One Hot Graph - Lab Report

Alexander Krauck
Department of Machine Learning
Johannes Kepler University Linz
Upper Austria, Austria
alexander.krauck@gmail.com

Abstract

Graph Neural Networks (GNNs) offer a reasonable inductive bias for many machine learning tasks. This includes drug discovery, social network analysis, traffic predictions and many more domains. Many different variations of GNNs, like graph attention networks (GAT) and graph isomorphism networks (GIN) exist. In this report I discuss my experiments which could in future work become the basis that leads to different improvements/variations in GNN architectures.

Firstly, some of my experiments go towards constructing a novel inductive bias for GNNs, the Sinkhorn GAT, which is based on the GAT architecture.

Secondly, a novel and more general addition to GNNs is introduced, the One Hot Graph (OHG). The idea behind the OHG is to create possibilities for more involved inductive biases which should pronounce the importance of the structural properties of each graph. Furthermore, it should open up the possibility of overcoming the bounds on expressivity of conventional GNNs like GAT which is upper bounded by the Weisfeiler-Lehman isomorphism test.

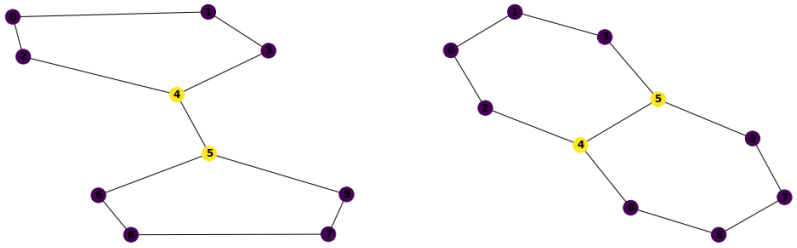
I present the datasets, the basic python code pipeline including some improvements/observations to the used PyTorch Geometric Framework and the carried out evaluation methods that were used for doing the experiments.

1 Introduction

Initially my project supervisor Andreas Mayr, Johannes Schimunek and I discussed that while generally Graph Neural Networks have some beneficial properties, in many tasks more primitive but also more established methods (e.g. methods that count fixed graph substructures) are still used as the primary way of doing graph-classification or other graph related tasks. This is partially due to the fact that the well known universal function approximator theorem [Hornik, 1991] does not directly apply to graph neural networks (GNNs). This means when using a *conventional graph neural network* (e.g. GIN, GAT, etc...), one can not generally approximate any function of graphs that maps to some latent space. This is due to the fact that conventional graph neural network architectures (see Figure 2) will cause some non-identical graphs to always have identical latent representations. An example of such two graphs that can not be distinguished by convenctional GNNs is depicted in Figure 1. A better theory for the expressivity of a graph neural network offers the Weisfeiler-Lehman isomorphism test [Weisfeiler and Lehman, 1968, Xu* et al., 2018, Morris et al., 2019, You et al., 2021].

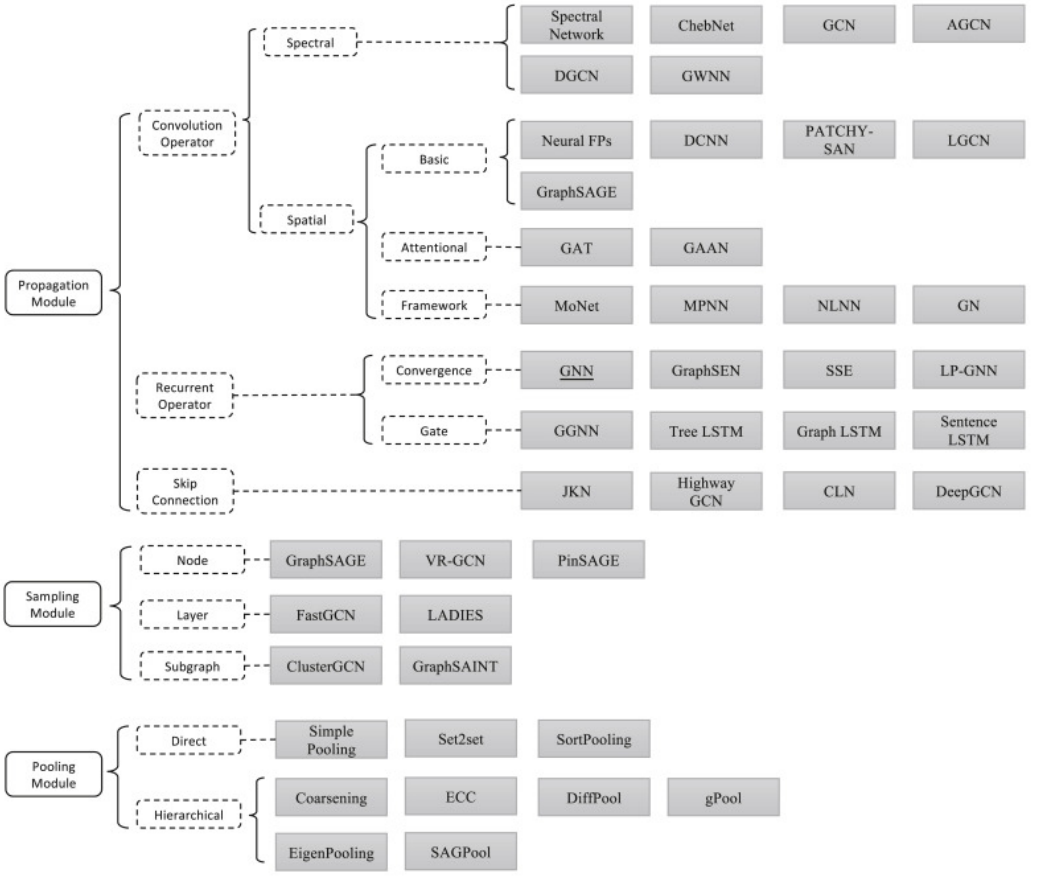
While there are few recent works [Bouritsas et al., 2020, You et al., 2021] that also propose approaches for overcoming the limitations of the expressiveness of conventional GNNs, in this project I focus on a novel way to overcome this limitation which should also provide further benefits to the workings

35th Conference on Neural Information Processing Systems (NeurIPS 2021), Sydney, Australia.



(a) Bicyclopentyl (b) Decalin

Figure 1: These two (molecule) graphs can neither be distinguished by the WL Graph Kernel, nor by any (conventional) GNN like GAT, GIN, GCN.



Source: [Zhou et al., 2020]

Figure 2: Conventional Graph Neural Networks

of the GNN. Furthermore, I investigate ideas for novel inductive biases that could benefit GNNs in some instances.

I first in Section 2 provide some overview on related works in the field of Graph Neural Networks, some important intuition about GNNs and the mayor terminology. Thereafter, in Section 3 I give an overview over the used software libraries, tools and hardware. In Section 4 I introduce the for the experiments used datasets. In Section 5 and Section 6 I give an intuition for the novel GNN ideas that Andreas, Johannes and I came up with and explain some technical details. Finally in Section 8 I discuss the results of my experiments and in Section 9 my project is concluded and I give a preview for possibilities of future work.

