Providing Bipartite GNN Explanations with PGExplainer

Tristan Marten Lewin Schulz

Bachelor of Science

May 19, 2025

Supervisors:
Prof. Dr. Stefan Harmeling
Lukas Schneider

Artificial Intelligence (VIII)
Department of Computer Science
TU Dortmund University

Abstract

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

Contents

List of Figures

List of Tables

List of variables

X A discrete random variable

$p(x)$ The probability mass function of X

$H(X)$ The entropy of the random variable X

$\mathbf{x} \in \mathbb{R}^n$ A feature vector of n features TODO: USE d??

$\mathbf{w} \in \mathbb{R}^n$ A weight vector of n weights

$\mathbf{c} \in \mathbb{R}^m$ A vector of m bias terms

$\mathbf{W} \in \mathbb{R}^m \times n$ A weight matrix of m units and n features

$\boldsymbol{\theta}$ A set of model parameters

$\mathbf{h} \in \mathbb{R}^n$ A vector of n hidden unit activations or hidden features

$b \in \mathbb{R}$ A bias scalar

$\Omega(\boldsymbol{\theta})$ A regularization term (e.g., an L_2-norm penalty on the parameters)

$\mathcal{B} \in \mathbb{R}^{m \times n}$ A minibatch design matrix of activations

$\mathcal{B}' \in \mathbb{R}^{m \times n}$ The normalized version of B

$\boldsymbol{\mu} \in \mathbb{R}^m$ A vector containing the mean of each unit

$\boldsymbol{\sigma} \in \mathbb{R}^m$ A vector containing the standard deviation of each unit

s The expectation of a function $f(x)$ under the distribution $p(x)$

$\hat{s}_N$ The Monte Carlo estimate of the expectation, based on N samples.

Chapter 1

Introduction

SAT motivation: Advancements in deep learning SAT solving, e.g. NeuroSAT. Need for evaluation of these models! use of GNN explainers, since SAT can be reduced to graph domain. GOAL: Application of PGExplainer on NeruoSAT to generate explanations. Explanations need gt to be evaluated on accuracy. Use concepts like unsat cores as gt and compare to explanations provided by PGExplainer to see whether NeuroSAT explanations align with "human-observable" principles. Concrete: Graph task: Prediction of UNSAT and unsat cores as gt. Node task: Difficult with NeuroSAT?

Therefore explanation and replication of PGExplainer. After successful replication, application on NeuroSAT.

PGExplainer claims to be model-agnostic - Works for any GNN. Additionally, able to generate explanations in an inductive setting, which is what we want to do.

Since our goal is application on a bipartite problem, reimplementation of PGExplainer that changes architecture of target GNN to a PyG layer that works on bipartite graphs and also allows passing of edge weights (required for PGE).

Pytorch reimplemenations already exist, eg. Replication study RE-PGE that focuses on orginal code to replicate results. We follow paper as close as possible, and also conduct the code to try different settings/hyperparameters for our changed GNN - OUR FOCUS: Evaluate whether PGExplainer still applicable for our slightly changed GNN model!

Therefore, use PGExplainer and Repliaction paper as baselines to compare our results (INDUCTIVE SETTING!)

Lastly, apply PGExplainer NeuroSAT, a solver for bipartite graph problem SAT inductively
Topic and context: Explainability in GNNs, show importance, examples,
Focus and scope: Focus on PGExplainer that allows inductive explanations + is model agnostic

Relevance and importance: Context to SAT? Need for explanations of the predictions made by NNs
Questions and objectives: Generate explanations and compare to UNSAT cores?

Chapter 2

Theoretical Background

In this chapter we define the necessary background for understanding PGExplainer as well as the follow-up work regarding its application on a machine learning solver for the Boolean Satisfiability Problem (SAT). We start by giving an introduction to deep learning in Section 2.1, including a specific architecture and regularization techniques. In Section 2.2 we define the necessary graph theory, including bipartite graphs and random graphs. To understand the motivation behind the objective of PGExplainer we outline the relevant concepts of information theory, namely entropy, cross-entropy and mutual information, in Section 2.3. Since the explainer model operates on Graph Neural networks, we introduce the historically relevant GNN model by Scarselli et al. [**4700287**] in Section 2.4, as well as two succeeding models. Lastly, in Section 2.5 we describe the boolean satisfiability problem that motivates this work.

2.1 Deep learning

In this chapter we introduce Deep Learning (DL) in the context of Machine Learning (ML) and their concepts required for this work. The definitions in this chapter loosely follow Goodfellow et al. [**Goodfellow-et-al-2016**].

ML algorithms generally learn to perform certain tasks from data that can broadly be divided into supervised and unsupervised learning. In this work, we focus on supervised learning, meaning that the algorithm learns from a dataset containing both features and labels or targets that the algorithm is supposed to predict. In other words, the algorithm learns from a training set of input examples, defined as vectors $\mathbf{x} \in \mathbb{R}^n$, with entries x_i denoting features, and output examples $\mathbf{y}$, whose shape depends on the task at hand. Its goal is to be able to associate unseen inputs from a set of test data, coming from the same distribution as the training data, with some target output.

This process estimates the generalization error of the learner after the training is complete. Common supervised ML tasks include classification, assigning an input to one of

k categories, and regression, where the program shall predict a numeric value given some input. TODO: WHAT IS A MODEL? WHAT IS A LAYER?

DL entails expanding the size of the model used in our algorithm to allow for representations of functions with increasing complexity. This enables many human-solvable tasks that consist of mapping an input vector to an output vector to be performed with DL, given sufficiently large models and datasets. Most notably, Krizhevsky et al. [**krizhevsky2012imagenet**] achieved record-breaking results by using a large, deep convolutional network for image classification.
However, increasing the size of a model comes with new challenges. Since models become more complex with increasing number of layers, subnetworks and parameters, they are commonly treated as a "black box", as it is difficult to explain their decision-making process [**noor2024survey**]. Thus, the need for methods that can explain and interpret such models arises. Furthermore, the increasing number of parameters directly raises the computational complexity, resulting in longer training times and higher hardware requirements. This may also impose limitations when deploying the model on low-resourced end-devices.
TODO: Formally define objective? What exactly does it do? In many cases DL involves optimizing an objective function, usually by minimizing $f(x)$, that guides the model. This objective function is also referred to as loss function in the context of minimization. To minimize a function $y = f(x)$, where $y, x \in \mathbb{R}$, we make use of the derivative $f'(x)$ that tells us how to change x in order to get an improvement in y. We can reduce $f(x)$ by moving x in the direction opposite of the sign of its derivative, since $f(x - \epsilon \text{ sign}(f'(x))) < f(x)$ for small enough ϵ. Since we usually want to minimize a function with multiple inputs, we let $\mathbf{x} \in \mathbb{R}^n$ be a vector with n elements. The partial derivative $\frac{\partial}{\partial x_i} f(\mathbf{x})$ is utilized, telling us how f changes when only x_i increases at point $\mathbf{x}$. To generalize this to a vector, the gradient $\nabla_x f(\mathbf{x})$ of f is defined as the vector containing all partial derivatives. The aforementioned technique used to minimize a function is hence called gradient descent (see Cauchy[**cauchy1847methode**]).
Ultimately, a DL algorithm typically consists of the following components: a dataset, an objective function, an optimization procedure like gradient descent, and a model.

We generally control an ML algorithm by tuning the hyperparameters, which are settings used to control its behavior that are not adapted by the algorithm itself. Settings that are difficult to optimize may be selected as hyperparameters. Common examples are the number of epochs - the iterations over the full training data - and the learning rate, that controls how much the model adjusts the weights at each step. An additional goal is finding the combination of hyperparameter settings that leads to the best performance of our model. This may also be automated with searching or learning algorithms.
To tune the hyperparameters we need a validation set of data that is not directly observed by the training algorithm, since the separate test data may not be used for making decisions

about our model. This set estimates the generalization error during training. The training data is therefore commonly split into two disjoint sets, the training set and the validation set, with a standard split being 80/20.

