link prediction, node classification, and community detection.

2.3.1 Graph2vec

Graph2vec is a method for learning distributed representations of graphs proposed by Narayanan et al. [13]. This approach builds on top of the previously described Doc2Vec method, but contrary to learning representations for words or paragraphs, it learns representations for whole graphs.

Graph2vec works by initially generating a set of subgraphs for each graph in the dataset, which can be seen as 'context graphs'. Those context graphs are then used to train a skip-gram neural network to learn embeddings for all graphs.

Context graphs are generated by performing a random walk on the whole graph. The next node to be selected during each step of the walk is chosen according to its distance to the currently selected one. The resulting sequence of nodes is then used to generate a set of subgraphs which are added to a pool of context graphs.

After that, once again the skip-gram model is used to learn embeddings for the graphs. Each graph is represented as a bag-of-context, i.e., a set of subgraphs that appear in the context of the graph and the skip-gram model then learns to predict the context graphs of a input graph, given the graph's embedding.

The skip-gram model used in Graph2vec is similar to the one used in word2vec, but instead of predicting the neighboring words of a given word, it predicts the context graphs of a given graph. The model takes as input a graph embedding and a context graph, and predicts whether the context graph appears in the bag-of-context-graphs of the input graph.

Graph2vec uses a negative sampling technique that is similar to the one used in Doc2vec to train the model whose objective it is to maximize the probability of predicting the correct context graphs of a given graph, while minimizing the probability of predicting randomly sampled negative context graphs.

The resulting embeddings capture the structural properties of the graphs, allowing them to be used for various downstream tasks. In our case, we will mainly use Graph2vec as a benchmark and reference point.

2.4 Transformers

In 2017 Vaswani et al. proposed a new network architecture for NLP problems, called the Transformer [15]. Abstractly speaking, Transformers consider input sentences as a connected system of words, thus attending on all other words while learning the meaning of one specifically.

The Transformer architecture is build of many layers, each containing a self-attention module and a position-wise feed-forward network. Self-attention is then calculated as described in [21]:

$$Q = HW_Q, \quad K = HW_K, \quad V = HW_V, \tag{1}$$

$$A = \frac{QK^\top}{\sqrt{d_K}}, \quad \mathrm{Attn}\,(H) = \mathrm{softmax}\,(A)\,V, \tag{2}$$

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{3}$$

Here Q, K, and V are calculated by multiplying the input H of the attention module with the respective matrices $W_Q \in \mathbb{R}^{d \times d_K}, W_K \in \mathbb{R}^{d \times d_K}$ and $W_V \in \mathbb{R}^{d \times d_V}$ which then represent Queries, Keys and Values of the attention module. Those matrices are learnable and are produced during the training of the model.

Queries are basically one-hot feature vectors, representing the features of interest, which are then multiplied with the transposed Key matrix to get feature specific masks (denoted as A above). Those masks are then used during Attention to isolate specific features from the values (denoted in $\mathrm{Attn}\,(H)$ above), increasing the correctness of value predictions.

d denotes the hidden dimensions, which in case of single-head self attention is the same as d_K and d_V. The hidden dimension is the dimension of the key vectors, in case of [15] this was 64.This leads to the equation for the attention matrix above and its adaption for Graphormer in Equation 4. The result of this equation is the softmax score, which is then multiplied with the value vectors. The sum of those weighted vectors is the output of the self-attention module. By doing this, other tokens of the input which are seen as relevant, will continue upwards while non relevant tokens are drowned with a small softmax score.

After each self-attention module follows a feedworard neural network, denoted in line (3) of Equation 3. During this step, ReLU is used to drown out negative values, x are masked word activities, $(W_1 + b_1)$ denotes a multi-word feature creation matrix and $(W_2 + b_2)$ is a transition matrix.

2.5 Graph Transformers

After the success of Transformers for NLP problems [8], many researchers tried adapting the ideas of Transformer for graph machine learning problems, taking advantage of the benefits of global attention in graph models, thus a multitude of graph transformer models [3, 21, 9, 11, 19] were published. Many of those ideas were leveraged in GraphGPS [14], as described in 2.5.3 and 3.3.