2 Related work, Intuition and Terminology

Usually, in GNNs some graph structure (e.g. a molecule graph) is used as input. A graph can be fully characterized by a node matrix $\mathbf{X}$ together with an adjacency matrix $\mathbf{A}$. While generally the adjacency matrix can be non-symmetric, which corresponds to a uni-directed graph, in this work I only consider symmetric adjacency matrices meaning only bi-directional graphs. A GNN usually has multiple layers, which depending on the task at hand produce some output. In this work I only focus on graph-level classification tasks where usually the final output of the GNN is pooled (e.g. max-pooling) and a function, (e.g. a multi layer perceptrone) is applied to this pooled graph representation to obtain the classification results.

The GNN itself usually consists of n layers, where in each layer information between the nodes in the graph is *propagated*. For further intuition it is useful to consider the message massing neural network (MPNN) framework (see Section 2.4 where I further detail the propagation rules relevant for this work). After the propagation usually an activation function (e.g. ReLU) and alternatively for training dropout or some batch-normalization is applied. For the i-th layer in the GNN, the activation is denoted by $\mathbf{H}^{(i)}$. Intuitively $\mathbf{H}^{(i)}$ for each i together with $\mathbf{A}$ again characterize a refined graph, which is analog to extracted features in deeper layers of a convolutional neural network (CNN) for image-tasks (this is also due to the fact that CNNs are just a special case of GNNs).

There exist many variations of the exact message passing rule (GAT, GIN, GCN, etc...), the final pooling function (e.g. max-pool, combined pooling of all layers, hierarchical pooling between layers similar to convolutional neural networks, etc..) and all in-between steps, making the GNN framework one of the most general neural network frameworks, if not the most general, to exist.

2.1 Convolutional Networks on Graphs for Learning Molecular Fingerprints

Originally [Duvenaud et al., 2015] proposed the first GNN. The idea was to use the same technique that is so successful for image-tasks, convolutions that is, to be used for arbitrary graphs (in the case of Duvenaud et al. for molecule graphs). That means the same filter (shared weights) should be applied to each node and its neighborhood. This new proposed method was able to outperform more primitive methods that simply use circular fingerprints (not-differentiable methods).

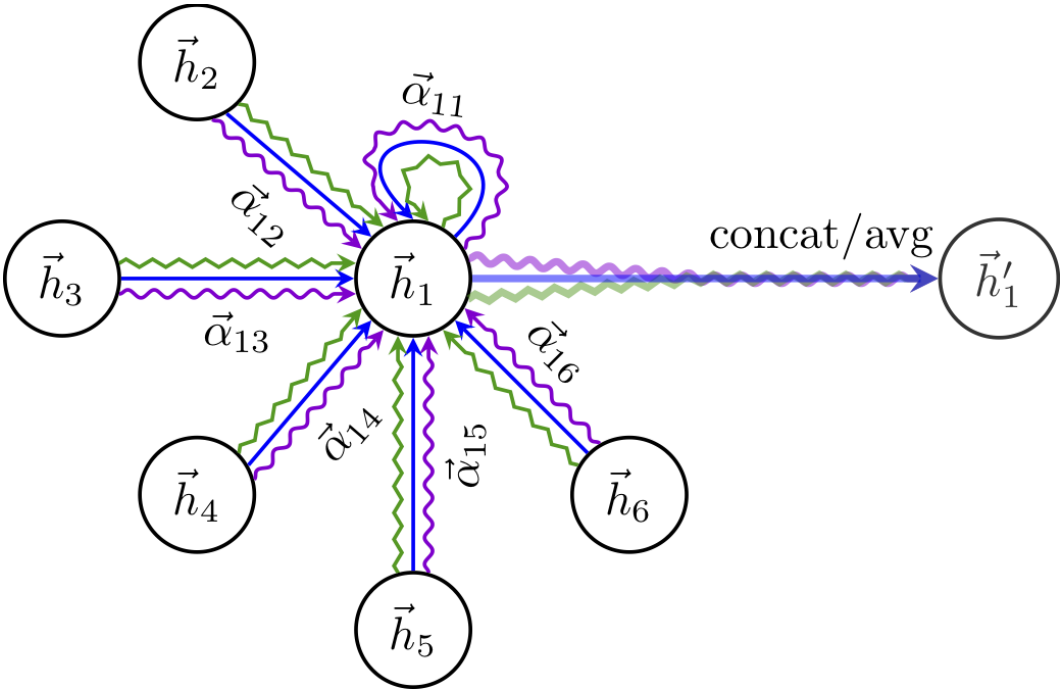
2.2 How Powerful are Graph Neural Networks?

In this work by [Xu* et al., 2018] it is discussed in detail that the expressivity of conventional GNNs is bounded by the Weisfeiler Lehman Graph Isomorphism test. Furthermore, a novel GNN architecture, the graph isomorphism network (GIN) is introduced which should come closest to this representational bound. In Section 6.4 I discuss the isomorphisom one hot graph which aims to extend the GIN architecture.

2.3 Graph Attention Network (GAT)

The GAT architecture originally introduced by [Veličković et al., 2018], uses an attention mechanism on top of the well known message passing GNN concept by [Kipf and Welling, 2016]. This means each message has a corresponding attention score or attention weight. Intuitively in each layer of the graph neural network, all nodes send messages to all their neighboring nodes (and usually to themselves), but each node can only receive messages with their weights α summing up to one. This can be seen in Figure 3. For the GAT, the attention weights themselves are determined exclusively by the current activation of the nodes. In particular the weight for one message always depends on the activation of the sending node and on the activation of the receiving node of this message. Looking at

the process from a different perspective, what happens is that in each layer of the GAT, the entries of each row of the graph adjacency matrix is weighted by the corresponding attention score, making the matrix a stochastic matrix (which means that the row sums are equal to one).



Source: [Veličković et al., 2018]

Figure 3: The GAT mechanism

2.4 Neural Message Passing for Quantum Chemistry

In this paper from [Gilmer et al., 2017] one of the currently most used GNN frameworks, the message passing neural network (MPNN), is introduced. The intuitive idea is that in each GNN layer l messages $\mathbf{M}^{(l)}$ (a tensor of order 3), which depend on the sending node i, the edge (which in my case is binary) and the receiving node j, to the neighboring nodes using some message function $\mathbf{M}_{ji}^{(l)} = \text{message}^{(l)}(\mathbf{H}_i^{(l)}, \mathbf{H}_j^{(l)}, A_{ij})$. Thereafter, for each node j the received messages are combined with the current activation of the node using some update function, which together with the layer activation, leads to the next GNN activation $\mathbf{H}_j^{(l+1)} = \text{activation}(\text{update}^{(l)}(\mathbf{M}_j^{(l)}, \mathbf{H}_j^{(l)}))$. In practice most operations are done sparsely.

This is particularly interesting for my work, as it provides a very intuitive and comprehensive framework that I use also for the implementation of the One Hot Graph framework as can be seen in Section 6.