A traditional approach for tuning hyperparameters is the grid search, as described in Liashchynskyi and Liashchynskyi [**liashchynskyi2019grid**]. This method involves defining a subspace of the complete hyperparameter space and searching for the best-performing training algorithm across all possible combinations within that subspace.

2.1.1 Multilayer Perceptron

A classical DL model is the multilayer perceptron (MLP), first proposed by Rosenblatt [**rosenblatt1958perceptron**], with the general goal of approximating a function f^*. In the case of classification we could define a function $y = f^*(\mathbf{x})$ that maps an input vector $\mathbf{x}$ to a label y. The MLP then defines the mapping $y = f(\mathbf{x}; \boldsymbol{\theta})$ and learns the value of the parameters $\boldsymbol{\theta}$ that best approximate the function. At the start of the training these parameters are initialized randomly or by a smart initialization strategy, like the Glorot/Xavier initialization [**glorot2010understanding**].

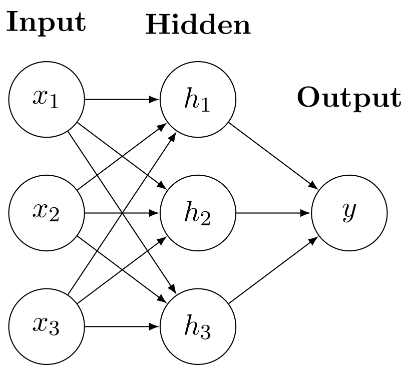


Figure 2.1: A simple MLP with one hidden layer.

These models are also referred to as feedforward neural networks, as they process information from $\mathbf{x}$, through the intermediate computations that define f, to the output y without feedback connections that would feed outputs back to itself. The name network is derived from their representation as a composition of multiple different functions, that are described by a directed acyclic graph. An example network is $f(\mathbf{x}) = f^{(2)}(f^{(1)}(\mathbf{x}))$, which processes an input vector $\mathbf{x}$ through a hidden layer $f^{(1)}$, and an output layer $f^{(2)}$ (see Figure 2.1). The length of this chain of functions defines the depth of a model, coining the term ”deep learning”.

The approximation is achieved by training our network with training data, that consists of approximated examples of $f^*(\mathbf{x})$ at different points in the training and labels $y \approx f^*(\mathbf{x})$. These training examples dictate the output layer to generate a value close to y for each input $\mathbf{x}$, or to produce a higher-level representation that can be used for subsequent tasks. The learning algorithm then learns to utilize the other hidden layers, without specified behaviors, to achieve the best approximation. It is to note that the hidden layers are vector-valued with each vector element, referred to as unit, loosely taking the role of a

neuron in neuroscience. This comparison is a common analogy to convey the idea of information processing. Models are therefore also referred to as neural networks.
An input or intermediate layer for our model could be defined as:

$$f^{(i)}(\mathbf{x}; \mathbf{W}, \mathbf{c}) = \mathbf{W}^\top \mathbf{x} + \mathbf{c}, \tag{2.1}$$

where $\mathbf{W} \in \mathbb{R}^{m \times n}$ contains the weight matrix for m units with n features and $\mathbf{c} \in \mathbb{R}^m$ is a vector of bias terms. To keep the model from strictly learning a linear function of its inputs we can calculate the vector of hidden units $\mathbf{h}$ by applying an activation function g to the output of the linear layer:

$$\mathbf{h} = g(f^{(i)}(\mathbf{x}; \mathbf{W}, \mathbf{c})). \tag{2.2}$$

The activation function g is typically applied element-wise to describe the features of a hidden layer:

$$h_i = g(\mathbf{x}^\top \mathbf{W}_{:,i} + c_i). \tag{2.3}$$

The hidden units $\mathbf{h}$ can then serve as input to the next hidden layer or directly to the output layer.
An example for a linear output layer that calculates a scalar for an input vector $\mathbf{h}$ is

$$f^{(i+1)}(\mathbf{h}; \mathbf{w}, b) = \mathbf{h}^\top \mathbf{w} + b, \tag{2.4}$$

with weight vector $\mathbf{w} \in \mathbb{R}^n$ and bias $b \in \mathbb{R}$ as parameters. This would define our model consisting of the two defined layers as $f(x; \boldsymbol{\theta})$ with parameters $\boldsymbol{\theta} = (\mathbf{W}, \mathbf{c}, \mathbf{w}, b)$. The learning algorithm would thus adapt the parameters $\boldsymbol{\theta}$ to approximate the target function as close as possible.
The activation function typically maps its input to a real number within a specific range, often between 0 and 1, conceptually imitating the activation of a biological neuron. We try to use activation functions that are continuous differentiable and easily calculated, to minimize computational complexity (see Liu et al. [**Liu2020**]). Examples are the Sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{2.5}$$

where e denotes the exponential function, and the Rectified Linear Unit (ReLU):

$$ReLU(x) = \begin{cases} 0 & x \leq 0, \\ 1 & x > 0. \end{cases} \tag{2.6}$$

Backpropagation

The back propagation algorithm is commonly used during neural network training. It optimizes the network parameters by leveraging gradient descent. It first calculates the values for each unit in the network given the input and a set of parameters in a forward order. Then the error for each variable to be optimized is calculated, and the parameters are updated according to their corresponding partial derivative backwards. These two steps will repeat until reaching the optimization target.

2.1.2 Regularization

A central problem of not only deep learning but machine learning in general is the creation of an algorithm that generalizes well, therefore performing not only on training data, but also on unseen test inputs [**Goodfellow-et-al-2016**]. There exists many strategies that aim to reduce the test error, sometimes at the expense of an increase in training error. These strategies are known as regularization. We introduce a few of these strategies that are relevant in the course of this work.

Parameter Norm Penalties

TODO: DEFINE OBJECTIVE J BEFOREHAND? $\mathbf{X}$ IS MATRIX OF (TRAINING-)INPUTS Many regularization approaches in deep learning restrict the capacity of a model by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function. Let J be the objective function and the regularized objective function

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha\Omega(\boldsymbol{\theta}), \tag{2.7}$$

where $\alpha \in [0, \infty)$ is a hyperparameter used to weigh the relative contribution of the norm penalty term $\Omega(\boldsymbol{\theta})$. $\alpha = 0$ results in no regularization, while larger values increase the regularization effect. During training the algorithm will not only minimize the objective function J, but also the regularization measure of the parameters $\boldsymbol{\theta}$, or a subset of these. We usually only regularize the weights $\mathbf{w}$ with Ω, since these specify the interaction of two variables, rather than the bias b that only controls a singular variable.
Inherently, different parameter norms Ω can result in different preferred solutions. One common example is the L^2 regularization, known as weight decay. It drives the weights closer to the origin by adding the term

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2}||\mathbf{w}||_2^2 = \frac{1}{2}\mathbf{w}^\top\mathbf{w}, \tag{2.8}$$

with $||\mathbf{x}||_p = (\sum_i |x_i|^p)^{\frac{1}{p}}$ denoting the L^p norm or size of a vector $\mathbf{x}$. Specifically, the L^2 norm denotes the Euclidean distance from the origin to the point $\mathbf{x}$, and the size of a vector $\mathbf{x}$ if squared: $||\mathbf{w}||_2^2 = \mathbf{x}^\top\mathbf{x}$.
Another option is the L^1 regularization, defined as :

$$\Omega(\boldsymbol{\theta}) = ||\mathbf{x}||_1 = \sum_i |\mathbf{x}_i|, \tag{2.9}$$

that provides a more sparse solution in comparison, meaning that some parameters have an optimal value of zero.

Norm Penalties as Constrained Optimization

TODO: Constraints in general, as applied in my work probably needed here

Early stopping

TODO: SOURCE?! When training a deep model, a commonly observable problem is the training error steadily decreasing over time, but the validation set error rising after some time, indicating that the model is overfitting to the train data. To obtain a better model we may then return to a parameter setting at an earlier point in the training, where the validation set error was at its lowest. In practice, we store a copy of the model parameters at any point in training where the validation error decreases. Thus, when the training is concluded we return the stored set of parameters, rather than the latest one. Additionally, we may stop our algorithm early if the validation error does not decrease over a set period of iterations.