2.5.1 Graph Transformer Networks

GTNs, proposed by Dwivedi et al. [3], are designed to leverage graph sparsity and positional encodings and add them as a bias during self-attention. Attending to the whole graph is often not feasible because of graph size and node count, therefore GTNs only attend to the local neighborhood of nodes. Additionally, positional encodings are leveraged, to alleviate the missing structural information using Transformers, thus encoding important graph structural information into the Transformer. Positional encodings are created using Laplacian eigenvectors [2], which are added to the node features.

The key differences of GTNs with respect to original Transformers are the following. Neighborhood connectivity is used as the function for the attention mechanism, which is done by computing a weighted sum of the surrounding node embeddings. Positional encodings are replaced by Laplacian eigenvectors, which are precomputed for each graph and additionally, edge representations are added, to engrain important edge information into the model. Those edge features are learned by a separate feed-forward neural network, which takes the concatenated node embeddings as an input and the resulting edge features are again used in self-attention to update node embeddings.

2.5.2 Graphormer

As an adaption of Transformers [15] for graph representation learning, Graphormer [21] was presented by Ying et al., providing a performant Transformer model, excelling in various graph representation learning tasks, particularly for complex structures.

Graphormer achieves such good results by introducing three types of encodings as a bias to the self-attention mechanism used in Transformer architecture. While Transformer models work by

computing weighted sum of input features based on attention scores between different tokens during self-attention, Graphormer incorporates structural information of different nodes in pairwise relation to each other into the self-attention mechanism, thus increasing the information encapsuled in the self-attention module, which is held by graph structures and features present.

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 learnable embedding vectors, one for the degree of ingoing edges and one for the outgoing ones. Those vectors are scalars which are added to the features of each node, so node importance and semantic correlation are respected during self attention. Those scalars are indexed from two vectors according to the respective degrees as seen in Figure 2.

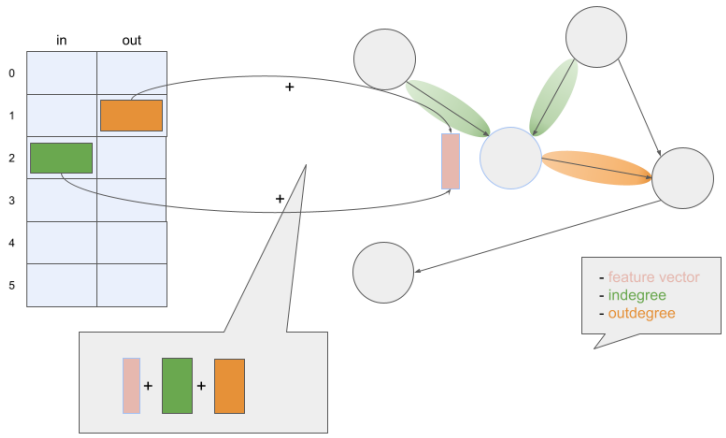

Figure 2: Centrality Encoding: learnable embedding vectors are added to feature vectors of nodes

The other two encodings are added as bias to the Attention Module used in Transformers as described in Figure 3 and the following equation.

$$A_{ij} = \frac{(h_i W_Q)(h_j W_K)^T}{\sqrt{d}} + b_{\Phi(v_i,v_j)} + c_{ij} \tag{4}$$

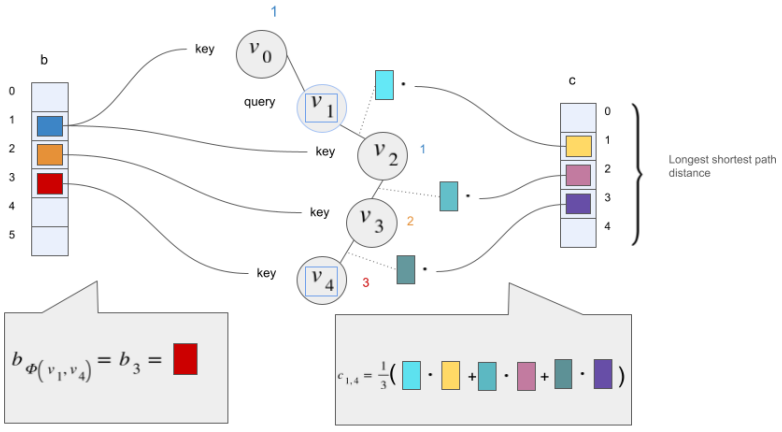

Figure 3: Spatial Encoding (left) and Edge Encoding (right) added as bias to Attention Module

