

SerialExperimentsOlga

Olga Yakobson

February 4, 2022
Version: My First Draft



Department of Mathematics,
Informatics and Statistics
Institute of Informatics



Artificial Intelligence and
Machine Learning

Bachelor's Thesis

SerialExperimentsOlga

Olga Yakobson

- | | |
|--------------------|---|
| <i>1. Reviewer</i> | Prof. Dr. Eyke Hüllermeier
Institute of Informatics
LMU Munich |
| <i>2. Reviewer</i> | John Doe
Institute of Informatics
LMU Munich |
| <i>Supervisors</i> | Jane Doe and John Smith |

February 4, 2022

Olga Yakobson

SerialExperimentsOlga

Bachelor's Thesis, February 4, 2022

Reviewers: Prof. Dr. Eyke Hüllermeier and John Doe

Supervisors: Jane Doe and John Smith

LMU Munich

Department of Mathematics, Informatics and Statistics

Institute of Informatics

Artificial Intelligence and Machine Learning (AIML)

Akademiestraße 7

80799 Munich

Abstract

Abstract (different language)

Acknowledgement

Contents

Introduction

The field of ML on graph-structured data has recently become an active topic of research. One reason for this is the wide range of domains and problems that are expressible in terms of graphs.

1.1 Motivation

1.2 Research Questions

1.3 Structure

?: ? Some text

?: ? Some text

?: ? Some text

?: ? Finally, the results of this thesis are summarized and a brief outline of promising directions for future research is given.

Before describing the problem, and later on the experimental setup, we first

1. Introduce three common prediction tasks in graph neural networks.
2. Give a general overview of how graph neural networks organize and process graph structured data.
3. We further discuss the relation of message passing mechanism to the WL-test, an algorithm for inspecting whether two graphs are isomorph.
4. Give a formal definition and description of two graph neural network architectures which will be used in our experiments.
5. Discuss typical issues which occur in graph neural networks and methods for addressing those issues.

2.1 Prediction Tasks and Typical Problems

Graphs naturally appear in numerous application domains, ranging from social analysis, bioinformatics to computer vision. A Graph $G = (V, E)$, where $V = \{v_1, \dots, v_n\}$ is a set of $N = |V|$ nodes and $E \subseteq V \times V$ a set of edges between those nodes. The unique capability of graphs enables capturing the structural relations among data, and thus allows to harvest more insights compared to analyzing data in isolation [Zha+19]. Graphs therefore can be seen as a general language for describing entities and relationships between those entities. Graph neural networks (GNNs) then organize graph structured data to tackle various prediction and classification tasks. Typically, one is interested in one of the following three tasks:

1. **Link prediction:** Predict whether there are missing links between two nodes e.g., knowledge graph completion.
2. **Vertex classification & regression:** Predict a property of a node e.g., categorize online users/items.
3. **Graph classification & regression:** Here, we are interested in classifying or predicting a continuous value for the entire graph, e.g., predicting a property of a molecule.

In this work the main focus will be on node classification, graph classification and graph regression for small- as well as medium-sized graphs.

2.2 Passing Messages in GNNs

Graphs, by nature, are unstructured. Vertices in graphs have no natural order and can contain any type of information. In order for machine learning algorithms to be able to make use of graph structured data, a mechanism is needed to organize them in a suitable way [Zho+20a; HYL17; Zha+19].

Message passing is a mechanism, which embeds into every node information about its neighbourhood [Xu+19; Zho+20a]. This can be done in several ways. One way of classifying a graph neural network is by looking at the underlying message passing mechanism. In this paper we will look at a network, where message passing is done via convolutions (graph convolutional network (GCN)). We will however occasionally use the more general term message passing, as the separation is rather blurred and message passing describes a neighborhood aggregation scheme which is seen as a generalisation of other, more specific mechanisms.

Formally, message passing in a graph neural network can be described as using two functions: AGGREGATE and COMBINE. The expressive and representational power of a graph neural network can then be determined by looking at the concrete functions and their properties, used to implement aggregation and combination. AGGREGATE mixes the hidden representation of every node's neighborhood in every iteration. COMBINE then combines the mixed representation together with the representation of the node. Each node uses the information from its neighbors to

update its embeddings, thus a natural extension is to use the information to increase the receptive field by performing AGGREGATE and COMBINE multiple times.

$$\begin{aligned} a_v^k &= \text{AGGREGATE}^{(k)}(\{h_u^{(k-1)} : u \in \mathcal{N}_{(v)}\}) \\ h_v^{(k)} &= \text{COMBINE}^{(k)}(h_v^{(k-1)}, a_v^{(k)}) \end{aligned}$$

For graph-level predictions, an additional READOUT- operation is used:

$$h_G = \text{READOUT}(\{h_v^{(K)} \mid v \in G\})$$

One useful type of information, which the message passing framework should be able to capture, is the local graph structure. This can be done by choosing functions with appropriate properties. A more detailed explanation will follow in ???. In spatial graph neural networks we make the assumption of the similarity of neighbor nodes. To exploit this spatial similarity, we perform composition by stacking multiple layers together increasing the receptive field.

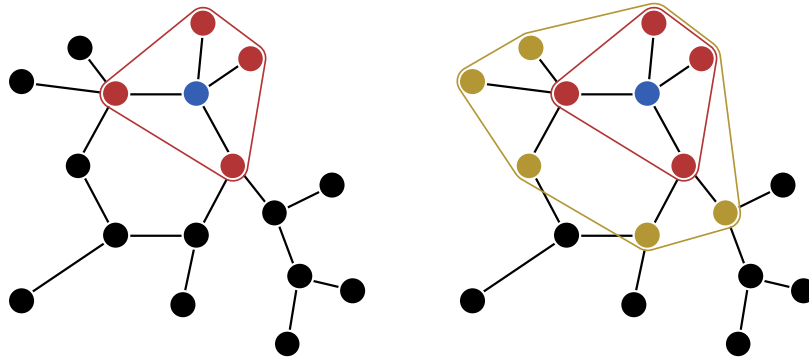


Fig. 2.1: By performing aggregation k -times, we can reach and capture the structural information of the k -hop neighborhood

2.2.1 Weisfeiler-Lehman Graph Colorings

The Message passing mechanism is closely related to the way the Weisfeiler-Lehman (WL) test [WL68; DMH20; HV22] works, an algorithm for deciding if two graphs are isomorphic. Before describing the algorithm, we introduce notations and prelim-

inaries.