Dropout

Dropout [**hinton2012improving**] is a regularization technique used to prevent a neural network from "overfitting" to the training data, which occurs due to feature detectors (TODO!?) of models being tuned to the training data, but not to the unseen test data. This is done by randomly omitting each hidden unit from the network with a common probability p for each presentation of each training instance, to avoid hidden units relying on the presence of other hidden units. Usually, to reduce the error on a test set, an average of the predictions of multiple networks is consulted, which requires training many separate networks. Random dropout provides a computationally cheap alternative to this approach. Effectively, at each presentation of each training instance a different network is used, but the present hidden units share the same weights across all these networks.
At test time, the "mean network" is used, that contains all hidden units with their outgoing weights scaled by the factor $\frac{1}{1-p}$, to account for more hidden units being present during this stage.

Batch Normalization

While not strictly a regularization technique, batch normalization [**ioffe2015batch**] often serves a similar purpose in practice by improving generalization. In deep models composed of several layers the gradient tells us how to update each parameter assuming that the other layers do not change. Since in practice all layers are updated simultaneously, this may lead to unexpected results. Batch normalization is an optimization technique that can be applied to any input or hidden layer to reduce the aforementioned problem. We follow the definition by Goodfellow et al. [**Goodfellow-et-al-2016**].
Let $\mathcal{B}$ be a mini-batch of activations of the layer to normalize, in the form of a design matrix with rows of activations. To normalize $\mathcal{B}$, it is replaced by

$$\mathcal{B}' = \frac{\mathcal{B} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}, \tag{2.10}$$

where $\boldsymbol{\mu}$ is a vector containing the mean of each unit and $\boldsymbol{\sigma}$ is a vector containing the standard deviation of each unit. Each activation is normalized individually with the $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ corresponding to its row. The network processes $\mathcal{B}'$ as functionally equivalent to $\mathcal{B}$. During training,

$$\boldsymbol{\mu} = \frac{1}{m} \sum_i \mathcal{B}_i \tag{2.11}$$

and

$$\boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{m} \sum_i (\mathcal{B}_i - \boldsymbol{\mu})^2}, \tag{2.12}$$

where δ is a small positive value introduced to avoid $\sqrt{z}$ at $z = 0$ and m is the number of elements in the batch. It is important to note that back propagation is performed through all three operations, to prevent the gradient from proposing an operation that solely increases the mean and standard deviation of the output of a layer. Batch normalization therefore reparameterizes the model to include some units that are standardized by definition.
When testing the model, we replace $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ with running averages that were collected during training, to enable the evaluation of individual examples outside mini-batches.

2.1.3 Monte Carlo Sampling

In order to best approximate a randomized algorithm we can make use of Monte Carlo methods as described in Goodfellow et al. [**Goodfellow-et-al-2016**]. A common practice in ML is to draw samples from a probability distribution and using these to form a Monte Carlo estimate of some quantity. This can be used to train a model that can then sample from a probability distribution itself.
More specifically, the idea of Monte Carlo sampling is to view a sum of function evaluations, such as those of a loss function, as if it was an expectation under some probability distribution. This estimate is approximated with a corresponding average. Let

$$s = \sum_x p(x) f(x) = \mathbb{E}_p[f(x)] \tag{2.13}$$

be the sum to estimate with p being a probability distribution over a random variable x and $\mathbb{E}$ denoting the expectation. Then s can be approximated by drawing n samples from p and constructing the empirical average

$$\hat{s}_n = \frac{1}{n} \sum_{i=1}^{n} f(x^{(i)}). \tag{2.14}$$

2.2 Graph Theory

These definitions loosely follow Liu and Zhou [**Liu2020**]. A graph is a data structure consisting of a set of nodes that are connected via edges, modeling objects and their relationships. It can be represented as $G = (V, E)$ with $V = \{v_1, v_2...v_n\}$ being the set of n nodes, and $E \in V \times V$ the set of edges. We adopt the conventions from Diestel [**Diestel2017**] to refer to the node and edges set of any graph G with $V(G)$ and $E(G)$ respectively, regardless of the actual names of the sets, as well as a referring to G with node set V as "G on V".

An edge $e = (u, v)$, sometimes accompanied by an edge weight $e_{u,v} \in (0, 1)$, connects nodes u and v, making them neighbors. The neighborhood $\mathcal{N}(u)$ of node u in $V(G)$ is defined as $\mathcal{N}(u) = \{v \in V(G) \mid (v, u) \in E(G)\}$. We denote the set of edges that are incident to u as $E(u) = \{(v, u) \in E(G) \mid v \in \mathcal{N}(u)\}$. Additionally, we define the k-hop neighborhood of node u in $V(G)$ as $\mathcal{N}_k(u) = \{v \in V(G) \mid dis_G(v, u) \leq k\}$, where $dis_G(v, u)$ denotes the distance of nodes v, u in G, defined as the length of the shortest path between the two nodes. Edges are either directed or undirected and lead to directed or undirected graphs if exclusively present.

The degree of a node $v \in V(G)$ is the number of edges connected to v and denoted by $d(v) = |\{(u, v) \in E(G) \mid u \in V(G)\}|$. G can be described by an adjacency matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, where

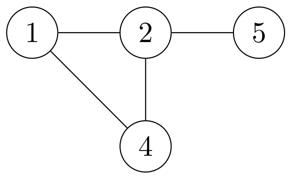
$$\mathbf{A}_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \in E \text{ and } i \neq j, \\ 0 & \text{otherwise.} \end{cases}$$

If G is an undirected Graph the adjacency matrix will be symmetrical, as seen in Figure 2.2. The diagonal degree matrix $D \in \mathbb{R}^{n \times n}$ of G is defined as:

$$D_{ii} = d(v_i).$$

G is called a subgraph of another graph $G' = (V', E')$ if $V(G) \subseteq V(G')$ and $E(G) \subseteq E(G')$. This is denoted as $G' \subseteq G$. The number of nodes $|V|$ in a graph is called its order and the number of edges $|E|$ is its size (see Diestel [**Diestel2017**]).
We additionally define bipartite graphs according to Asratian et al. [**asratian1998**]: A graph G is bipartite if the set of nodes V can be partitioned into two disjoint sets V_1 and V_2 such that: $V = V_1 \cup V_2, V_1 \cap V_2 = \emptyset$ and $\forall (i, j) \in E(G) : (i \in V_1 \wedge j \in V_2) \vee (i \in V_2 \wedge j \in V_1)$. This formalizes that no two nodes from the same set are adjacent. The sets V_1 and V_2 are called color classes and (V_1, V_2) is a bipartition of G. This means that if a graph is bipartite all nodes in V can be colored by at most two colors so that no two adjacent nodes share the same color. This is visualized in Figure 2.3.

Random Graphs
Gilbert [**gilbert1959random**] describes the process of generating a random graph of order n by assigning a common probability of existence to each potential edge between any two nodes. For each of these potential edges an experiment is performed independently to determine whether it shall be included in the resulting graph. Note that this process can be modeled using a Bernoulli distribution.



$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Figure 2.2: A simple undirected graph G on V, with $V = \{1, 2, 4, 5\}$ and $E(G) = \{\{1, 2\}, \{2, 4\}, \{1, 4\}, \{2, 5\}\}$ (left), and its adjacency matrix $\mathbf{A}$ (right).

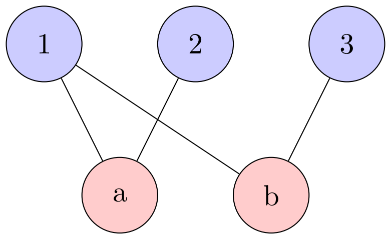


Figure 2.3: A bipartite graph with color classes $V_1 = \{1, 2, 3\}$ (blue) and $V_2 = \{a, b\}$ (red).