The bias $b_{\Phi(v_i,v_j)}$ is produced by the so called spatial encoding, which captures positional information for each node with respect to the network. This is achieved by using the distance of the shortest path

between two connected nodes, described by the function Φ, combined with a learnable vector which is indexed according to Φ. The resulting scalar 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 the additional bias term c_{ij} to the attention module (Equation 5). This bias is created by averaging the dot-products of edge features (x_{e_n}) and a learnable scalar ($(w_n^E)^T$), which is indexed according to the distance of the edge to the start node for each edge along the shortest path between each two connected nodes.

$$c_{ij} = \frac{1}{N} \sum_{n=1}^{N} x_{e_n} (w_n^E)^T \tag{5}$$

2.5.3 GraphGPS

L. Rampásek et al. introduced GraphGPS in their paper 'Recipe for a General, Powerful, Scalable Graph Transformer.' [14] The objective of this model recipe is to provide a foundation for incorporating 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.

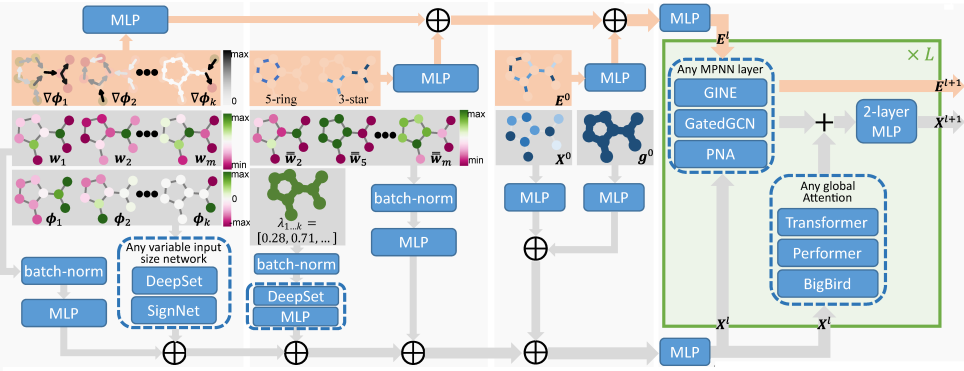

Figure 4: General overlook of GraphGPS taken from [14]

2.5.4 Heterogeneous Interpolation on Graphs

A winning entry for the dataset ogb-molpcba, called HIG-GraphClassification [16, 17] 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, even after interpolating some feature vectors. The general idea was visualized by Wang et al. in Figure 5.

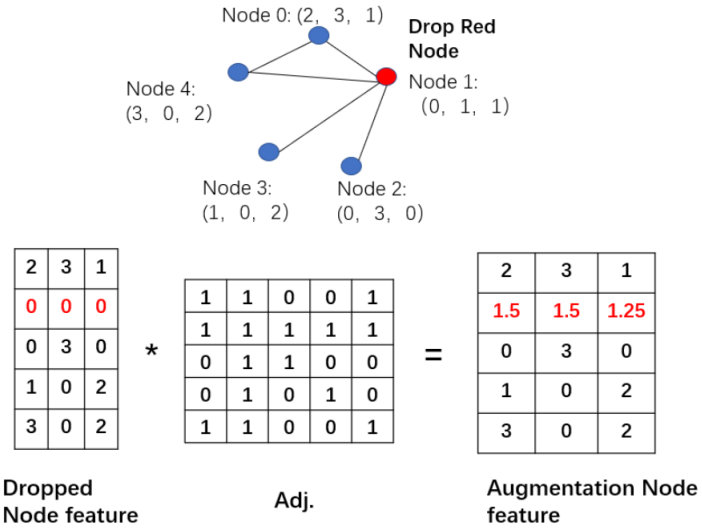

Figure 5: Visualization of HIG-GraphClassification by Wang et al., source: [17]

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

3 Our approach

To bolster our understanding, we decided to implement a graph machine learning method ourselves. Recognizing that it would be unlikely for us to devise a groundbreaking approach, we aimed to explore the simplest method we could imagine: a walk-based embedding.