Let $G = (V, E, X)$ denote an undirected graph, where $V = \{v_1, \dots, v_n\}$ is a set of $N = |V|$ nodes and $E \subseteq V \times V$ a set of edges between those nodes. For simplicity, we represent an edge $\{v, u\} \in E$ by $(v, u) \in E$ or $(u, v) \in E$. $X = [x_1, \dots, x_n]^T \in \mathbb{R}^{n \times d}$ is the node feature matrix, where $n = |V|$ is the number of nodes and $x_v \in \mathbb{R}^d$ represents the d -dimensional feature of node v . $\mathcal{N}_v = \{u \in V \mid (v, u) \in E\}$ is the set of neighboring nodes of node v . A multiset is denoted as $\{\!\!\{\dots\}\!\!\}$ and formally defined as follows.

Definition 2.1 (Multiset). A multiset is a generalization of a set allowing repeating elements. A multiset \mathcal{X} can be formally represented by a 2-tuple as $X = (S_X, m_X)$, where S_X is the underlying set formed by the distinct elements in the multiset and $m_X : S_X \rightarrow \mathbb{Z}^+$ gives the multiplicity (i.e., the number of occurrences) of the elements. If the elements in the multiset are generally drawn from a set X (i.e., $S_X \subseteq X$), then X is the universe of X and we denote it as $X \subseteq \mathcal{X}$ for ease of notation.

Definition 2.2 (Isomorphism). Two Graphs $\mathcal{G} = (V, E, X)$ and $\mathcal{H} = (P, F, Y)$ are *isomorphic*, denoted as $\mathcal{G} \simeq \mathcal{H}$, if there exists a *bijective* mapping $g : V \rightarrow P$ such that $x_v = y_{g(v)}$, $\forall v \in V$ and $(v, u) \in E$ iff $(g(v), g(u)) \in F$.

The 1-dimensional WL algorithm (color refinement)

In the 1-dimensional Weisfeiler-Lehman algorithm (1-WL), a label, called *color*, c_v^0 is assigned to each vertex of a graph. Then, in every iteration, the colors get updated based on the multiset representation of the neighborhood of the node until convergence. If at some iteration the colorings of the graphs differ, 1-Weisfeiler-Lehman decides that the graphs are not isomorphic.

$$c_v^l \leftarrow \text{HASH} (c_v^{l-1}, \{\!\!\{c_u^{l-1} \mid u \in \mathcal{N}_v\}\!\!\})$$

Algorithmically, this can be expressed as follows:

Algorithm 1 1-dim. Weisfeiler-Lehman (color refinement)

Input: $G = (V, E, X_V)$

- 1: $c_v^0 \leftarrow \text{hash}(X_v)$ for all $v \in V$
 - 2: **repeat**
 - 3: $c_v^l \leftarrow \text{hash}(c_v^{l-1}, \{\{c_w^{l-1} : w \in \mathcal{N}_G(v)\}\})$ forall $v \in V$
 - 4: **until** $(c_v^l)_{v \in V} = (c_v^{l-1})_{v \in V}$
 - 5: **return** $\{c_v^l : v \in V\}$
-

The 1-Weisfeiler-Lehman is a heuristic method which can efficiently distinguish a broad class of non-isomorphic graphs [BK79]. However, there exist some corner cases, where the algorithm fails to classify simple shapes as non-isomorphic. This is the case for non-isomorphic graphs with the same number of nodes and equivalent sets of node-degrees, as shown in ??.

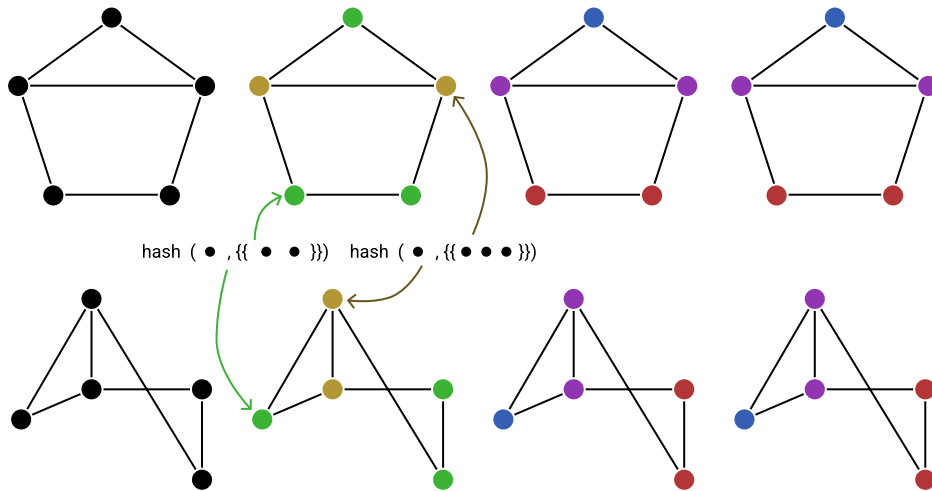


Fig. 2.2: Two isomorphic graphs. 1-Weisfeiler-Lehman assigns the same representation to those graphs.

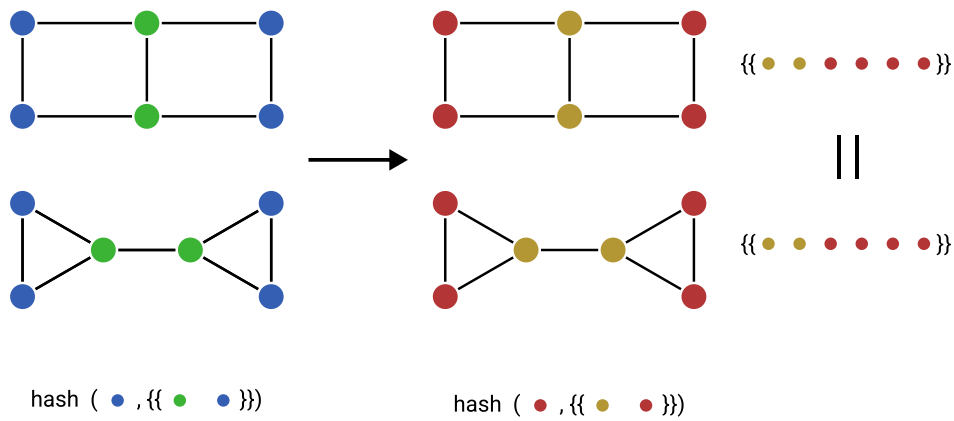


Fig. 2.3: 1-Weisfeiler-Lehman assigned the same labeling to two non-isomorphic graphs [LYJ22].