A random graph is further described by Diestel [**Diestel2017**] as follows. Let $V = \{0, ..., n-1\}$ be a fixed set of n elements. Say we want to define the set $\mathcal{G}$ of all graphs on V as a probability space, which allows us to ask whether a Graph $G \in \mathcal{G}$ has a certain property. To generate our random graph we then decide from some random experiment whether e shall be an edge of G for each potential $e \in V \times V$. The probability of success - accepting e as edge in G - is defined as $p \in [0, 1]$ for each experiment. This leads to the probability of G being a particular graph G_0 on V with e.g. m edges being equal to $p^m q^{\binom{n}{2}-m}$ with $q := 1 - p$.

2.3 Information Theory

To fully understand the learning objective of PGExplainer it is necessary to define the concepts of entropy and mutual information. We follow the definitions by Cover and Thomas [**Cover2005**] if not stated otherwise.

2.3.1 Entropy

Entropy is used to describe the uncertainty of a random variable. It quantifies the average amount of information produced by the outcomes of said variable. This is commonly illustrated as the average number of bits needed to encode its values if optimal code is used. Let X be a discrete random variable with alphabet $\mathcal{X}$ and probability mass function $p(x) = Pr\{X = x\}$ for $x \in \mathcal{X}$. TODO: EINHEITLICHE DEFINITION! SIMPLY USE

THIS DEFINITION IN MONTE CARLO SAMPLING? The entropy $H(X)$, also written as $H(p)$, is defined as

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log p(x). \tag{2.15}$$

The log is to the base e and entropy is measured in natural units of information (nats) in our case. We will use the convention from Cover and Thomas [**Cover2005**] that $0 \log 0 = 0$, as terms of zero probability do not change the entropy.
The conditional entropy of Y given X is defined as the expected value of the entropies of the conditional distributions, averaged over the conditioning random variable. If $(X, Y) \sim p(x, y)$ for a pair of discrete random variables (X, Y) with joint distribution $p(x, y)$, the conditional entropy is defined as

$$\begin{aligned} H(Y|X) &= -\sum_{x \in \mathcal{X}} p(x) H(Y|X = x) && (2.16) \\ &= -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log p(y|x) && (2.17) \\ &= -\mathbb{E} \log p(Y|X). && (2.18) \end{aligned}$$

2.3.2 Relative Entropy and Cross-Entropy

The relative entropy between two distributions is a measure of "distance" between the two. It measures the inefficiency of assuming a distribution to be q when the true distribution is p. Note that it is not a true measure of distance as it is not symmetrical. The relative entropy takes a value of 0 only if $p = q$. We define the KL divergence or relative entropy between two probability mass functions $p(x), q(x)$ as

$$D_{KL}(p||q) = \sum_{x \in \mathcal{X}} p(x) \log \frac{p(x)}{q(x)}. \tag{2.19}$$

We use the convention from Cover and Thomas [**Cover2005**] that $0 \log \frac{0}{0} = 0$, $0 \log \frac{0}{q} = 0$ and $p \log \frac{p}{0} = \infty$. Suppose we know the true distribution p of our random variable. We could then construct a code with an average description length of $H(p)$. If we used the code for the distribution q instead, we would need $H(p) + D_{KL}(p||q)$ nats to describe the random variable on average. This is also referred to as the cross-entropy (see Goodfellow et al. [**Goodfellow-et-al-2016**]):

$$H(p, q) = H(p) + D_{KL}(p||q) \tag{2.20}$$

Since this is later applied in the PGExplainer (see Equation 4.6), we derive for the discrete case with mass probability functions p, q defined on the same support $\mathcal{X}$:

$$\begin{aligned} H(p,q) = H(p) + D_{KL}(p||q) &= -\sum_{x\in\mathcal{X}} p(x)\log p(x) + \sum_{x\in\mathcal{X}} p(x)\log\frac{p(x)}{q(x)} \\ &= -\sum_{x\in\mathcal{X}} p(x)\log p(x) + \sum_{x\in\mathcal{X}} p(x)\log p(x) - \sum_{x\in\mathcal{X}} p(x)\log q(x) \\ &= -\sum_{x\in\mathcal{X}} p(x)\log q(x) \end{aligned} \tag{2.21}$$

2.3.3 Mutual Information

A closely related concept is mutual information. It measures the amount of information that one random variable contains about another or the reduction in uncertainty of said variable due to knowing the other. A high mutual information therefore implies that the information of one variable can be gathered from the other.
Let X and Y be two random variables with the joint probability mass function $p(x, y)$ and marginal probability mass functions $p(x)$ and $p(y)$. Mutual information $I(X; Y)$ is the relative entropy between the joint distribution and the product distribution $p(x)p(y)$:

$$I(X;Y) = \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log\frac{p(x,y)}{p(x)p(y)} \tag{2.22}$$

$$= H(X) - H(X|Y) \tag{2.23}$$

2.4 Graph Neural Networks

Graph Neural Networks(GNNs) [**4700287**] are a DL-based approach that operates on graphs. Due to their unique non-Euclidean property they find usage in many areas, including node classification [**gao2019graph**], graph classification [**xu2018powerful**] and link prediction tasks [**zhang2018link**]. Their high interpretability and strong performance have led to GNNs becoming a commonly employed method in graph analysis. They combine the key features of convolutional neural networks [**726791**], such as local connection, shared weights, and multi-layer usage, with the concept of graph embeddings [**cai2018comprehensive**] to leverage the power of feature extraction and representation as low-dimensional vectors for graphs (see Liu and Zhou [**Liu2020**]).

TODO: CONCRETELY DEFINE NODE AND GRAPH TASKS! VISUALIZE? Graphs are a common way of representing data in many different fields, including ML. ML applications on graphs can mostly be divided into graph-focused tasks and node-focused tasks. For graph-focused applications our model does not consider specific singular nodes, but rather implements a classifier on complete graphs. In node-focused applications however the model is dependent on specific nodes, leading to classification tasks that rely on the properties of each node. The study of GNNs was first introduced in [**gori2005new**]

and refined in [**4700287**]. Thus, we describe the supervised GNN model by Scarselli et al. [**4700287**] that aims to preserve the important, structural information of graphs by encoding their topological relationships among nodes.
A node is naturally defined by its features as well as its related notes in the graph. The goal of a GNN is to learn state embeddings $\mathbf{h}_v \in \mathbb{R}^n$ for each node v, that map the neighborhood of a node into a representation. These embeddings are used to obtain outputs $\mathbf{o}_v$, that e.g. may contain the distribution of a predicted node label. The GNN model proposed by Scarselli et al. [**4700287**] uses undirected homogeneous graphs with $\mathbf{x}_v$ describing the d-dimensional features of each node and x_e the optional features of each edge. The model updates the node states according to the input neighborhood with a local transition function f that is shared by all nodes. Additionally, the local output function g is used to produce the output of each node. $\mathbf{h}_v$ and $\mathbf{o}_v$ are therefore defined as

$$\mathbf{h}_v = f(\mathbf{x}_v, \mathbf{x}_{E(v)}, \mathbf{h}_{\mathcal{N}(v)}, \mathbf{x}_{\mathcal{N}(v)}), \tag{2.24}$$

$$\mathbf{o}_v = g(\mathbf{h}_v, \mathbf{x}_v), \tag{2.25}$$

with $\mathbf{x}$ denoting input features and $\mathbf{h}$ the hidden state. $\mathbf{x}_v, \mathbf{x}_{E(v)}, \mathbf{h}_{\mathcal{N}(v)}, \mathbf{x}_{\mathcal{N}(v)}$ denote the features of the node v and of its incident edges, as well as the states and features of its neighboring nodes, respectively. We define $\mathbf{H}, \mathbf{O}, \mathbf{X}$ and $\mathbf{X}_N$ as the matrices that are constructed by stacking all states, outputs, features, and node features, respectively. This allows us to define with the global transition function F and the global output function G, which are stacked versions of their local equivalent for all nodes in a graph:

$$\mathbf{H} = F(\mathbf{H}, \mathbf{X}), \tag{2.26}$$

$$\mathbf{O} = G(\mathbf{H}, \mathbf{X}_N). \tag{2.27}$$

Note that F is assumed to be a contraction map and the value of $\mathbf{H}$ is the fixed point of equation (2.26). To compute the state the iterative scheme

$$\mathbf{H}^{t+1} = F(\mathbf{H}^t, \mathbf{X}) \tag{2.28}$$

is used with $\mathbf{H}^t$ denoting iteration t of $\mathbf{H}$. The computations of f and g can be understood as the feedforward neural network.
To learn the parameters of this GNN, with target information t_v for a specific node v, the loss is defined as

$$loss = \sum_{i=1}^{p} (t_i - \mathbf{o},) \tag{2.29}$$

where p are the supervised nodes. A gradient-descent strategy is utilized in the learning algorithm, which consist of the following three steps: the states $\mathbf{h}_v^t$ are updated iteratively using equation (2.24) until time step T. We then obtain an approximate fixed point solution of equation (2.26): $\mathbf{H}(T) \approx \mathbf{H}$. For the next step the gradients of the weights W are calculated from the loss. Finally, the weights W are updated according to the computed