For our walk-based embedding we focused on the embedding of whole graphs, as a similar method to embed nodes already exists (see [5]). 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.

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[18]. 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.

Our method can be described at a high level as follows: Given a graph, we generate a list of walks over the graph, in which the node IDs are substituted with the corresponding node features. This process results in a text document-like representation of the graph. Utilizing natural language processing techniques, we then transform the text document into a vector representation. These vector representations can subsequently be employed to train a model capable of predicting various graph characteristics.

3.1 Walk based graph embeddings

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.

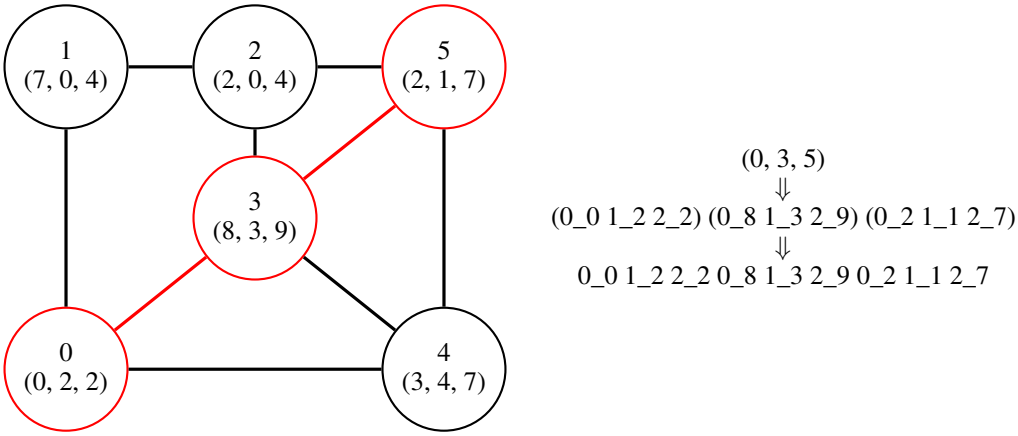

Figure 6: Example of graph on which a walk is done that is converted into a sentence

Lets talk about a concrete example. In the left side of figure 6 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.

Now 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 $W = \ w_i \mid i \in 0 \dots n$. Next, we create a new set W' in which we substitute the node ID with the node features for each walk. If we consider each walk as a sentence, we obtain a set of documents (comprising multiple sentences). These documents are then fed into a text embedding method. The pseudocode for our initial implementation of the graph embedding method is provided in algorithm 1.