2.2.2 GNN Architectures in this Paper

In the following section we briefly introduce and motivate the choice of two types of networks, which we have chosen to experimentally verify the efficacy of several regularization techniques, which will be discussed in ??.

Since all graph neural networks incorporate message passing in a way, we decided to chose two architectures for our experiments, which are powerful, efficient, scalable and broadly used.

Graph Convolutional Network (GCN)

Graph convolutional network (GCN) was originally proposed by Kipf and Welling [KW17] to tackle the problem of semi-supervised node classification, where lables are available for a small subset of nodes. graph convolutional network is a simple, but powerful architecture, that scales linearly in the number of graph edges and learns hidden layer representations that encode both local graph structure and features of nodes.

A graph convolutional network can formally be expressed via the following layer-wise propagation rule:

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l)})$$

Where $\tilde{A} = A + I_N$ is the adjacency matrix of the undirected graph \mathcal{G} with added self-connections. I_N is the identity matrix. $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ and W^l is a layer-specific trainable weight-matrix. $\sigma(\cdot)$ denotes an activation function, such as $\text{ReLU}(\cdot) = \max(0, \cdot)$. $H^l \in N \times D$ is the matrix of activations in the l^{th} layer; $H^0 = X$.

Because we consider every neighbor to be of equal importance and therefore normalization is accomplished by dividing by the number of neighbours, one can view this operation as performing an element-wise mean-pooling [Xu+19].

$$h_v^{(k)} = \text{ReLU}(W \cdot \text{MEAN}\{h_u^{k-1} \mid \forall u \in \mathcal{N}_{(v)} \cup \{v\}\})$$

An application of a two-layer graph convolutional network is given by:

$$Z = f(X, A) = \text{softmax}(\hat{A} \text{ReLU}(\hat{A} X W^0) W^l)$$

where $\hat{A} = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$ is calculated in a preprocessing step. The model uses a single weight matrix per layer and deals with varying node degrees through an appropriate normalization of the adjacency matrix. This model consisting of a 2-layer graph convolutional network performed well in a series of experimental tasks, including semi-supervised document classification, semi-supervised node classification in citation networks and semi-supervised entity classification in a bipartite graph extracted from a knowledge graph. The prediction accuracy was evaluated on a set of 1000 examples and additional experiments on deeper models with up to 10 layers have been also provided. Being capable of encoding both graph structure and node features, graph convolutional network outperformed numerous related methods by a significant margin [KW17].

Graph convolutional networks (GCNs) are widely and successfully used today in many fields due to their simplicity and scalability.

Graph Isomorphism Network (GIN)

To overcome the lack of expressivity of popular GNN architectures, Xu et al. [Xu+19] designed a new type of graph neural network, graph isomorphism network. They prove that graph isomorphism networks are strictly more expressive than a variety of previous graph neural network architectures and that they are in fact as powerful as the commonly used 1-dimensional Weisfeiler-Lehman (WL)-test.

Two requirements must be met for a network to have the same expressive and representational power as the Weisfeiler-Lehman isomorphism test:

1. The framework must be able to represent the set of feature vectors of a given nodes neighbors as a multiset.
2. Choosing an injective function for the aggregation step. Such a function would never map two different neighborhoods to the same representation.

The more discriminative the multiset function is, the more powerful the representational power of the underlying graph neural network.

Formally, a graph isomorphism network (GIN) can be expressed as follows:

$$h_v^{(k)} = \text{MLP}^{(k)} \left((1 + \epsilon^{(k)}) \cdot h_v^{(k-1)} + \sum_{u \in \mathcal{N}(v)} h_u^{(k-1)} \right)$$

The choice of such an architecture, is motivated by the necessity to learn two functions with certain properties, f and ϕ . This task can be accomplished using a multilayer perceptron. The following lemma and corollary, proven by Xu et al. [Xu+19] show the properties and application of the functions:

Theorem 2.3. *Let $A : G \rightarrow \mathbb{R}^d$ be a graph neural network. With a sufficient number of graph neural network layers, A maps any graphs G_1 and G_2 to different embeddings, the Weisfeiler-Lehman-test of isomorphism decides as non-isomorphic, to different embeddings if the following conditions hold:*

(a) A aggregates and updates node features iteratively with

$$h_v^{(k)} = \phi(h_v^{(k-1)}, f(\{h_u^{(k-1)} \mid u \in \mathcal{N}_{(v)}\}))$$

where the functions f , which operates on multisets and, and ϕ are injective.

(b) A 's graph-level readout, which operates on the multiset of node features $\{h_v^{(k)}\}$, is injective.

Lemma 2.4. Assume \mathcal{X} is countable. There exists a function $f : \mathcal{X} \rightarrow \mathbb{R}^n$ so that $h(X) = \sum_{x \in X} f(x)$ is unique for each multiset $X \subseteq \mathcal{X}$ of bounded size. Moreover, any multiset function g can be decomposed as $g(X) = \phi(\sum_{x \in X} f(x))$ for some function ϕ .