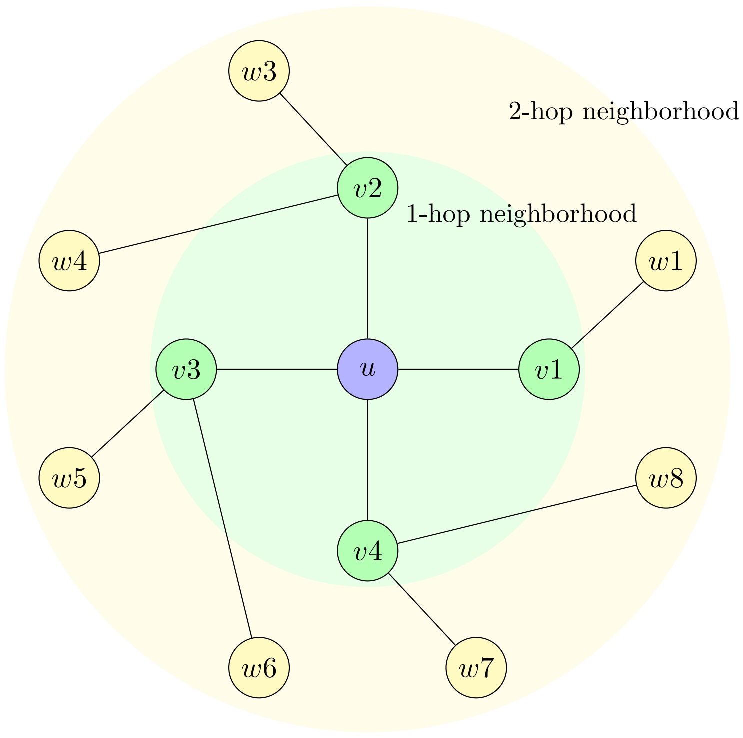


Figure 2.4: Visualization of the k-hop neighborhood of u.

gradient. This allows us to train a model for specific supervised or semi-supervised tasks, referred to as downstream task, and get hidden states of nodes in a graph.

Though the architecture proposed by Scarselli et al. [**4700287**] proved to be powerful for modeling structural data, this initial approach suffers from a few limitations. Most notably, it uses the same parameters in the iteration, while nowadays it is common practice to use different parameters in different layers. Stacking k GNN layers allows each node to aggregate information from nodes within its k-hop neighborhood, represented in Figure 2.4, seeking an increase in performance. It is important to note that this approach may also increase the noisy information spread by the exponentially increasing neighborhood nodes [**Liu2020**].

Another drawback is the computational inefficiency, as hidden states have to be updated T times until reaching the fixed point. A relaxation of the fixed point assumption enables multi-layer GNNs to provide stable representations of the node and its neighborhood [**li2015gated**]. Additionally, this architecture is unable to model informative edge features, which limits its capacity to learn meaningful hidden representations for edges.

Therefore, we briefly present two subsequent models that are used in the course of this work, as well as a more general framework relevant in the context of NeuroSAT.

Graph Convolutional Network

Graph convolutional networks (GCNs) aim to generalize the convolution operation of convolutional neural networks (CNNs) to the graph domain. An example is the model proposed by Kipf and Welling [**kipf2016semi**] that introduces a simple, layer-wise propagation rule for multi-layer GCNs as

$$\mathbf{H}^{(l+1)} = \sigma(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}), \tag{2.30}$$

where $\tilde{\mathbf{A}} = \mathbf{A}+\mathbf{I}_N$ is the adjacency matrix of the undirected input graph G with added self-connections. $\mathbf{I}_N$ is the identity matrix, $\tilde{\mathbf{D}}$ is the degree matrix and $\mathbf{W}^{(l)}$ is a layer-specific trainable weight matrix. $\sigma(\cdot)$ denotes an activation function, such as ReLU. $\mathbf{H}^{(l)} \in \mathbb{R}^{n\times d}$ denotes the matrix of node activations in the l-th layer, where $\mathbf{H}^{(0)} = X$. Note that this definition differs from the iterative formulation in Equation 2.28, in the sense that each layer applies a different function $F^{(l)}$, rather than a shared function F. $\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}$ describes a normalization of the adjacency matrix, referred to as renormalization trick. This architecture aims to alleviate the problem of overfitting on local neighborhood structures for graphs with very wide node degree distributions (TODO: Source).

Higher-Order GNN
TODO: A basic GNN model can be implemented as follows (Hamilton, Ying, and Leskovec 2017b) Morris et al. [**morris2019weisfeiler**] propose an implementation for a GNN model consisting of stacked neural network layers, that each aggregate the local neighborhood information of a node and pass it to the next one. This network is defined specifically for graphs that can be partitioned into r color classes and therefore applicable to bipartite graphs. The new features of node i are then computed with:

$$\mathbf{x}_i^{(l)} = \sigma(\mathbf{W}_1^{(l)}\mathbf{x}_i^{(l-1)} + \mathbf{W}_2^{(l)} \cdot \sum_{j\in\mathcal{N}(i)} e_{j,i} \cdot \mathbf{x}_j^{(l-1)}), \tag{2.31}$$

where $W_1^{(l)}$ and $W_2^{(l)}$ are two layer-specific trainable weight matrices.
TODO: MORE MOTIVATION FOR THIS!

Message Passing Neural Network The Message Passing Neural Network (MPNN) proposed by Gilmer et al. [**gilmer2017neural**] seeks to unify GNN models, by providing a general framework for supervised learning on GNNs, since many by now existing models operate similarly. Typically, messages derived from node features are passed along edges of a graph to be able to learn meaningful graph and node representations that can be used for downstream tasks [**kipf2016semi**] [**4700287**] [**velivckovic2017graph**] [**xu2018powerful**]. Therefore, GNNs may be expressed as MPPNs, where the forward pass consist of a message passing phase and a readout phase. The message passing phase runs for T time steps and is defined by the message function M_t, as well as the node update function U_t. At each time step t, the hidden state $\mathbf{h}_v^{t+1}$ of a node v in the graph G

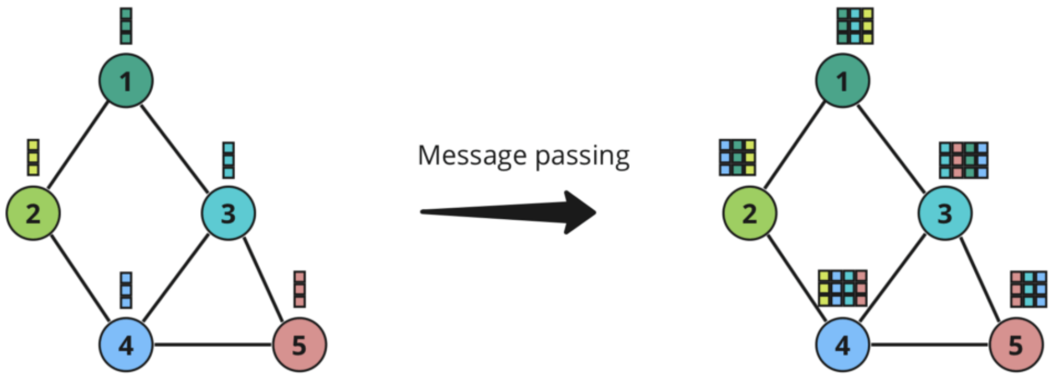


Figure 2.5: Node features passed along edges. Reprinted from [**ali2023gnns**].