2.5 Semi-Supervised Classification with Graph Convolutional Networks

In this publication by [Kipf and Welling, 2016], the well known Graph Convolutional Network (GCN) was introduced. The GCN propagation rule is motivated by a first-order approximation of localized spectral filters on graphs. Intuitively, what happens is that nodes with lots of connections will send less information to each connected node but will also receive less information from each connected node relative to a node that only has few connections.

In particular this is interesting for my project, as the Sinkhorn Graph Attention Network intuitively does something similar, as it forces each node to be sending only information of weight one and receiving only information of weight one. This directly means that nodes with more connections will send less information to each node than a node that only has a few connections.

2.6 DeepTox: Toxicity Prediction using Deep Learning

In this publication, [Mayr et al., 2015] first introduced the use of deep learning architectures on molecule data. Molecule data is important for many research fields including drug discovery. In the work by Mayr et al. the winning solution for a toxicity prediction challenge of molecule data using deep learning is proposed. Furthermore the work offers insight into the required considerations for drug discovery tasks and the handling of molecule data.

2.7 Large-scale comparison of machine learning methods for drug target prediction on ChEMBL

In this work, [Mayr et al., 2018] compare the performance of deep learning methods to other target prediction methods for drug discovery on a large scale dataset. Furthermore, they provide insight into their use of cluster-cross validation as performance evaluation method to keep the validation results as unbiased as possible which is very important for evaluating new methods.

2.8 Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks

In this publication, [Morris et al., 2019] proposes a higher-order version of GNNs, the k-GNN. This can overcome the bound of expressivity of conventional GNNs that is given by the WL-test. The general idea of this approach is to extract all sub-graphs of this graph with size k. Then this extracted sub-graphs are treated as the new nodes of this graph. The publication contains some variations, like a hierarchical variant or a variant where only some sub-graphs are sampled to keep the computational cost at bay.

The computational costs of this approach are for increased k and bigger graphs infeasible. Furthermore, with this approach it is only possible to do graph level tasks, but node- or edge-classification tasks are not possible. However, it might be interesting to view this approach not so much as a whole GNN variant but only take into account that using this approach a sort of graph pooling is established which could have potential for future work.

2.9 Weisfeiler-Lehman Graph Kernels

In the work of [Shervashidze et al., 2010] many variations of the Weisfeiler-Lehman Graph Kernel are proposed. This is particularly relevant for my work as the WL-test upper bounds the representational power of conventional GNNs.

2.10 Identity-aware Graph Neural Networks

Recently [You et al., 2021] proposed a novel GNN architecture, the ID-GNN, that successfully overcomes the representational bounds of the WL-test in a K-hop distance from each node. The idea of this approach is that given a graph with n nodes, n sub-graphs "ego networks" with K hop distance that are centered at the n different nodes are extracted and the center node of each sub-graph is given an binary identity coloring. Now each of the n sub-graphs is simply passed thru a conventional GNN (e.g. GIN), where nodes that are in the computational graph at the identity coloring are using a different message function (with different weights) than the other nodes. Finally only the center node of each sub-graph is kept as the final representation.

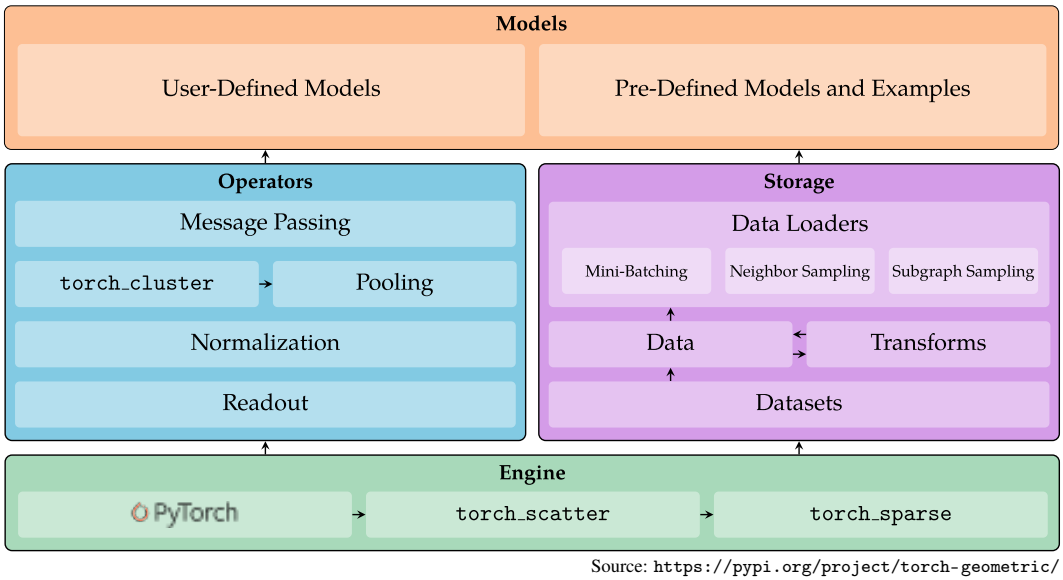
While this approach has good results and can increase the representational power of all conventional GNNs, with increase graph size and increased K-hop distance this approach becomes very computationally costly. However, a mayor benefit of this approach is that graphs of any size can be, even tho it might be slow, computed, as basically only small graphs of K-hop distance are required in memory. This means one could for example input a gigantic social media graph with 1 million nodes but as only K-hop sub-graphs are used this can be done effortlessly, provided the required parallelization routine.

3 Libraries, tools and hardware

3.1 GNN implementation

PyTorch Geometric (PyG) [Fey and Lenssen, 2019] (Version 1.7.2), which is based on the PyTorch framework [Paszke et al., 2019] (Version 1.9.0), was used as the main infrastructure for my experiments. This framework promises efficient training and inference of GNNs and many ready-to-go GNN architectures like GAT, GIN and GCN. Furthermore, the framework implements an interface for loading various datasets that are fit for doing experiments with graph neural networks. The rich infrastructure of PyG is depicted in Figure 4.

However, I was writing or rewriting big parts of the GNNs as can be seen with more details in Section 7 that I was doing experiments with myself, as the PyG framework does not allow for too different GNN styles or would greatly decrease the performance when being used for my purposes.



Source: `https://pypi.org/project/torch-geometric/`

Figure 4: PyTorch Geometric Infrastructure

3.2 Result analysis

TensorBoard from the TensorFlow framework [Abadi et al., 2015] was used for logging and analyzing the training statistics of my novel GNN implementations as well as the baselines for comparison. In particular for train-, test- and validation-sets I logged the area under the receiver operating characteristic curve (AUC-ROC) for each label or the accuracy, depending on the dataset. Additionally I logged the training loss (binary-cross entropy). Furthermore, I logged the exact hyperparameter configuration and seed for each run together with the best scores to easily make hyperparameter settings comparable. I also included a routine that makes multiple runs on different seeds and averages the results.