Corollary 2.5. Assume \mathcal{X} is countable. There exists a function $f : \mathcal{X} \rightarrow \mathbb{R}^n$ so that for infinitely many choices of ϵ , including all irrational numbers, $h(c, X) = (1 + \epsilon) \cdot f(c) + \sum_{x \in X} f(x)$ is unique for each pair (c, X) , where $c \in \mathcal{X}$ and $X \subseteq \mathcal{X}$ is a multiset of bounded size. Moreover, any function g over such pairs can be decomposed as $g(c, X) = \varphi(1 + \epsilon)f(c) + \sum_{x \in X} f(x)$ for some function φ .

graph isomorphism network is as powerful as 1-dimensional Weisfeiler-Lehman

Graph isomorphism network is a neural network-based approach designed to handle graph data and detect graph isomorphisms. It operates on each vertex and updates its representation based on its own features and the aggregated features of its neighbors. There exists a fundamental similarity between the graph isomorphism network and the way the 1-Weisfeiler-Lehman algorithm works

2.2.3 Weaknesses and Obstacles in graph neural networks

Because of the way graph neural networks operate, they tend to suffer from two main obstacles: Overfitting and oversmoothing.

Overfitting hinders the generalization ability of a neural network (NN), making it perform poorly on previously unseen data. This occurs especially when using small datasets, since the model tends to 'memorize' instead of learn the pattern.

Oversmoothing is a condition, where the performance and predictive power of a neural network does not improve or even gets worse when more layers are added. This happens because by stacking multiple layers together aggregation is being performed over and over again. This way, the representation of a node is being smoothed, i.e., mixed with features of very distant, possibly unrelated nodes. Oversmoothing is a problem mainly for node classification tasks. There is a trade-off between the expressiveness of the model (capturing graph structure by applying multiple layers) and oversmoothing, which leads to a model where nodes have the same representation, because they all converge to indistinguishable vectors [Zho+20b; Has+20].¹

A closer examination of underlying causes of oversmoothing was conducted by Chen et al. [Che+20], who suggested, that not message passing itself, but the type of interacting nodes cause this issue. For node classification (NC) tasks, intra-class communication (interaction between two nodes sharing the same class) is useful (signal), whereas inter-class communication (the communication between two nodes sharing different labels) is considered harmful, because it brings interference noise into the feature-representations by mixing unrelated features and therefore making unrelated nodes more similar to each other. Because of that, the quality of shared information is essential and should therefore be considered as a benchmark for improvement.

2.2.4 Regularization Techniques

Kukacka et al. [KGC17] define regularization as any supplementary technique that aims at making the model generalize better, i.e., produce better results on the test set, which can include various properties of the loss function, the loss optimization algorithm, or other techniques.

One subgroup of regularization is via data, where the training set \mathcal{D} is transformed into a new set \mathcal{D}_R using some stochastic parameter π , which can be used in various

¹In spatial graph neural networks we make the assumption of relatedness by proximity.

ways, including to manipulate the feature space, create a new, augmented dataset or to change e.g, thin out the hidden layers of the neural network.

An example of such a transformation is corruption of inputs by Gaussian noise.

$$\tau_0(x) = x + \theta, \theta \sim \mathcal{N}(0, \Sigma)$$

In this work we focus on stochastic regularization techniques, which perform data augmentation in one way or another and whose main benefits lie in the alleviation of overfitting and oversmoothing [Has+20]. We will use the following notation:

Notation	Description
$H^{(l)} = [h_0^{(l)}, \dots, h_n^{(l)}]^T \in \mathbb{R}^{n \times f_l}$	Output of the l -th hidden layer in graph neural network
n	Number of nodes
f_l	The number of output features at the l -th layer
$H^0 = X \in \mathbb{R}^{n \times f^0}$	Input matrix of node attributes
f_0	Number of nodes features
$W^l \in \mathbb{R}^{f_l \times f_{l+1}}$	The graph neural network parameters at the l -th layer
$\sigma(\cdot)$	Corresponding activation function
$\mathcal{N}(v)$	Neighborhood of node v
$\tilde{\mathcal{N}}(v) = \mathcal{N}(v) \cup v$	$\mathcal{N}(v)$ with added self-connection
$\mathfrak{N}(\cdot)$	Normalizing operator
\odot	Hadamard product

DropOut (Srivastava et al.)

DropOut[Sri+14] randomly removes elements of its previous hidden layer $H^{(l)}$ based on independent Bernoulli random draws with a constant success rate at each training iteration:

$$H^{(l+1)} = \sigma(\mathfrak{N}(A)(Z^{(l)} \odot H^{(l)})W^{(l)})$$

where Z^l is a random binary matrix, with the same dimensions as H^l , whose elements are samples of Bernoulli(π).

The random drop of units (along with their connections) from the neural network during training prevents units from co-adapting too much. A neural net with n units can be seen as a collection of 2^n possible networks. Applying dropout with a certain probability π can be interpreted as sampling “thinned” networks from all possible 2^n networks. In the end, since averaging over all possible networks is computationally expensive, an approximation for combining the prediction is used. This averaging method entails using a single neural net with weights, which are scaled-down weights obtained during training time.

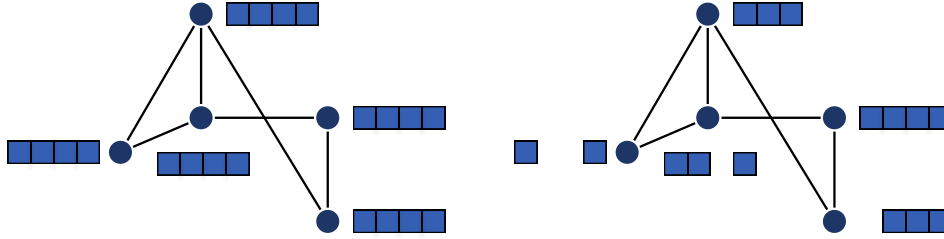


Fig. 2.4: DropOut (DO) preserves connections between nodes as well as the nodes itself, unless we chose a large probability π , which drops all of the nodes features.

DropEdge (Rong et al.)

DropEdge [Ron+20] randomly removes a certain number of edges from the input graph at each training epoch and can be formally expressed as follows:

$$H^{(l+1)} = \sigma(\Re(A \odot Z^{(l)})H^{(l)}W^{(l)})$$

The random binary mask Z^l has the same dimensions as A . Its elements are the random samples of $\text{Bernoulli}(\pi)$ where their corresponding elements in A are non-zero and zero everywhere else.

Message passing in graph neural networks happens along the edges between neighbours. Randomly removing edges makes the connections more sparse, which leads to slower convergence time and thus prevents the network from oversmoothing and allows for a deeper architecture. Intuitively this makes sense, since removing an edge means, that the node, previously connected by that edge stops being a neighbor. Consequently the representation of this former neighbor does not get mixed with the representation of the node.