is updated based on the message $\mathbf{m}_v^{t+1}$, which aggregates information from the previous hidden states of its neighbors according to the message function M_t. Formally, this is defined as

$$\mathbf{m}_v^{t+1} = \sum_{w \in \mathcal{N}(v)} M_t \left(\mathbf{h}_v^t, \mathbf{h}_w^t, \mathbf{e}_{v,w} \right) \tag{2.32}$$

$$\mathbf{h}_v^{t+1} = U_t \left(\mathbf{h}_v^t, \mathbf{m}_v^{t+1} \right), \tag{2.33}$$

where $\mathbf{e}_{v,w}$ denotes possible edge features of edge (v, w). In the readout phase a feature vector $\hat{\mathbf{y}}$ is computed for the entire graph from the final node states using a readout function R:

$$\hat{\mathbf{y}} = R \left(\{ \mathbf{h}_v^T \mid v \in G \} \right). \tag{2.34}$$

Note that the three functions M_t, U_t and R_t are learned and differentiable.
The higher-order GNN for example may also be defined with the following message and update functions:

$$M_t(\mathbf{h}_v^t, \mathbf{h}_w^t) = c_{v,w} \mathbf{h}_w^t, \tag{2.35}$$

$$U_v^t(\mathbf{h}_v^t, \mathbf{m}_w^{t+1}) = \mathrm{ReLU}(\mathbf{W}^t \mathbf{m}_v^{t+1}), \tag{2.36}$$

with $c_{v,w} = (d(v)d(w))^{-\frac{1}{2}} \mathbf{A}_{v,w}$.

2.5 Boolean Satisfiability Problem

We define the Boolean Satisfiability Problem (SAT) according to Guo et al. [**guo2023machine**]: A Boolean formula is constructed from Boolean variables, that only evaluate to True (1) or False (0), and the three logic operators: conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$). SAT aims to evaluate whether there exists a variable assignment for a formula constructed of said parts so that it evaluates to True. If so, the formula is said to be satisfiable or unsatisfiable otherwise. Every propositional formula can be converted into an equivalent formula in conjunctive normal form (CNF), which consists of a conjunction of one or more clauses. These clauses must contain only disjunctions of at least one literal - a variable or its negation. In this work we consider only formulae in CNF, as NeuroSAT [**selsam2018learning**] assumes SAT problems to be in CNF. An example of a satisfiable formula in CNF over the set of variables $V = \{x_1, x_2\}$ is

$$\psi(V) = (x_1) \land (\neg x_1 \lor x_2) \land (\neg x_2 \lor x_2)$$

with satisfying assignment $A : \{x_1 \mapsto 1, x_2 \mapsto 1\}$. Furthermore, SAT is NP-complete, meaning that if there exists a deterministic algorithm able to solve SAT in polynomial time, then such an algorithm exists for every NP problem (see Cook [**cook2023complexity**]). Current state-of-the-art SAT solvers apply searching based methods such as Conflict Driven Clause Learning [**marques1999grasp**] or Stochastic Local Search [**selman1993local**] with exponential worst-case complexity.

2.5.1 Representation as Bipartite Graph

SAT has extensively been studied in the form of graphs. Guo et al. [**guo2023machine**] describe four different types of graph representations for CNF formulae with varying complexity and information compression. Since we want to minimize the loss of information for SAT we adapt the information-richest form of a literal-clause graph (LCG). A LCG is a bipartite graph that separates literals and clauses, with edges connecting literals to the clauses they appear in (see Figure 2.6). The resulting graph G_b can formally be described by a biadjacency matrix $\mathbf{B}$ of shape $L \times C$, with (L, C) being a bipartition of G_b into literals and clauses.
Following Sun et al. [**articleBiadjacency**], let $\mathbf{A} \in \mathbb{R}^{(L+C)\times(L+C)}$ be the adjacency matrix of our bipartite graph. Since for the bipartite case edges exist only between the two color classes L and C, the adjacency matrix can be represented as

$$\mathbf{A}(i, j) = \begin{bmatrix} \mathbf{0}_{L\times L} & \mathbf{B} \\ \mathbf{B}^T & \mathbf{0}_{C\times C} \end{bmatrix}, \tag{2.37}$$

where $\mathbf{0}$ denotes a zero matrix in the shape of their subscript.

2.5.2 Unsatisfiable Cores

The core of an unsatisfiable formula in CNF is a subset of the formula that is also unsatisfiable. Every unsatisfiable formula therefore is a core on its own, but can be broken down into smaller cores. The smaller a core the more significance it holds. A minimal unsatisfiable core is also referred to as a minimal unsatisfiable subset (MUS). SAT solvers like

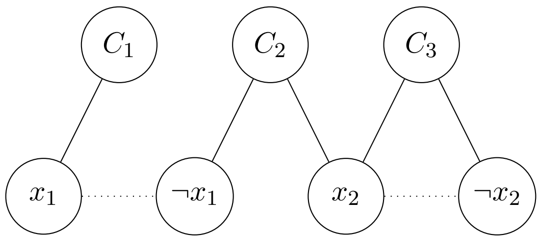


Figure 2.6: LCG representation of $\psi(V)$ with dashed lines representing the connection between complementary literals relevant for the message passing in GNNs.

MiniSat [**een2003extensible**] are able to compute unsatisfiable cores but do not generally provide a MUS due to high computational cost. However, several deletion-based algorithms exist for computing MUSs (see Torlak et al. [**10.1007/978-3-540-68237-0˙23**]).

Chapter 3

Related Work

Yuan et al. [**yuan2022explainability**] performed an extensive taxonomic survey on explainability in graph neural networks. We use this survey to discuss different approaches and motivate the selection of the PGExplainer for our work in Section 3.1. In Section 3.2 we briefly introduce important work related to the PGExplainer, as well as the model itself, since it is the core of our work. Lastly, we refer to NeuroSAT, which we use as a downstream model for the PGExplainer to generate explanations for its predictions on the SAT problem in Section 3.3.

3.1 Explainability in GNNs

Methods in DL have seen growth in performance in many tasks of artificial intelligence, including GNNs, since graphs are able to capture real-world data such as social networks or chemical molecules [**ying2018graph**], [**ma2021deep**]. However, the interpretability of these models is often limited due to their black-box design [**noor2024survey**]. Explainability methods aim to bypass this limitation by designing post-hoc techniques that provide insights into the decision-making process in the form of explanations. Such human-intelligible explanations are crucial for deploying models in real-world applications, especially when applied in interdisciplinary fields [**ribeiro2016should**].

There exist several different approaches for explaining predictions of deep graph models, that can be categorized into instance-level methods and model-level methods (see Yuan et al. [**yuan2022explainability**]). Instance-level methods aim to explain each input-graph by identifying important input features for its prediction, leading to input-dependent explanations. These can further be grouped by their importance score calculation into four branches. Gradients/feature-based methods use gradients as approximations of importance scores. Sensitivity Analysis [**baldassarre2019explainability**] is an example that directly uses squared values of gradients as importance scores of input features. This enables the scores to be calculated directly with back-propagation.

Perturbation-based approaches like GNNExplainer [**ying2019gnnexplainer**] and PGExplainer [**luo2020parameterized**] study the variation of the output with regard to different input perturbations. The intuition behind this is that when input information crucial to the prediction is kept, the new prediction should roughly align with the prediction from the original input. PGExplainer aims to improve the GNNExplainer by providing a way of generating explanations with a global understanding of the GNN, significantly improving the computational cost. Another approach is SubgraphX [**yuan2021explainability**], which utilizes Monte Carlo Tree search to generate subgraph-level explanations. This does however entail a higher computational cost.