My personal takeaway from this part of my experiments is that it is extremely important to first decide on a very clear scheme of how to do the logging and the experiment-analysis and only if this is very clear, to start running the experiments. The reason is that with many experiments there will be loads of different log-files, run-configurations, etc... and it can become hard to impossible to keep the big picture of the experiments in mind. In particular for this project I started out only logging some statistics and for later experiments I added many more. Furthermore, as it happens, in some earlier versions there were few little bugs or technical mistakes in the implementation but I did not remove the corresponding log files or document this but just fixed the bugs and continued. Thereby I lost track of many of the earlier valuable results that I had as there was just too much information.

Table 1: Used datasets

Dataset	Graph Type	#Graphs	average #nodes	min #nodes	max nodes
Tox21	Molecule	11758	19.1	2	132
NCI1	Molecule	3288	29.9	3	111
Proteins	Protein	891	39.1	4	620

3.3 Hardware and environment

For doing the different experiments a GeForce GTX 1080 Ti GPUs, which come with slightly more than 11GB RAM, was used. While, I manually run multiple different experiments at once on multiple of such GPUs, I did not implement a pipeline which can automate the parallelization which generally could be beneficial. The used machine runs with the Rocky Linux release 8.4 (Green Obsidian) operating system.

4 Datasets

Andreas Mayr suggested me to look into the below described Tox21 Dataset as it contains molecule graph data which was already used by him and the institute. Initially, my goal was to achieve similar or better results than the achieved results of the machine learning institute at the Tox21 challenge. However, the used result evaluation for this challenge was not a cross validation but only a single test set was used therefore which in turn means the results might be biased.

Therefore, I later on decided to use cross validation on other datasets found on the paperswithcode website and compare the results to the ones from the other contributors. This website provides results of the latest machine learning methods on relevant datasets in a uniform fashion. Basic statistics about my used datasets are in Table 1.

4.1 Tox21 Dataset

Tox21 is a dataset for multi label classification tasks on graph structured molecule data for toxicity prediction. While a version of this dataset is already implemented in the MoleculeNet dataset-class of PyG, Andreas and I decided that I should use the original version of this dataset, which can also be found at the institutes website[1] for the reason that this version is more complete. Moreover, with this version of the dataset it is possible to compare my scores to the scores of the machine learning challenge where this dataset originates from.

This Tox21 dataset comes in a file with the sdf-filetype, which is commonly used for storing molecule data, and an additional metadata file, which also includes the labels of the data and furthermore the original train/validation/test-split.

In my code infrastructure I built a child class of the MoleculeNet dataset-class from PyG to seamlessly infuse this original data into the PyG framework. It's worth mentioning that 4 graph-samples in the dataset couldn't be used, because the rdkit python-package, which was responsible for converting the sdf-file encoded molecule-graphs into PyG encoded graphs, could not read those samples. As this problem concerned only a very small subset of the data, I ignored this.

4.2 NCI1 Dataset

The NCI1 Ddataset contains molecule graphs that either indicate lung cancer or not and thus can be used for a binary classification task.

I acessed this dataset with the TUDataset[2] class from PyG, which makes many graph-datasets from the Technical University of Dortmund directly accessible [Morris et al., 2020].

[1] `http://bioinf.jku.at/research/DeepTox/tox21.html`

[2] `graphlearning.io`

4.3 Proteins Dataset

The Proteins dataset contains protein data where each node is a amino acid and 2 nodes are connected if they are less then 0.6 nanometres apart. The goal is to predict if a protein is an enzyme or not. The reason why I chose this dataset is that I wanted to find out how my different methods perform on other data than molecule-graphs.

This dataset was also accessed using the TUDataset lass from PyG.

5 Sinkhorn graph attention network

5.1 The idea

The first idea that I looked into during my project, was to refine the GAT architecture. In each layer of the GAT, I wanted the attention matrix which decides how much information from one node should go to a neighboring node to be double stochastic (non-negative and all row sums and all column sums should be 1). That means each node should send the same amount (in terms of attention-weight) of information as it receives. This I implemented using the iterative Sinkhorn-Knopp [Sinkhorn and Knopp, 1967] algorithm on the attention matrix.

5.2 Implementation details

For this implementation I oriented myself on the PyG implementation of the GAT and changed the message function which includes the main attention mechanism. While GAT uses softmax for normalizing the receiving attention, it turned out that this is not beneficial for the Sinkhorn GAT network. Particularly, while the softmax function normalizes the exponentiated elements using the sum of all receiving edges, I use a sinkorn calculated normalization weight which is on both, the receiving and sending edges and approximates a double stochastic matrix. That means I first take the exponent of the un-normalized pre-attention weights like in a softmax but then I normalize with a sinkhorn-algorithm calculated columnnorm and rownorm value.

$$\alpha_{ij} = \exp \text{preatt}_{ij} \cdot \text{columnnorm}_j \cdot \text{rownorm}_i$$

with

$$\text{sum}(\alpha_i) \approx 1 \text{ and } \text{sum}((\alpha^T)_j) \approx 1$$

The way the Sinkhorn-Knopp algorithm is executed while still allowing efficient back-propagation is similar to the reparameterization trick which is used for VAEs which allows propagating through a probabilistic distribution such as a Gaussian. The inspiration for this idea came from Andreas as he said that it would not be reasonable to back-propagate thru the iterative Sinkhorn-Knopp algorithm. While the mathematical sophistication of using this trick here is out of the scope of my project, it yielded satisfactory results.

Initially my implementation only worked for single head attention but I later added support for multi head attention which also boosted the performance.

Furthermore, the sinkhorn algorithm usually continues till convergence but I found that it is faster and satisfactory in performance to fix the number of sinkhorn iterations as a hyperparameter.

Finally, as the idea behind the sinkhorn graph attention network is quite similar to the GCN I implemented some by the GCN structure inspired code. However, this code did not perform as smooth as I would have expected and thus I did not investigate so much more even tho this might have potential for future work.

6 One Hot Graph

6.1 The idea

The idea is to give each node in a graph an unique identity, using a one-hot encoding. More specifically, this one-hot encoding should be treated as an additional vector for each node in a graph, in the following called *one-hot vector*. When passing the one-hot vectors in a similar fashion as the

node vectors thru a GNN and using an appropriate function to add this information to the output of this GNN, then it should be possible to distinguish any two graphs from each other, provided that the GNN is deep enough.

Furthermore, this one-hot vectors could be used to add an useful inductive bias to GNNs. The specific idea was to use an algorithm for the one-hot vector which basically counts up how often the information from one node has been delivered to the other node. Using this count information it is possible to implement an attention mechanism that gives more attention to neighboring nodes that deliver information that has not been seen in one node as can be seen in Section 6.3. It seems reasonable to prefer new information before old in each node. MOreover, this one-hot vector could be used in each layer of the GNN to add further information to the MLP that is present in the GIN as can be seen in Section 6.4.