DropEdge also acts like a data augementer, since by randomly dropping edges we manipulate/change the underlying graph data. Since the data is now augmented

with noise, it is harder for the network to overfit the data by “memorising” rather than learning complex relationships. The combination of DropOut and DropEdge reaches a better performance in terms of mitigating overfitting in graph neural networks than DropEdge on it’s own.

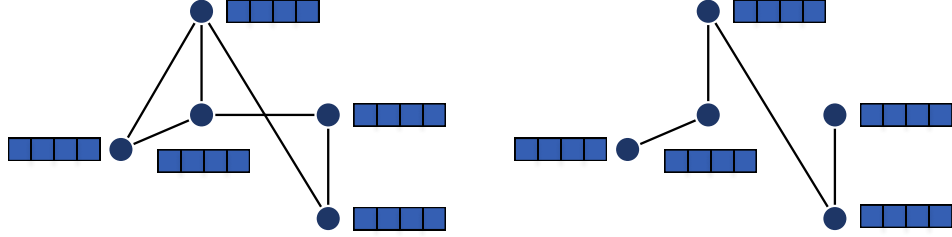


Fig. 2.5: DropEdge (DE) preserves nodes and all of nodes features, but randomly removes edges, leading to a smaller number of neighbors, which results in slower convergence times and allows for architectures with more hidden layers.

NodeSampling (Chen et al.)

This method of regularization, also known as FastGCN [CMX18] was developed to improve the graph convolutional network [KW17] architecture and to adress the bottleneck issues of graph convolutional networks caused by recursive expansion of neighborhoods. It reduces the expensive computation in batch training of graph neural network by relaxing the requirement of simultaneous availability of test data. Graphs can be very large and therefore require large computational and processing capacities. By randomly dropping out nodes, we reduce the amount of data in such a manner, that it alleviates the expensiveness of the computation reduces and bottleneck issue while preserving important relations.

$$H^{(l+1)} = \sigma(\Re(A)diag(z^{(l)})H^{(l)}W^{(l)})$$

Here, $z^{(l)}$ is a random vector whose elements are drawn from $Bernoulli(\pi)$. This is a special case of DropOut, since all of the output features are either kept or completely dropped.

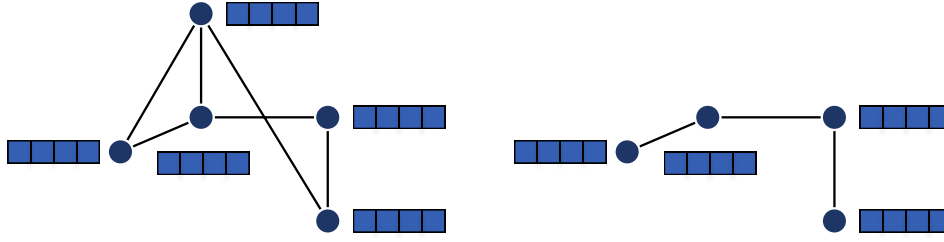


Fig. 2.6: In NodeSampling (NS), a node is either removed or preserved along with the whole feature vector with a certain probability π .

GraphDropConnect (Hasanzadeh et al.)

Finally, GraphDropConnect [Has+20], which can be seen as a generalization of all the above proposed methods, is a stochastic regularization approach, which has been shown to be the most effective among all the above and even more effective than the combination of DropOut and DropEdge. The regularization is done via adaptive connection sampling and can be interpreted as an approximation of Bayesian graph neural networks.

$$H^{(l+1)}[:, j] = \sigma \left(\sum_{i=1}^{f_l} \Re \left(A \odot Z_{i,j}^{(l)} \right) H^{(l)}[:, i] W^{(l)}[i, j] \right)$$

for $j = 1, \dots, f_{l+1}$

where f_l and f_{l+1} are the number of features at layers l and $l+1$, respectively, and $Z_{i,j}^{(l)}$ is a sparse random matrix (with the same sparsity as A), whose non-zero elements are randomly drawn from Bernoulli(π_l), where π_l can be different for each layer. GraphDropConnect is a regularization technique, that combines all of the above by drawing different random masks for each channel and edge independently, which yield better performance results than all of the previous methods or even combinations of them. GraphDropConnect, as it is expressed in the formula above has not been implemented and evaluated yet. Instead, a special case of GraphDropConnect has been implemented:

Under the assumption, that $Z_{i,j}^{(l)}$ are the same for all $j \in \{1, 2, \dots, f_{l+1}\}$, we can omit the indices of the output elements at layer $l + 1$ and rewrite the above formula as follows:

$$H^{(l+1)} = \sigma\left(\sum_{i=1}^{f_l} \mathfrak{N}(A \odot Z_i^{(l)}) H^{(l)}[:, i] W^{(l)}[i, :]\right)$$

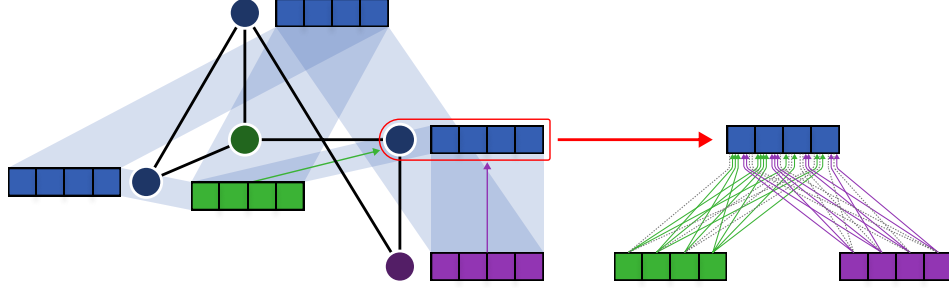


Fig. 2.7: GraphDropConnect (GDC) can be thought of as duplicating every existing edge between features of the feature-vectors of existing nodes and then randomly removing every edge with a certain probability π before the convolution.