Surrogate methods for deep graph models are inspired by surrogate methods for image data, that rely on neighboring areas of an input. Since graph data is concrete and contains topological information it is difficult to define the neighboring regions of an input graph. The idea is to obtain a local dataset containing neighboring data objects and predictions and fitting a simple, interpretable surrogate model to learn the local dataset. The explanations of the surrogate model are then regarded as the explanations of the original model. GraphLime [**huang2022graphlime**] is an example that considers the N-hop neighboring nodes of a target node as the local dataset, where N may be the number of GNN-layers. The weighted features of a non-linear surrogate model are then regarded as explanations. However, this method only explains node features, rather than the graph structure.

Decomposition methods, also motivated by success in the image domain, aim to measure input feature importance by decomposing the prediction into several terms, regarded as feature dependant importance score. Approaches for deep graph neural networks, like Layer-wise Relevance Propagation [**baldassarre2019explainability**], decompose the output prediction score to node importance scores. The decomposition rule is based on hidden features and weights, only enabling the study of node importance rather than graph structures.

Model-level methods, on the other hand, aim to explain GNNS without considering specific inputs, leading to input-independent, high-level explanations.

To fully trust the explanations provided by an explainer model, they must satisfy certain criteria, since there often is a mismatch between the optimizable metrics like accuracy and the actual metric of interest, which may not be measurable (see Ribeiro et al. [**ribeiro2016should**]). First and foremost, an explanation should be **interpretable** and therefore provide qualitative, human-understandable interpretations, that also consider the possibility of limited user knowledge. Additionally, **local fidelity** asserts that explanations should be faithful in a local context and consider the models' behavior in the vicinity of predicted instances. Explainers that treat the model to be explained as a black-box are **model-agnostic** and should therefore be able to explain any model. Lastly, a **global perspective** is needed to explain a model fully, allowing us to take sample explanations of individual predictions that serve as representation of the model.

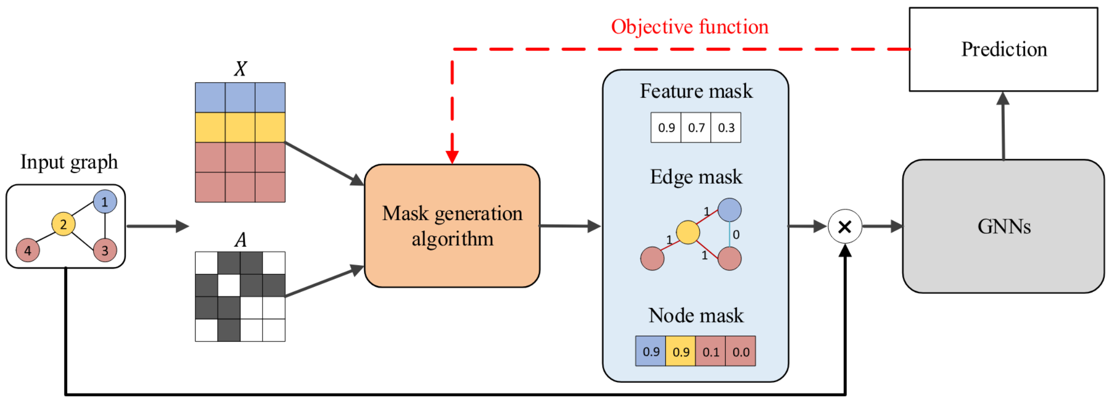


Figure 3.1: General pipeline of perturbation-based methods with a soft mask for node features, a discrete mask for edges, and an approximated discrete mask for nodes. The prediction of the GNN for the masked input graph is used in the objective function to train the generation algorithm and learn explanations. Reprinted from [**yuan2022explainability**].

The perturbation-based PGExplainer [**luo2020parameterized**] claims to satisfy all the criteria, while also maintaining reasonable computational cost. Though Yuan et al. note that the PGExplainer "is not performing as promising as its original reported results" [**yuan2022explainability**], we select this model for the course of this study with regard to its applicability in the inductive setting.
The general pipeline for different perturbation based approaches (see Figure 3.1) can be described as follows: First, the important features from the input graph are converted into a mask by our generation algorithm, depending on the explanation task at hand. These masks are applied to the input graph to highlight said features. Lastly, the masked graph is fed into the trained GNN to evaluate the mask. The mask generation algorithm is updated according to the similarity of the predictions on the original and masked graphs.

These different approaches mostly differ in the specific mask generation algorithm, the type of mask used and the objective function. It is important to distinguish between soft masks, discrete masks and approximated discrete masks. Soft masks take continuous values between $[0, 1]$ which enables the graph algorithm to be updated via backpropagation. A downside of soft masks is that they suffer from the "introduced evidence" problem (see Dabkowski and Gal [**dabkowski2017real**]). Any mask value that is non-zero or non-one may add new semantic meaning or noise to the input graph, since graph edges are by nature discrete. Discrete masks however always rely on non-differentiable operations, e.g. sampling. Thus, the approximated discrete masks utilize reparameterization tricks to avoid the "introduced evidence" problem while also enabling back-propagation.

Explanations can on the one hand be evaluated by visualizing the graph and considering the "human-comprehensibility". Since this requires a ground truth (GT), is prone to

the subjective understanding and is usually performed for a few random samples, it is important to apply stable evaluation metrics.
One relevant accuracy metric for synthetic datasets with ground truths is the Area Under the Receiver Operating Characteristic Curve (ROC-AUC) (see Richardson et al. [**RICHARDSON2024100994**]). The Receiver Operating Characteristic (ROC) curve plots the False Positive Rate (FPR) on the x-axis against the True Positive Rate (TPR), across different classification thresholds. The area under the curve (AUC) is calculated for said curve, resulting in the ROC-AUC, also referred to as AUROC. It is important to note, that a value of 0.5 equals random guessing, while a score of 1.0 indicates perfect classification. TODO: Other metrics include fidelity

3.2 TODO: SECTION NAME Explainer Models

GNNExplainer
Ying et al. proposed the GNNExplainer [**ying2019gnnexplainer**] - the first general, model-agnostic explainer for graph neural networks on any graph-based machine learning task. It is able to identify a concise subgraph structure and a subset of node features, that play a crucial role in the prediction of the underlying graph neural network. This is generally understood as an explanation. The work by Ying et al. serves as the main baseline for the PGExplainer, that seeks to improve its predecessor. Many concepts, experiments and specifications of PGExplainer were adapted from the GNNExplainer, which we seek to process in our work.

Parameterized Explainer for Graph Neural Network
The Parameterized Explainer for Graph Neural Network (PGExplainer) by Lou et al. [**luo2020parameterized**] is the main subject of our work. The idea of the framework is to collectively explain predictions of GNNs on a set of instance, while at the same time being able to generalize the explainer model to unexplained instances in an inductive setting. To achieve this, the method adapts a deep neural network to parameterize the generation process of explanations.
We reimplement the original work using PyTorch [**paszke2019pytorch**] and PyTorch Geometric [**Fey/Lenssen/2019**] instead of TensorFlow [**tensorflow2015-whitepaper**], while also emphasizing its application in the inductive setting, where test instances are unseen during training, as opposed to the original collective setting.
A secondary study on the inductive performance was also performed by the authors, which we want to extend by applying it on a graph neural network with a slightly different architecture, testing whether the explainer proves to be model-agnostic. The goal of this thesis is to apply the explainer model to a deep learning approach for solving a bipartite graph problem - specifically, the boolean satisfiability problem - and generate proofs for the deep model's predictions.