6.2 Passing the one-hot vector

Initially, as there is one one-hot vector for each node in a graph with n nodes and the adjacency matrix $\mathbf{A}$ we have a one-hot matrix $\mathbf{O}^{(0)} = \mathbf{I}$ that is equal the identity matrix of size $n \times n$. The propagation rule for the k-th layer of the GNN with respect to the one-hot part then is in a very basic form

$$\mathbf{O}^{(k)} = (\mathbf{A} + 2\mathbf{I}) \cdot \mathbf{O}^{(k-1)}$$

The reason for adding the identity twice is to make sure that the position of the initial one hot encoding always stays the biggest number in the one-hot vector.

Moreover, as using this update rule will lead to an explosion of the values of the one hot matrix it is possible to use some monotonous function after each iteration. I decided to implement two different versions. In one version I only add 1 to the corresponding index if the neighboring node has an entry greater or equal 1 in at that index. In the other version the an index is at most increased to 1. However, it is imaginable that more involved functions here, like the logarithm could be even more reasonable.

6.3 Attention One Hot Graph (AOHG)

In this architecture the aforementioned idea of the inductive bias which intends to increase the attention-weight for new information in the message passing step of the GAT is realized.

The AOHG adds another source of structural information to those attention weights which can potentially be a valuable inductive bias. Similarity scores of the one-hot vectors of the sending and the receiving nodes are calculated (e.g. using cosine similarity or intersection over union). Inverting those measures of similarity provides a measure of distance. If the one-hot vectors of two neighboring nodes are very distant by those measures, then this indicates that the first node must have received some new information which the second node has not yet received. If this is the case, then those two nodes should attend more to each other such that more diverse information tends to be spread farther in the graph.

6.3.1 Technical Details

The goal is for each neighboring pair to have a one-hot refined attention weight. That now means we have the usual attention weight α_{GAT} between two nodes in a GAT and additionally the one-hot vectors of the sending and receiving node o_s and o_r respectively in the message function. Using a measure of distance on o_s and o_r (e.g. inverse cosine similarity) yields an attention weight α_{OHG}. Now those two attention weights can simply be multiplied to obtain the final attention score.

Moreover, is important for the distance function to be invariant of permutation. The reason therefore is that the index of a node in a graph is completely random and the result of the attention weights should be the same even if the indices are shuffled. Additionally, it is important for the distance function to also take inputs of different size because different graphs have different sized one-hot vectors.

Finally, as the one-hot vectors tend to explode with the rule that I used, I decided to also downscale o_s and o_r with a version of the logarithm function.

6.3.2 Implementation details

At first I tried again, as with the Sinkhorn GAT, to use the same basic structure as PyG suggests. This direction turned out to be not very promising for several reasons. The code of the PyG framework, while working quite well for basic architectures or for architectures that are already implemented in the framework, is not very readable and hard to understand as it uses a quite particular python code structure. There are several, PyG framework intern, high level python instances used, which are not for making the code more efficient but only for creating more abstraction of the python code. While this odd structure might be powerful if implementing it as an own library and making it common fashion to be used in python, it makes it very difficult when trying to understand what is happening. Class examples where this odd stucture can be seen include the MessagePassing class and the dataloader infrastructure of PyG.
In combination with the special mini-batching algorithm in the PyG it took me several days to really get a hang of everything that is going on.

In the first version of the implementation I still tried to hang on to the structures of Pytorch Geometric, which is not easy, as the addition of the one-hot vector which is required makes everything a bit more complicated. I ended up making a for loop over each minibatch, which worked, but resulted in a really slow training and inference performance. This was required as otherwise the AOHG requires to allocate a matrix in the size of the number of nodes in the minibatch squared which would sometimes go over 100GB in the case of the Tox21 dataset.

As training a single AOHG network with that structure took more than 10 hours, I decided to tear up the PyG structure entirely, which included a different way of mini-batching the dataset than PyG suggests as can be seen in Section 7.2. The resulting structure is while consuming more memory than the original mini batching of pytorch geometric faster than the original PyG, making it a core contribution of my work.
In particular using the original procedure with the GIN implemented in PyG we can make approximately 7 mini-batches with 256 samples each per second during training. In comparison my new mini-batching procedure can achieve more than 8.2 mini-batches of the same size per second. That is, using the same underlying GNN architecture (I.e. using the same weights, the output is equivalent). It is left for future work to investigate the exact reason for this behaviour and/or to implement this type of batching in a GNN infrastructure library like PyG.

If using only one attention head, the AOHG consistently outperforms the GAT architecture with more than 2% MAR performance boost increase on the Tox21 dataset. However, when using more heads (e.g. 8) the GAT comes closer and closer to the AOHG, while the performance of AOHG does not increase much. It seems that while the GAT is more consistent with the scores, the AOHG has, when using different random environment seeds, higher variance of performance peaks, providing slightly better models, sometimes. That means if the mean of the scores of AOHG and GAT across seeds is compared, they are rather similar, but when taking the maximum, the AOHG wins, coming close to the performance of GIN in some cases (while still being slower than the GIN architecture).

6.4 Isomorphism One Hot Graph (IOHG)

This architecture is more concerned with the initial idea of graph isomorphism. The mini batching procedure used for this architecture is the same as for the attention one hot graph. The basic idea was to use a similar structure as the GIN architecture, which is already concerned with coming as close as possible to the representational power of the WL test, but then to enhance this architecture with the one hot information using some invariant function. Johannes Schimunek initially suggested just taking the maximum of the one hot vector, which is invariant and concatenating this to the usual representation of the nodes. My idea was to instead somehow use the descendent sorted one hot vector which is also invariant. Additionally, before concatenating this to the usual representation Andreas and I thought about first using some activation function which downscales these highest n elements. I proposed two additional possible ways of enhancing this additional information. Either one cuts off the first n highest (and first) elements of this vector and feeds it into some layers of a FFNN and concatenates this. The other idea was to use a 1D convolution and a GAP to bring the representation to a fixed size and concatenate this to the node vectors. Everything else about the GIN architecture stayed the same. I was able to show that this architecture produces different final graph representations for the two above isomorphic graphs, while the AOHG still produces the same

fingerprint for both. While this is a success, the results are, at least on the datasets where I run my experiments, not much better than the results of the GIN. (check experiments)

6.4.1 Technical details

7 Implementation and Pipeline

In python I implemented an experiment pipeline which enabled me to do extensive and random hyperparameter grid search using an external configuration yml file, environment seeding, n-fold cross validation, TensorBoard logging and saving the best scoring architectures to disk. Furthermore, without changing any code it is possible to switch between different datasets (Tox21, NCI1, Proteins) using this pipeline. GIN, GAT and GCN architectures were used as baselines for all my experiments. A basic sketch of the training pipeline can be seen in Figure 5.