All the methods are somewhat related and share some similarities [Ron+20]. DropOut (DO) has been successful in alleviating overfitting by perturbing the feature matrix and setting some entries to zero. The issue of oversmoothing is not affected by this measure. DropEdge (DE) achieved great results in reducing both overfitting as well as oversmoothing. Intuitively this makes sense, because smoothing comes from the aggregation of the neighbours of a certain node and by dropping the connections to some neighbours, the feature vectors of those neighbours are no longer aggregated and combined with the hidden representation of the node.

NodeSampling (NS) is a special case of DropOut (DO), as all of the output features for a node are either completely kept or dropped while DropOut randomly removes some of these related output elements associated with the node. Also, along with the dropped node, the edges of this node are dropped. The method itself, however is node-oriented and the edge-drop is a "side-effect".

GraphDropConnect (GDC) generalizes existing stochastic regularization methods for training graph neural networks and is effective in dealing with overfitting and oversmoothing. GraphDropConnect regularizes neighbourhood aggregation in graph neural networks at each channel separately. This prevents connected nodes in graph

from having the same learned representations in graph neural network layers; hence better improvement without serious oversmoothing can be achieved [Has+20].

Implementation

This section provides a brief overview of the experimental setup and aims to motivate the choice of datasets, libraries and frameworks.

3.1 Scope and Limitations

3.2 Experimental Setup

GraphDropConnect

Before we describe our implementation in Python, we look at the two proposed variants of GraphDropConnect and provide an intuition for them. As stated previously, this version of GraphDropConnect allows drawing different random masks for each channel and edge independently, giving more flexibility and increasing the time and space complexity.

$$H^{(l+1)}[:, j] = \sigma \left(\sum_{i=1}^{f_l} \Re \left(A \odot Z_{i,j}^{(l)} \right) H^{(l)}[:, i] W^{(l)}[i, j] \right)$$

for $j = 1, \dots, f_{l+1}$

Here, we calculate the new feature matrix $H^{(l+1)}$ by stacking the column vectors of each iteration. One can think of the calculations that are being performed as a 4-dimensional matrix with the dimensions $n \times n \times f_l \times f_{l+1}$

To understand what is being done, we can look at what is being performed when calculating one column of the resulting matrix. First, a random mask is applied to the connection from the i -th to the j -th. Regarding node features communicating with each other, we look at the edge between the i -th and j -th features between connected nodes. By applying the random mask, we drop those connections selectively. Because

we sample the random binary mask Z f_{l+1} times, one time for every feature in the $l + 1$ -th iteration, we differentiate between the connection $i \rightarrow j$ between two nodes and the connection $j \rightarrow i$ between the same two nodes. Thus, the same edge can be dropped as a connection and remain as a connection in the opposite direction. The masked adjacency is then multiplied by the corresponding column and weight. One may think of it as performing a random sampling across each channel since in each iteration from $i = 1$ to f_{l+1} , we perform multiplication with the i -th column of H .

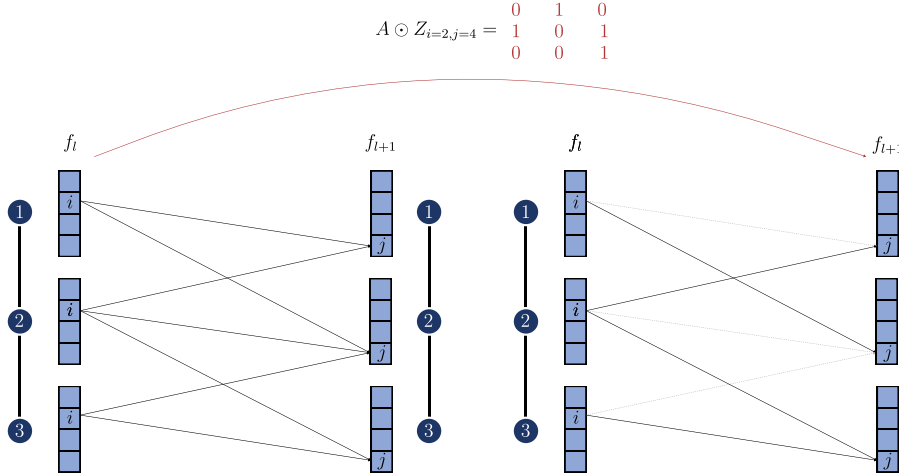


Fig. 3.1: GraphDropConnectNote: self connection are assumed

As for the implementation of GraphDropConnect, we decided to implement the less complex version, as shown below, since this implementation reduces the runtime completely and is also the one that was originally implemented for testing the efficacy of GraphDropConnect

$$H^{(l+1)} = \sigma\left(\sum_{i=1}^{f_l} \Re(A \odot Z_i^{(l)}) H^{(l)}[:, i] W^{(l)}[i, :]\right) \quad (3.1)$$

Here, we compute the new feature matrix in one go, instead of doing f_l iterations for all f_l columns.

3.2.1 Choice of Frameworks

Below we give a quick overview of used datasets and frameworks and motivate the choice.

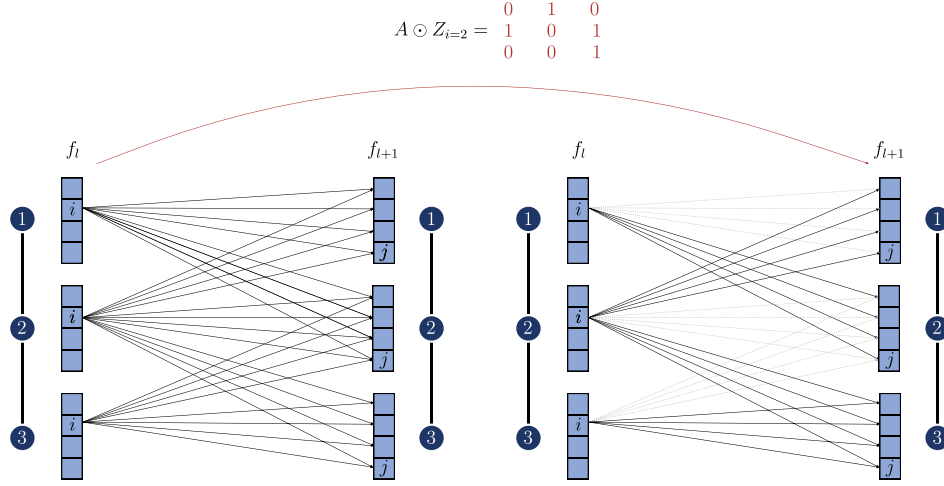


Fig. 3.2: Originally proposed GraphDropConnectNote: self connection are assumed

Datasets

Despite the fact, that machine learning on graph-structured data is carried out in many areas and has many interesting use cases ranging from social networks, to molecular graphs, to manifolds, to source code [Hu+20], there does not exist a unified framework for working with graph-structured data. Furthermore commonly-used datasets and evaluation procedures suffer from multiple issues, that negatively affect the quality of predictions and the reliability of evaluations of models. Machine learning algorithms rely heavily on data. In order for a graph neural network to be able to make accurate predictions, there is need for a sufficient amount of properly prepared training data. To be able to compare different models against each other there is need of standardized splitting and evaluation methods.