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()
```

3.2 walk generation

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.

In our initial approach, we employed the all-pair shortest paths method; however, we quickly discovered that this was impractical for large datasets, as the sheer volume of text data overwhelmed the document embedding technique. Consequently, we opted for an alternative solution that limits the number of walks for each graph to a parameter referred to as sample_size in the code.

By performing the random selection of two nodes sample_size times, we obtain the shortest path between them and add it to the list of walks. Assuming the sample_size is sufficiently large, it is reasonable to expect that every node will be visited multiple times. This alternative approach offers a more manageable solution when working with extensive datasets.

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 7. It is evident that each potential walk over one two graphs can be accommodated within the other graph as well.

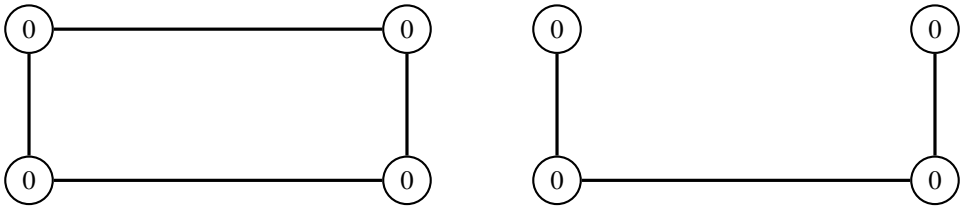

Figure 7: Example of two graphs for whom each walk done one graph could also be done on the other

In the case of regular graphs, where all vertices share the same degree, nodes with identical features often appear indistinguishable during walks. As a result, due to these inherent limitations, we did not expect walk-based whole graph embeddings to produce outstanding results for such graphs.

3.2.1 text embedding

Throughout the entirety of our project, we employed Doc2Vec[10] for transforming our walks into vector representations. Doc2Vec facilitates the training of a custom model using our specific vocabulary, allowing for the embedding of documents into vectors within this vocabulary. Despite our efforts to identify an alternative document-to-vector model that was user-friendly, we were unable to locate a suitable candidate.

While Doc2Vec is easy to use, we don't use it the way it is intended to be used. As natural language and our document 'language' certainly differ, we expect that there is a better to encode our documents. Doc2Vec generates word vectors by trying to predict the surroundings of the word (or the word by its surroundings). In natural language processing it makes sense to define the surroundings over the boundaries of sentences, but in our case, a new path/sentence has no semantic connection to the last one.

3.2.2 downstream training

For downstream training of the regressor or classifier, we followed scikit-learn's recommendations and chose Support Vector Regression (SVR) or Support Vector Classification (SVC) to train and generate the label prediction given the vector generated by out method. In the case of multi-output prediction, we employed scikit-learn's multi-output methods but sticked to the mentioned regressors and classifiers for the individual task. To replace missing values in the training data, we utilized scikit-learn's SimpleImputer.

3.2.3 results

Given the significant amount of data generated by random walks, which subsequently resulted in an extensive volume of training data for Doc2Vec, we focused on smaller datasets from the Open Graph Benchmark (OGB). For datasets with fewer than 10,000 graphs, we ran our model ten times;

for those with less than 100,000 graphs, we executed it three times; and for larger datasets, we carried out a single run. The outcomes are presented in Table 4.

For most datasets, our methodology did not yield significant outcomes for the majority of the datasets. To validate this observation, we generated random vectors with dimensions equivalent to those produced by our model and employed these vectors in identical downstream training processes. These random vectors serve as a benchmark for evaluating our model's performance. To assess the probability that the observed sample sets are derived from the same underlying distribution (i.e., both are random), we performed a Kolmogorov-Smirnov test on the final results of our model and the predictions based on random vectors.