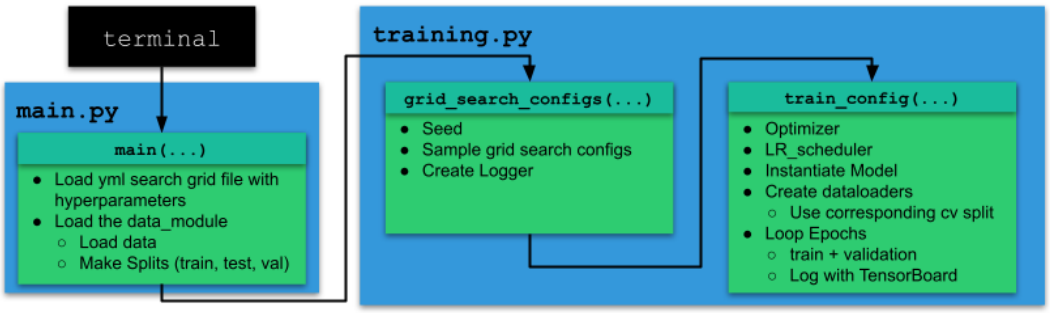


Figure 5: A basic sketch of the used training pipeline

7.1 Loading Data

It depends which dataset and which configuration I am using for this part of the pipeline. In particular when loading the data for the first time it first is being converted to a PyG dataset class. For the used NCI1 and Proteins datasets this was already done fully by PyG but for using the original tox21 dataset the data first needed to be converted from the provided sdf files and also the original train,test and validation splits very considered. Therefore I had to modify the ExtendedMoleculeNet class of PyG to load the original dataset instead of the modified.

7.2 Improved Graph Minibatches

The commonly used method of minibatching graph data for deep learning and that also PyG uses has reasonable advantages but also mayor setbacks which made it impossible to keep it for the one hot graph architectures.

In particular usually all graphs in a minibatch are intuitively being merged together while still being unconnected in the adjacency matrix. This gives the advantage that only the lowest possible memory is being used for processing graphs.

For one hot graphs however, this becomes infeasible as it is required to have memory space of $\mathcal{O}(n^2)$ where n is the number of nodes. If all graphs in a minibatch are combined to one graph then this graph has for my purposes too many nodes to be kept in memory.

The solution for this is to change the way the mini-batches are being formed while still staying memory and computationally reasonable. I considered a method that is also being used in transformer neural networks. Transformers always take input of the same length but when you have an input that has a shorter length a 0 symbol to fill the remaining input space is being applied (e.g. if you have a shorter sentence in a natural language processing transformer). I decided for each minibatch to determine the largest graph in terms of nodes and also the largest graph in terms of adjacency matrix entries. Then I extend each graph by nodes filled with 0 to become the size of the largest graph in terms of nodes and the adjacency "matrix" (which is an 2 dimensional array where the first dimension is for the sending node and the second dimension is for the receiving node) is extended by 1 entries

to be the same size of the largest adjacency "matrix" in the minibatch. After those graphs have been extended to be of uniform size they all can be combined into one big tensor which is then passed to the GNN.

While this solution takes strictly more memory than the method of PyG, in my experiments it had faster computation times than the PyG alternative also when using this minibatching method for GNN variants that PyG implements itself like GIN. It is worth mentioning that the minibatches produced with my method are are not directly compatible with the GNNs of PyG as the internal operations (propagation rules) have to be different.

My mini-batching algorithm is similar to the algorithm that is used in transformer models. It memory scales for usual graphs like GIN in $\mathcal{O}(bnm + be)$, where b corresponds to the mini-batch-size, m to the size of hidden vector, e to the number of edges and n to the number of nodes. Where e and n are taken from the maximum that appears in the particular mini-batch. For a AOHG with 1 head in $\mathcal{O}(bnm + bn^2 + ne))$. As can be seen, the AOHG only scales quadratically in the maximum number of nodes in one graph, which is no problem for smaller graphs (e.g. for molecules), but is left for future work to make more memory efficient. In particular it is not possible represent the one-hot vectors in a sparsely in a straight forward manner, as we can for the adjacency matrix. The reason therefore is that the "index-select" operation, which is required for propagating graph messages from the sending to the receiving nodes is not fast, when applied to a sparse one-hot matrix, from a sparse adjacency matrix.

7.3 Seeding and cross validation

For keeping experiments reproducible I decided to work with environment seeds. However, to keep the pipeline a bit simpler I decided to combine seeding and cross validation. That means if I e.g. decided to make a 10-fold cross validation, for each fold the environment got the next seed. Furthermore I decided to by default and when not using cross validation always take the same seed when using a grid search to avoid wrong feedback from using different configurations.

While for the purposes of my experiments it was sufficient to implement seeding and cross validation in that way, it would be an improvement in my pipeline to make cross validation and seeding independent of each other.

7.4 Hyperparameters

For testing out my different architectures I created some hyperparameters additional to the ones that are already commonly used (in GIN,GAT,GCN, etc..) to better understand the behaviour of the architectures. Not all hyperparameters turned out to be useful but yet gave valuable insights. Below i will describe all hyperparameters that i have used during my experiments.

number of hidden channels decides the size of each node in the graph should be.

number of linear layers decides how many layers of feed forward nn should be after the gnn to come to a prediction for the graph.

number of graph layers decides how many layers the graph should have or how often the graph should propagate.

dropout decides how many layers of feed forward nn should be after the gnn to come to a prediction for the graph.

learning rate and weight decay parameters are as in a regular nn. I used 10^{-3} for the most part as it performed best.

batch size decides how many graphs each minibatch should contain. For the nci1 and tox21 datasets i stuck to 256 graphs per minibatch but for proteins I had to reduce to 128 graphs per minibatch as this dataset contains some graphs that are very large and with my speed improved minibatching method this leads to a quite big increase in required momory per minibatch that contains a large graph.

one hot channels is the number of channels that should be used of the one hot vector to be added to the hidden vector of each node in the graph to enhance the node information. Primarily this is used in the IOHG but I also added the possibility to the AOHG to allow further experiments with combination of both ideas.

one hot mode decides which function should be used on the one hot vector before it is added to the hidden vector of each node. Possibilities that I implemented are feed forward network, 1-d convolutional network or just the identity.

first n one hot decides how big the input to the one hot mode function should be. Depending on this the one hot vector is after sorting cut off or extended with zeros.

one hot increase decides the one hot vectors are increased after each propagation of the graphs. Options are "add" which simply adds the vectors together, "binary add" which adds only ones for each neighbor that has values greater zero for each one hot position respectively and "indicator" which increases each position in the one hot vector to at most 1.

use normal attention determines in the AOHG if the usual attention mechanism that also is used in a GAT architecture should also be used or if only the one-hot attention mechanism should be used.

heads decides how many attention heads the AOHG should have.

one hot attention constant decides the (initial) value of the constant that is used to avoid division by zeros in the one hot attention calculations. By default this is set to 1.

train one hot attention constant can enable training of the above mentioned constant. Results with this constant being trainable are rather confusing as interestingly often the constants are being pushed towards being negative.

8 Experiment evaluation and results

Initially I decided to run some tests with built in architectures from PyG like GIN or GAT which was also helpful for building an initial pipeline. Moreover I decided to initially only use the Tox21 dataset with the train/test/validation split that was also used during the official Tox21 challenge as my supervisor recommended using this data. For evaluating my experiments I studied the plots that tensorboard produced. For each epoch I made evaulations on both the validation and also the official test set. My reasoning for also evaluating on the test set was to better understand how results with different graphs would differ with this kind of data. I compared the results of GIN and GAT to the results of the official Tox21 challenge[3] where it turned out that those two architectures alone are already competitive for this task even without fine tuning. However, the best validation result does not directly yield the best test result and vice versa because clearly those two sets are different and thus, even tho the results are still reasonable they are not better than the challenge results.