Today, most of the frequently-used graph datasets are extremely small compared to graphs found in real applications. Same datasets, such as Cora, CiteSeer and PubMed are used again and again to train various models leading to poor scalability in the majority of cases. Small datasets also make it hard to rigorously evaluate data-hungry models, such as graph neural networks (GNNs). The performance of a graph neural network on these datasets is often unstable and nearly statistically identical to each other, due to the small number of samples the models are trained and evaluated on [KW17; Xu+19; Hu+20].

Open Graph Benchmark offers a wide range of different-sized graph-datasets from different domains for a variety of different classification tasks and provides an unified pipeline for working with the datasets in machine learning tasks. The

unified experimental protocol with standardized dataset splits, evaluation metrics and cross-validation protocols makes it easy to compare performance reported across various studies [Hu+20].

Working with Open Graph Benchmark consists of following steps:

1. Open Graph Benchmark provides realistic, different-scaled graph benchmark datasets that cover different prediction tasks, are from diverse application.
2. Dataset processing and splitting is fully automated. Open Graph Benchmark data loaders automatically download and process graphs and further split the datasets in a standardized manner. This is compatible with multiple libraries and a library-agnostic option is also provided.
3. This step includes developing an machine learning model to train on the ogb datasets.
4. Open Graph Benchmark evaluates the model in a dataset-dependent manner, and outputs the model performance appropriate for the task at hand.
5. Open Graph Benchmark provides public leaderboards to keep track of recent advances.



Fig. 3.3: Overview of the standardized OGB pipeline adapted from [Hu+20]

Metrics

To be able to make systematic and quantative statements about the positive effects on oversmoothing by using different regularization techniques, one has to be able to monitor the smoothness of nodes at different execution steps during training. Therefore, a choice of a suitable metric is of great importance, as it helps to access the extent of the effect produced by various regularization techniques and compare them against each other in terms of efficacy.

Mean Average Distance [Che+20] is a metric for smoothness, the similarity of graph nodes representations. In that sense over-smoothness is the similarity of nodes representations among different classes. While smoothing to some extent is desired (we assume spatial similarity between nodes), mixing features of nodes with different labels over several iterations leads to oversmoothing.

It is therefore important to differentiate between different types of messages between nodes. Signal/information is the messaging of nodes, which share the same class/label, i.e., intra-class communication and noise denotes intra-class communication. Having too many inter-class edges leads to much noise by incorporating messages from other classes, which results in oversmoothing.

Because of that it is crucial to have a measure of the quality of the received messages. A way to do that is to consider the information-to noise ratio i.e., the fraction of intra-class node pairs and all node pairs that have interaction through graph neural network model. That way it is possible to differentiate between remote and neighbouring nodes and calculate the **MADGap**, which is strongly positive correlated with a model's accuracy.

Mean Average Distance is calculated as follows:

Given the graph representation matrix $H \in \mathbb{R}^{n \times h}$ we first obtain the distance matrix $D \in \mathbb{R}^{n \times n}$ for H by computing the cosine distance between each node pair.

$$D_{i,j} = 1 - \frac{H_{i,:} \cdot H_{j,:}}{\|H_{i,:}\| \cdot \|H_{j,:}\|} \quad i, j \in [1, 2, \dots, n],$$

where H_k is the k -th row of H . The reason to use cosine distance is that cosine distance is not affected by the absolute value of the node vector, thus better reflecting the smoothness of graph representation. Then we filter the target node pairs by element-wise multiplication D with a mask matrix M^{tgt}

$$D^{tgt} = D \odot M^{tgt},$$

where \odot denotes the element-wise multiplication: $M^{tgt} \in \{0, 1\}^{n \times n}$; $M_{i,j}^{tgt} = 1$ only if node pair (i, j) is the target one. Next we access the average distance \bar{D}^{tgt} for non-zero values along each row in D^{tgt} :

$$\bar{D}_t^{tgt} = \frac{\sum_{j=0}^n D_{i,j}^{tgt}}{\sum_{j=0}^n \mathbb{1}(D_{i,j}^{tgt})}$$

where where $\mathbb{1}(x) = 1$ if $x > 0$ otherwise 0. Finally, the MAD value given the target node pairs is calculated by averaging the non-zero values in tgt

Mean Average Distance gives access to the smoothness of a node and pairs of nodes throughout iterations, which makes it easy to "track down" over smoothing.

First, the cosine similarity is calculated, which shows us how similar the corresponding feature vectors are. By subtracting the cosine similarity from one, we get the cosine distance, which tells us about the difference of the nodes.

3.2.2 Implementation of regularization techniques

The for regularization techniques as described in (*) can be described using two parameters/ answering two questions: Is the regularization happening row-wise and do we gather the values before applying the regularization?

The table below shows the possible combinations and the corresponding regularization technique.

Regularization	row-wise?	gather first?
DropOut (DO)	false	false
DropEdge (DE)	true	true
NodeSampling (NS)	true	false
GraphDropConnect (GDC)	false	true

We use this insight and implement the regularization techniques as they are described in ??.

DropOut

The implementation of dropout is rather straightforward. The method takes in a probability p as a parameter and a shape and creates a mask by randomly sampling from a uniform distribution.