dataset	paths2vec		random vectors		t-test	
	mean	std	mean	std	D	p-value
ogbg-molesol	1.4651	0.0309	2.2844	0.0239	1	1.08E-05
ogbg-molfreesolv	5.7668	0.0557	6.7620	0.0296	1	1.08E-05
ogbg-mollipo	TODO	TODO	TODO	TODO		

Table 4: comparison of the rmse for 10 runs on the dataset regression datasets

3.2.4 conclusion

We guess that the observed suboptimal performance may be attributed to the fact that our methods are only capable of capturing certain types of information from the graph. As previously discussed, walk-based embeddings encounter difficulties in accurately representing the structural aspects of a graph, such as size and subgraphs. Our interpretation of these results is, that our model is able to capture the composition of the graph, specifically the relationships between different types of nodes. However, due to our limited knowledge about the datasets, we are unable to judge whether this hypothesis is consistent with the tasks associated with these datasets.

Some of the classification datasets are also quite unbalanced, as can be seen in table 2.1, which make them harder to train on. Regression datasets in a way are easier to train on, as the they benefit even from a slight increase in information in the produced vectors.

3.2.5 outlook

We don't think that our approach by itself will generate good results for whole graph embeddings. Nevertheless, it might be interesting to search for a reason, why our methods produces meaningful results on some datasets but not on others. This might give insight into the dataset and the task to solve but also give insight into graph embeddings in general.

It might me also interesting to change out Doc2Vec for another, maybe faster method that offers the possibility to run the ogb large scale graph embedding dataset 'PCQM4Mv2'. Instead of concatenating shortest paths, it might also be interesting to use one, continuous, random walk. This would also solve the problem of the windows_size of Doc2Vec.

3.3 GraphGPS with HIG and IAOTR

The design of GraphGPS prioritizes flexibility, making it easy to modify and expand upon. This inspired us to integrate HIG-GraphClassification into GraphGPS, combining the concepts of two existing papers. Our focus was on incorporating the Positional Encodings (as shown in Figure 8) into the HIG model. However, during the implementation process, we unexpectedly designed what we call Illegal Arithmetic Operation Tensor Replacement (IAOTR). The nodes that would get replaced by an interpolation of feature vectors, get replaced by a very high negative number, e.g. -9223372036854775808.

3.3.1 Implementation

The implementation of the heterogeneous Interpolation, which involves dropping the feature vector (Table 2) of several randomly selected nodes and replacing them with an interpolated feature mix,

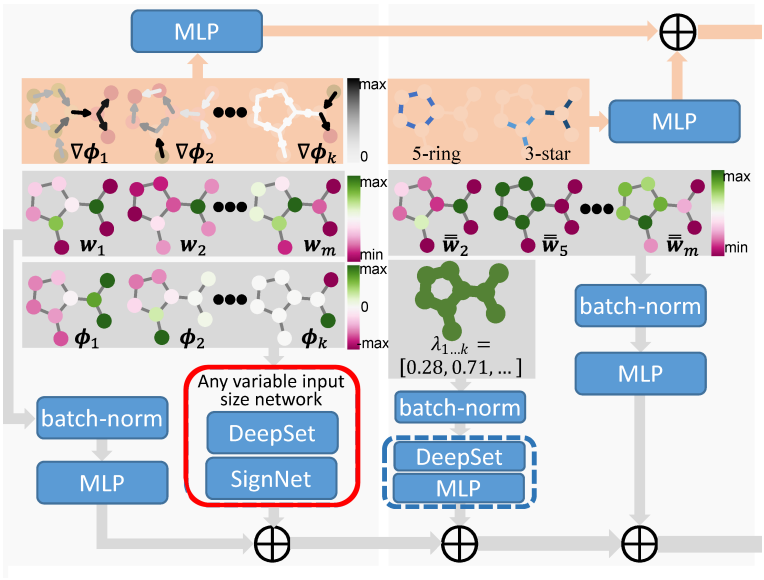

Figure 8: Position of HIG in GraphGPS structure

can be found in algorithm 2. While our implementation does not include the mixing ratio, we have added a Replacement-p parameter and a min_size parameter. The Replacement-p parameter enables us to specify the probability with which the graph replaces a node with an interpolated feature mix. On the other hand, the min_size parameter sets a minimum size for the graph. This is because a smaller graph tends to lose more information if a node is replaced, which could be disadvantageous. Additionally, we have included the nodes_replaced parameter, which allows us to specify the number of nodes that get replaced instead of randomly choosing how many get replaced.

Previously the if-condition in line 15 was never satisfied, which resulted in a division by 0 in line 27. The values of the Node feature vector are then replaced by very high negative values, e.g. -9223372036854775808. This still executed and lead to the results in Table 5.

Algorithm 2: HiG-Code in GraphGPS