As GAT and GIN gave me a decent intuition on how results should look like I decided to use those two as benchmarks to compare the results of my new architectures to. As the idea of AOHG was to enhance the attention mechanism inside a GAT my goal for AOHG was to produce better results than GAT. Similar the goal for IOHG was to enhance the GIN architecture and thus my goal was to produce better results with IOHG than I could with GIN.

The SGAN is a modification of the GAT imposing a strong inductive bias on a network. However, it is not necessarely true that this inductive bias should perform better than GAT on all tasks. While I have tested SGAN it was difficult for me to achieve great results with it on Tox21. However, I assume given the right type of graph this architecture could still be useful.

8.1 Train set versus test set evaluation

Generally in machine learning tasks it is more important to look at the scores on the test set than on the train set because of overfitting. However, for my purpose there are some good reasons to also

[3] `https://tripod.nih.gov/tox21/challenge/leaderboard.jsp`

consider the results on the train set. Firstly, this is due to the fact that I introduce some completely new architectures that might or might not learn at all. Thus it makes sense to study the loss and scores (AUC-ROC) on the train set. Secondly, I also changed a lot in the infrastructure of PyG and thus I had to confirm that the new method of minibatching for graphs generally does the same thing and learns. While I numerically confirmed that the results are equivalent it was also useful for me to always keep an eye on the train loss to gain more intuition. Finally, at least IOHG is concerned with the task to improve expressivity of GNNs and thus it was interesting for me to keep an eye out for the performance on the training set. The idea is that the ability to fit should be better than for GIN as IOHG is simply an extension of GIN. Generally for graph tasks, the unavoidable error for IOHG should be smaller than for GIN. However, while I did not observe mayor improvements on the train set between IOHG and GIN, there might be data sets that would greatly benefit from the increased expressivity of IOHG and thus yield significantly better results.

8.2 Cross validation and cluster cross validation

While always in the back of my head, initially i did not implement the pipeline for cross validation. The reason therefore is that I stuck to the fixed train/validation/test split that came with the Tox21 dataset in order to have more comparable results with the Tox21 challenge results. However, later on I realized that those results might be biased and enhanced my pipeline to allow cross-validation. As [Mayr et al., 2018] proposed, it would also be reasonable, in particular for molecule data, to use some more advanced form of cross validation like cluster cross-validation. As I wanted to compare my results to the results of other people or of state of the art machine learning architectures it was hard for me to use something like cluster cross-validation as still usually just conventional cross validation is being used to score a method. Furthermore, I thought it would go beyond the scope of my project to implement a pipeline therefore.

8.3 Setbacks and difficulties while evaluating

GNNs being a rather fresh area of machine learning for me and in general made it difficult for me on many occasions to draw meaningful conclusions from the results that I had. While scores on the test set were even better than the results of the winning team in the Tox21 challenge this was fairly inconsistent with the results on the validation set. Moreover, in the over 150 experimental runs that I made and constant changes in the pipeline to create improvements I have to admit that I was not prepared for the overload of information. Furthermore, it was very difficult to keep track of all the different runs that I made, some made with errors in the pipeline and not deleted or some just forgotten. Finally, as I also tried experiments where I used different seeds for the same architecture it turns out that results can strongly vary. Concluded it can be very difficult to make the right or any conclusions when having different results in the dimensions of data-fold, seed and hyperparameter configuration.

8.4 Results on Tox21

For the results that I present for Tox21 (not for proteins or NCI1) I did not look at the validation set to pick the best result there and take the corresponding result on the test set but instead I simply choose the best result in the test set. This is because I was not so much interested in having a extremely realistic scenario but more in gaining more insight into the techniques and theoretical ability of my gnn implementations. For further experiments or and official paper it would be good to do the test-validation procedure like hinted.

The best result on Tox21 of all my experiments in terms of AUC-ROC on the test set was achieved by the IOHG with 83.58%. As this is only 0.15% ahead of the GIN I would not draw any final conclusions here but it is an indication that it could have potential.

For an attention based algorithm when only using 1 attention head the GAT could not achieve 76% on the test set while the AOHG with normal attention disabled could achieve over 81.97%[4]. Furthermore, the sinkhorn variant could achieve 81.01% which is also better than the GAT without any modification.

[4]hidden channels: 1024, linear layers: 2, graph layers:5 , linear dropout: 0.2, graph dropout: 0.2. lr: 0.001, weightdecay: 1e-10, batchsize: 256, onehot attention: dot, onehot incay: add, heads: 1, use normal attention: False

Table 2: Proteins Result Accuracy

Method	Average CV	Best Fold	Worst Fold	Best Test Result
AOHG	0.773	0.802	0.748	0.839
IOHG	0.680	0.703	0.640	0.757

Table 3: NCI1 Result Accuracy

Method	Average CV	Best Fold	Worst Fold	Best Test Result
AOHG	0.737	0.762	0.706	0.781
IOHG	0.838	0.866	0.81509	0.871

For attention based GNNs generally the GAT achieved a maximum AUC-ROC of 81.63% over all runs while the AOHG could achieve 83.14% which is more than 1% improvement and almost as good as the GIN variants. However, those results are not particularly consistent for the AOHG as even with the exact same hyperparameter configuration and only by changing the environment seed the result can differ by multiple percent AUC-ROC as depicted in Figure 6.

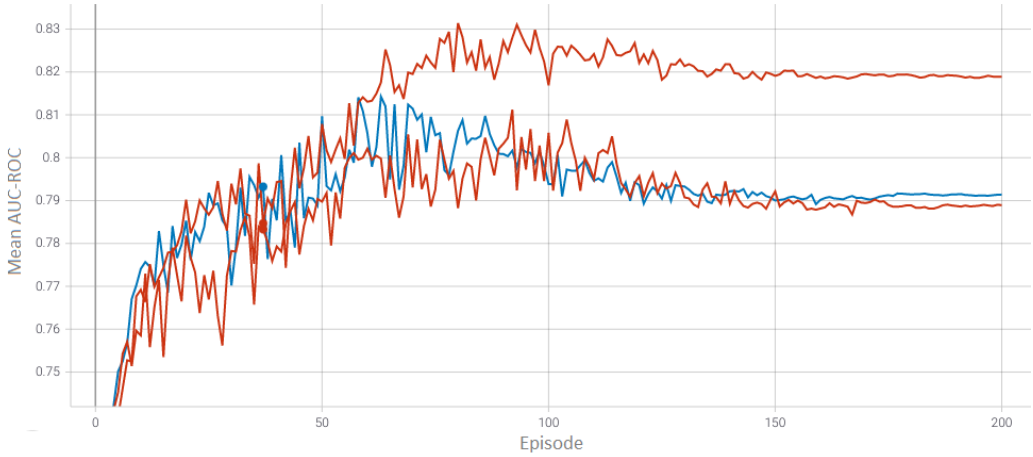


Figure 6: Different test results of the same environmental setting with differing seeds.