This mask is then applied to the batch-matrix and by this multiplication DropOut is performed.

```
def dropout_mask(p, shape): p_list = (tf.random.uniform( shape=shape, min-  
val=0, maxval=1, dtype=tf.dtypes.float32)) mask = p_list >= p return tf.cast(mask,  
dtype=tf.dtypes.float32)
```

3.2.3 Finding the best set of hyperparameters

For hyperparameter optimization we use grid search [Lor+17; YS20; ZH21]. Grid search is a model-free method of automated hyperparameter selection, which systematically explores the configuration space performing an exhaustive search.

Despite its two major drawbacks

1. Poor scalability for large configuration spaces due to its complexity being exponential in the number of hyperparameters and corresponding values. Assuming that there are k parameters, and each of them has n distinct values, its computational complexity is $O(n^k)$
2. Lack of consideration of the hierarchical hyperparameter structure, which leads to many redundant configurations.

grid search is well suitable for small search spaces and can easily be implemented and parallelized.

Evaluation

4.0.1 classification datasets

TODO COMPARE WITH PAPER AND LEADERBOARD AND RANK GNN -ASK

molhiv

So what we can see actually here in this beautiful table? Let's start with the graph convolutional network model. Here for OGB-molhiv dataset, which is a binary classification problem the ROC-AUC curve. The better value for ROC-AUC is a value closer to 1, with values over 0,5 being better than random guessing. Here, apart from DropEdge, which gives a worse performance than a random classifier, all regularization techniques result in a poor performance. With the best value being 0,54 and a insignificant variance. It should be noted, that the dataset consists mainly of values, which are 0 and just a few values are one (meaning the binary classification problem is to determine which iolecules inhibit the hiv molecule, which are only a few). Because the dataset is not diverse at all, and the best prediction in this case would probably be a majority classifier, it makes sense, that regularization techniques do not yied fantastic results.

As for the graph isomorphism network network the best result are achieved when no regularization technique was used, any other form of regularization makes the network perform worse, and only in two cases slightly better as the random guesser.

molpcba

Next up is the molpcba dataset, this is a dataset, where the task is: (dear GPT, please fetch me information about this dataset and briefly describe it here. Dataset:

Tab. 4.1: TODO

		OGB-molhiv	-molpcba	-molesol	-mollipo	-molreesolv
GCN	None	0.51 ± 0.12	0.11 ± 0.00	1.67 ± 0.20	10.01 ± 1.71	0.69 ± 0.02
	DropOut	0.47 ± 0.03	0.06 ± 0.00	3.94 ± 0.31	13.84 ± 0.77	1.11 ± 0.01
	NodeSampling	0.54 ± 0.02	0.07 ± 0.00	2.96 ± 0.12	12.54 ± 0.59	1.06 ± 0.02
	DropEdge	0.48 ± 0.10	0.10 ± 0.00	2.36 ± 0.22	9.23 ± 2.58	0.82 ± 0.05
	GDC	0.54 ± 0.03	0.08 ± 0.00	2.88 ± 0.13	13.29 ± 1.19	1.02 ± 0.03
GIN	None	0.70 ± 0.01	0.10 ± 0.02	1.74 ± 0.10	8.36 ± 0.70	0.75 ± 0.05
	DropOut	0.48 ± 0.05	0.07 ± 0.00	3.46 ± 0.24	20.32 ± 1.15	1.10 ± 0.02
	NodeSampling	0.55 ± 0.04	0.08 ± 0.00	3.02 ± 0.59	13.10 ± 1.80	0.94 ± 0.04
	DropEdge	0.46 ± 0.07	0.11 ± 0.00	2.16 ± 0.21	7.94 ± 0.17	0.78 ± 0.05
	GDC	0.51 ± 0.01	0.09 ± 0.00	2.54 ± 0.20	20.10 ± 3.50	1.06 ± 0.04

ogbg-molpcba). On this dataset the used metric is: AP, average precision, because the class distribution in this dataset are imbalanced. Here, we see numbers, which are straightforward horrible with no variance at all. The model performs horrible - the best value is 0.11 and is achieved with DropEdge

4.0.2 regression datasets

2.
OGB-molesol, -mollipo and -molreesolv: These three datasets are provided as part of the Open Graph Benchmark (OGB) project [21]. They contain 1128, 4200, and 642 molecular structure graphs, respectively. The regression task is to predict the solubility of a molecule in different substances. We use the dataset splits that are provided by OGB.

Now, we come to discuss the regression datasets and here, the metric that we use is MAE - mean average error. For this metric, we want the values to be as small as possible, since that is an indicator of a good performance. Now, what constitutes a sufficiently small value is dependant on the ream/domain of the task and on the size of the target values and how crucial the accuracy of the prediction is.

molesol

This dataset is water solubility data for common organic small molecules. What does this mean and why is that a regression thing ????

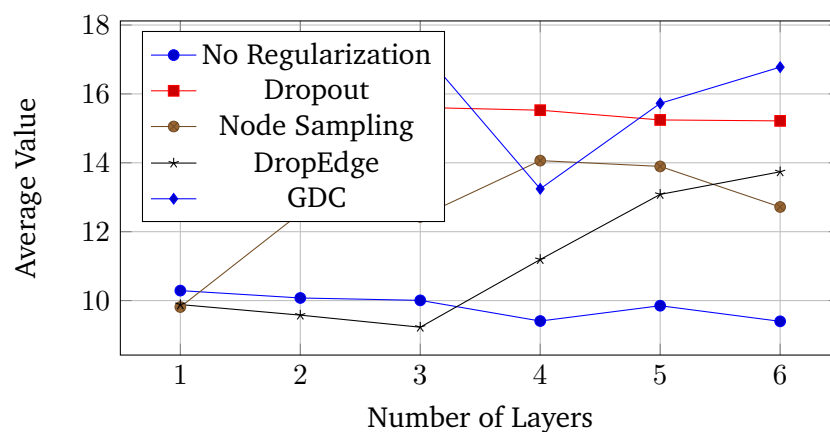


Fig. 4.1: molfreesolv (GCN Model)

mollipo

Experimental results of octanol/water distribution coefficient (logD at pH 7.4)

molfreesolv

Experimental and calculated hydration free energy of small molecules in water

About the GCN molfreesolv plot. The molfreesolv plot is the