```
if 'HiG' in pe_types then
    Replacement_chance = cfg.posenc_HIG.Replacement_p
    is_interpolating = np.random.choice(2, 1, p=[1-Replacement_chance,
     Replacement_chance])[0]
    minimum_node_size = cfg.posenc_HIG.minimum_node_size
    nodes_replaced = cfg.posenc_HIG.nodes_replaced
    minimum_node_size = max(minimum_node_size, nodes_replaced)
    if data.num_nodes > minimum_node_size and is_interpolating == 1 then
        rand_ints = np.random.choice(data.num_nodes, nodes_replaced
        sum = {}
        for rand_int in rand_ints do
            data.x[rand_int] = 0
            sum[rand_int] = 0
        end
        for i,j in zip(data.edge_index[0], data.edge_index[1]) do
            if i.item() in rand_ints then
                if j.item() in rand_ints then
                    continue
                end
                relevant_node = j; data.x[i] = torch.add(data.x[i], data.x[relevant_node])
                sum[i.item()] += 1
            end
        end
        for rand_int in rand_ints do
            if (sum[rand_int] ==0) then
                continue
            end
            data.x[rand_int] = data.x[rand_int] / sum[rand_int]
        end
    end
end
```

3.3.2 Results

In table 6 are the results of implementing HIG in GraphGPS. The lowest performance is when 5 nodes are replaced by an Replacement. For IAOTR the Parameter combination of 0.75 Replacement-p, 1 Node Replaced and 20 minimum graph Size performs best. For HIG this Parameter combination also performs best. HIG outperforms IAOTR, the more nodes are replaced. With the replacement-p going towards 0.5, IAOTR outperforms HIG.

Table 5: IAOTR on obgb-molhiv

Model Focus	Replacement-p	Nodes Replaced	min.graph size	Test-AUC
IAOTR	1.0	1	5	0.75274
Replacement-p	0.5	1	5	0.77894
	0.1	1	5	0.77033
	0.01	1	5	0.7625
Nodes Replaced	1.0	2	5	0.73967
	1.0	3	5	0.73882
	1.0	5	10	0.71498
min. graph size	1.0	1	10	0.76795
	1.0	1	20	0.77084
	1.0	1	50	0.7566
Combination	0.5	1	20	0.77164
	0.75	1	20	0.78321

Table 6: Our HIG-Implementation on obgb-molhiv

Model Focus	Replacement-p	Nodes Replaced	min.graph size	Test-AUC
No HIG				0.7710
HIG	1.0	1	5	0.77384
Replacement-p	0.5	1	5	0.76834
	0.1	1	5	0.76564
	0.01	1	5	0.76275
Nodes Replaced	1.0	2	5	0.74348
	1.0	3	5	0.73382
	1.0	5	10	0.72977
min. graph size	1.0	1	10	0.76878
	1.0	1	20	0.76636
	1.0	1	50	0.78039
Combination	0.75	1	20	0.7792

3.3.3 Conclusion

In conclusion, our experiment revealed that the IAOTR outperformed our HIG-Implementation, despite the unexpected success given that the code was not intended to function in this way. We believe that this is due to the IAOTR being more effective in preventing overfitting. On the other hand, when more nodes are replaced, HIG's performance surpasses that of the IAOTR, which may be due to the latter losing too much information at each node. Interestingly, our results also suggest that IAOTR on smaller molecules may eliminate crucial information, which is why HIG has a better performance in the standard setting.

Our explanation for why IAOTR works is because the high negative value gets replaced by 0 in a ReLU function that the standard GraphGPS [14] uses. Further research into IAOTR could shed more light on this phenomenon and offer insights into optimizing molecular representation learning.

Acknowledgments

The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

References

[1] Zhenqin Wu et al. *DeepChem*. `https://github.com/deepchem/deepchem`. 2023.

[2] Mikhail Belkin and Partha Niyogi. "Laplacian Eigenmaps for Dimensionality Reduction and Data Representation". In: *Neural Computation* 15.6 (2003), pp. 1373–1396. DOI: `10.1162/089976603321780317`.

[3] Vijay Prakash Dwivedi and Xavier Bresson. *A Generalization of Transformer Networks to Graphs*. 2021. DOI: `10.48550/arXiv.2012.09699`.

[4] Alessandro Epasto. *DeepChem*. `https://ai.googleblog.com/2019/06/innovations-in-graph-representation.html`. 2019.

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

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

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

[8] Katikapalli Subramanyam Kalyan, Ajit Rajasekharan, and Sivanesan Sangeetha. *AMMUS : A Survey of Transformer-based Pretrained Models in Natural Language Processing*. 2021. DOI: `10.48550/arXiv.2108.05542`.

[9] Devin Kreuzer et al. *Rethinking Graph Transformers with Spectral Attention*. 2021. DOI: `10.48550/arXiv.2106.03893`.

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

[11] Grégoire Mialon et al. *GraphiT: Encoding Graph Structure in Transformers*. 2021. DOI: 10.48550/arXiv.2106.05667.

[12] Tomas Mikolov et al. *Distributed Representations of Words and Phrases and their Compositionality*. 2013. DOI: 10.48550/arXiv.1310.4546.

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

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

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

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

[17] 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.

[18] Dominik Wittmann et al. "Reconstruction of graphs based on random walks". In: *Theoretical Computer Science* 410 (Sept. 2009), pp. 3826–3838. DOI: 10.1016/j.tcs.2009.05.026.

[19] Zhanghao Wu et al. *Representing Long-Range Context for Graph Neural Networks with Global Attention*. 2022. DOI: 10.48550/arXiv.2201.08821.

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

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