8.5 Result for Proteins

For this dataset in Table 2 it is clear that the AOHG with at most performed much better than the IOHG. Moreover, comparing the cross-validation result from the AOHG to the results on paperswithcode, it is clear that this method is already part of the better methods. For both runs on this dataset I only used the best performing hyperparameter configurations that I got from the Tox21 dataset for those tests.

8.6 Result for NCI1

For this dataset in Table 3 it is obvious the IOHG performed much better than the AOHG. When comparing the cross validation result of the IOHG with the performance on paperswithcode also here as with the proteins dataset it is clear that IOHG is competitive already, beating most other methods even without finetuning on this dataset.

9 Conclusions and Future Work

It is apparent that both, the IOHG and AOHG can perform well with different datasets. A thesis of me is that the IOHG is a better fit for molecule data and the AOHG better for protein related data.

Moreover, I observed that results can heavily vary even for the same data and hyperparameter settings only by changing the environment seed that initializes the neural network.

It might be interesting to further investigate the connection of Sinkhorn GAT and GCN, as both architectures are concerned with normalizing the adjacency matrix row and column wise (i.e. for the sending and receiving nodes). It might also be interesting to investigate a combination of AOHG and IOHG. Moreover, it might be possible to include some optimizations mentioned in this report to improve the PyG framework. Finally, one of the most important ideas for future work is the mathematical fundament for all the architectures and ideas proposed here. This includes convergence proves, mathematical bounds for the power of representation of the IOHG and AOHG. (Maybe a GNN universal function approximator theorem?).

References

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

G. Bouritsas, F. Frasca, S. Zafeiriou, and M. Bronstein. *Improving Graph Neural Network Expressivity via Subgraph Isomorphism Counting*. June 2020.

D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams. Convolutional Networks on Graphs for Learning Molecular Fingerprints. In *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL `https://papers.nips.cc/paper/2015/hash/f9be311e65d81a9ad8150a60844bb94c-Abstract.html`.

M. Fey and J. E. Lenssen. Fast Graph Representation Learning with PyTorch Geometric, May 2019. URL `https://github.com/pyg-team/pytorch_geometric`. original-date: 2017-10-06T16:03:03Z.

J. Gilmer, S. Schoenholz, P. Riley, O. Vinyals, and G. Dahl. Neural Message Passing for Quantum Chemistry. Apr. 2017.

K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4 (2):251–257, 1991. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(91)90009-T. URL `https://www.sciencedirect.com/science/article/pii/089360809190009T`.

T. N. Kipf and M. Welling. Semi-Supervised Classification with Graph Convolutional Networks. Nov. 2016. URL `https://openreview.net/forum?id=SJU4ayYgl`.

A. Mayr, G. Klambauer, T. Unterthiner, and S. Hochreiter. DeepTox: Toxicity Prediction Using Deep Learning. *Frontiers in Environmental Science*, 3, Dec. 2015. doi: 10.3389/fenvs.2015.00080.

A. Mayr, G. Klambauer, T. Unterthiner, M. Steijaert, J. Wegner, H. Ceulemans, D.-A. Clevert, and S. Hochreiter. Large-scale comparison of machine learning methods for drug target prediction on ChEMBL. *Chemical Science*, 9, June 2018. doi: 10.1039/C8SC00148K.

D. Mesquita, A. Souza, and S. Kaski. *Rethinking pooling in graph neural networks*. Oct. 2020.

C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe. Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):4602–4609, July 2019. ISSN 2374-3468. doi: 10.1609/aaai.v33i01.33014602. URL `https://ojs.aaai.org/index.php/AAAI/article/view/4384`.

C. Morris, N. M. Kriege, F. Bause, K. Kersting, P. Mutzel, and M. Neumann. TUDataset: A collection of benchmark datasets for learning with graphs. *arXiv:2007.08663 [cs, stat]*, July 2020. URL `http://arxiv.org/abs/2007.08663`. arXiv: 2007.08663.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox,

and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

N. Shervashidze, P. Schweitzer, E. Jan, V. Leeuwen, K. Mehlhorn, and K. Borgwardt. Weisfeiler-Lehman Graph Kernels. *Journal of Machine Learning Research*, 1:1–48, Jan. 2010.

R. Sinkhorn and P. Knopp. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific Journal of Mathematics*, 21(2):343–348, 1967.

P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph Attention Networks. In *International Conference on Learning Representations*, Feb. 2018. URL https://openreview.net/forum?id=rJXMpikCZ.

B. Weisfeiler and A. Lehman. THE REDUCTION OF A GRAPH TO CANONICAL FORM AND THE ALGEBRA WHICH APPEARS THEREIN. page 11, 1968.

K. Xu*, W. Hu*, J. Leskovec, and S. Jegelka. How Powerful are Graph Neural Networks? In *International Conference on Learning Representations*, Sept. 2018. URL https://openreview.net/forum?id=ryGs6iA5Km.

J. You, J. M. Gomes-Selman, R. Ying, and J. Leskovec. Identity-aware Graph Neural Networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(12):10737–10745, May 2021. ISSN 2374-3468. doi: 10.1609/aaai.v35i12.17283. URL https://ojs.aaai.org/index.php/AAAI/article/view/17283.

Z. Zhang, J. Bu, M. Ester, J. Zhang, Z. Li, C. Yao, D. Huifen, Z. Yu, and C. Wang. Hierarchical Multi-View Graph Pooling with Structure Learning. *IEEE Transactions on Knowledge and Data Engineering*, PP:1–1, June 2021. doi: 10.1109/TKDE.2021.3090664.

J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020. ISSN 2666-6510. doi: https://doi.org/10.1016/j.aiopen.2021.01.001. URL https://www.sciencedirect.com/science/article/pii/S2666651021000012.

A Further GNN topics

In the last weeks of my internship I developed an interest in current approaches for graph pooling, as graph pooling can have many beneficial properties, of which some can also be used to overcome the upper bound of expressivity that the WL isomorphism test sets. In particular the papers I looked into are "Rethinking pooling in graph neural networks" by [Mesquita et al., 2020], "Hierarchical Multi-View Graph Pooling with Structure Learning" by [Zhang et al., 2021] and finally "Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks" by [Morris et al., 2019] which I already used for my initial research with the WL bounds for representation of GNNs.

Some possibly good properties or applications of local graph pooling are the ability to do computations on sub-graphs more efficiently, the benefits of a larger receptive field, potentially overcoming the bounds on expressivity that are set by the WL isomorphism test, Generative Graph Neural Networks, partial propagation or distributed partial propagation for multi GPU for very large graphs.

One of the mayor above mentioned ideas is the generative graph neural networks which could also be useful for the domain of drug discovery. E.g. when wanting to find new molecules with a particular property (e.g. biochemical reaction in the human body) instead of having a GNN which classifies for random input graphs if they have those properties, one could just generate new graphs with this property in the first place.