[Re] Parameterized Explainer for Graph Neural Network
Holdijk et al. [**holdijk2021re**] performed a replication study on the PGExplainer that focuses on reimplementing the method in PyTorch, testing whether the claims with respect to the GNNExplainer hold and discussing whether the used evaluation method makes sense. They highlight a large discrepancy between the paper and codebase, making a replication that includes the evaluation method from the paper alone impossible. With help of the codebase, the authors are able to replicate the experiments and verify the main claims of the original paper. However, they express some concerns regarding the evaluation setup and note a large difference between the originally noted AUC scores and their results for most of the datasets. Additionally, they question the general approach for evaluating graph data with ground truths, as done in GNNExplainer and PGExplainer, which we will discuss in **??**. We use this work as an additional baseline for our approach, but note that the replication was also done in the collective setting. Therefore, the difference to our work lies mainly in the architecture used for the downstream model and the setting of the experiments.

3.3 TODO: SECTION NAME Downstream Model

NeuroSAT
Proposed by Selsam et al., NeuroSAT [**selsam2018learning**] is a machine learning approach for solving the boolean satisfiability problem using a message passing neural network. It is able to detect satisfying assignments, but lacks proofs of unsatisfiability. The authors performed a small study on the detection of unsatisfiable cores, revealing that NeuroUNSAT is able to detect unsatisfiable cores if the UNSAT problems contain a specific core. However, this is expected to be due to the model memorizing the cores, rather than generalizing to any unsatisfiable core. We want to test this by applying the PG-Explainer to the NeuroSAT model, and evaluate whether the generated explanations do align with such cores, more specifically with MUSs.

Chapter 4

PGExplainer - Methodology

In this chapter we present the Methodology (TODO: ?) of our work, including the detailed theory of the explainer model used in our approach. Thus, we start by presenting the concept of the PGExplainer [**luo2020parameterized**] in Section 4.1.

In Section 4.2 we present our idea of applying the PGExplainer on the NeuroSAT framework to generate explanations for a machine learning SAT-solving approach and comparing these to the "human-understandable" concept of unsatisfiable cores.

We then describe our implementation in detail (see Section 4.3), including the changes made and difficulties during the process, as well as the adaptations for the application on NeuroSAT.

4.1 Theoretical Foundations of PGExplainer

In the following subchapter, we introduce the PGExplainer [**luo2020parameterized**] and its concepts. The idea is to generate explanations in the form of edge distributions or soft masks using a probabilistic generative model for graph data, known for being able to learn the concise underlying structures from the observed graph data. The explainer uncovers said underlying structures, believed to have the biggest impact on the prediction of a GNNs, as explanations. This approach may be applied to any trained GNN model, henceforth referred to as the target model (TM).

By utilizing a deep neural network to parameterize the generation process, the explainer learns to collectively explain multiple instances of a model. Since the parameters of the neural network are shared across the population of explained instance, PGExplainer provides "model-level explanations for each instance with a global view of the GNN model" [**luo2020parameterized**]. Furthermore, this approach cannot only be used in a collective setting, but also in an inductive setting, where explanations for unexplained nodes can be generated without retraining the explanation model. This improves the generalizability compared to previous works, particularly the GNNExplainer by Ying et al.

[**ying2019gnnexplainer**] and focuses on explaining graph structure rather than graph features.

We follow the structure of the original paper [**luo2020parameterized**] and start by describing the learning objective in Section 4.1.1, the utilized reparameterization trick in Section 4.1.2, the idea of global explanations in Section 4.1.3 and finally the applied regularization terms in Section 4.1.4.

4.1.1 Learning Objective

To explain the predictions made by a GNN model for an original input graph G_o with m edges we first define the graph as a combination of two subgraphs: $G_o = G_s + \Delta G$, where G_s represents the subgraph holding the most relevant information for the prediction of a GNN, referred to as explanatory graph. ΔG contains the remaining edges that are deemed irrelevant for the prediction of the GNN. Inspired by GNNExplainer [**ying2019gnnexplainer**], the PGExplainer then finds G_s by maximizing the mutual information between the predictions of the target model and the underlying G_s:

$$\max_{G_s} MI(Y_o; G_s) = H(Y_o) - H(Y_o|G = G_s), \tag{4.1}$$

where Y_o (TODO: $\in (0,1)^c$??) is the prediction of the target model with G_o as input and number of possible classes c. This quantifies the probability of prediction Y_o when the input graph is restricted to the explanatory graph G_s, as in the case of $I(Y_o; G_s) = 1$, knowing the explanatory graph G_s gives us complete information about Y_o, and vice versa. Intuitively, if removing an edge (i, j) changes the prediction of a GNN drastically, this edge is considered important and should therefore be included in G_s. This idea originates from traditional forward propagation based methods for white-box explanations (see Dabkowski and Gal [**dabkowski2017real**]). It is important to note that $H(Y_o)$ is only related to the target model with fixed parameters during the evaluation/explanation stage. This leads to the objective being equivalent to minimizing the conditional entropy $H(Y_o|G = G_s)$.

To optimize this function a relaxation is applied for the edges, since normally there would be 2^m candidates for G_s. The explanatory graph is henceforth assumed to be a Gilbert random graph, where the selections of edges from G_o are conditionally independent to each other. However, the authors describe a random graph with each edge having its own probability, rather than a shared probability as described in Section 2.2, as follows: Let $e_{i,j} \in V \times V$ be the binary variable indicating whether the edge is selected, with $e_{i,j} = 1$ if edge (i, j) is selected to be in the graph, and 0 otherwise. For the random graph variable G the probability of a graph G can be factorized as

$$P(G) = \prod_{(i,j) \in E} P(e_{i,j}). \tag{4.2}$$

TODO: Inhomogeneous Erdős Rényi model? Mention that this is a generative model? (A Gilbert random graph is an example of a generative probabilistic model on graph data?) $P(e_{i,j})$ is instantiated with the Bernoulli distribution $e_{i,j} \sim Bern(\theta_{ij})$, where $P(e_{i,j} = 1) = \theta_{ij}$ is the probability that edge (i, j) exists in G. After this relaxation the learning objective becomes:

$$\min_{G_s} H(Y_o|G = G_s) = \min_{G_s} \mathbb{E}_{G_s}[H(Y_o|G = G_s)] \approx \min_{\Theta} \mathbb{E}_{G_s \sim q(\Theta)}[H(Y_o|G = G_s)], \tag{4.3}$$

where $q(\Theta)$ is the distribution of the explanatory graph that is parameterized by Θ's.

4.1.2 Reparameterization Trick

As described in Section 3.1, a reparameterization trick can be utilized to relax discrete edge weights to continuous variables in the range $(0, 1)$. PGExplainer uses the reparameterizable Gumbel-Softmax estimator [**jang2016categorical**] to allow for efficiently optimizing the objective function with gradient-based methods. This method introduces the Gumbel-Softmax distribution, a continuous distribution used to approximate samples from a categorical distribution.
A temperature τ is used to control the approximation, usually starting from a high value and annealing to a small, non-zero value. Samples with $\tau > 0$ are not identical to samples from the corresponding continuous distribution, but are differentiable and therefore allow back-propagation [**abid2019concrete**]. The sampling process $G_s \sim q(\Theta)$ of PGExplainer is therefore approximated with a determinant function that takes as input the parameters Ω, a temperature τ and an independent random variable ϵ: $G_s \approx \hat{G} = f_\Omega(G_o, \tau, \epsilon)$.
The binary concrete distribution [**maddison2016concrete**], also referred to as Gumbel-Softmax distribution, is utilized as an instantiation for the sampling, yielding the weight $\hat{e}_{i,j} \in (0, 1)$ for edge (i, j) in $\hat{G}_s$, computed by:

$$\epsilon \sim \text{Uniform}(0, 1), \qquad \hat{e}_{i,j} = \sigma((\log \epsilon - \log(1 - \epsilon) + \omega_{ij}/\tau), \tag{4.4}$$

where $\sigma(\cdot)$ is the Sigmoid function and $\omega_{ij} \in \mathbb{R}$ is an explainer logit for the corresponding edge used as a parameter. When $\tau \to 0$, e.g. during the explanation stage, the weight $\hat{e}_{i,j}$ is binarized with the sigmoid function $\lim_{\tau \to 0} P(\hat{e}_{i,j} = 1) = \frac{e(\omega ij)}{1 + e(\omega ij)}$. Since $P(e_{i,j} = 1) = \theta_{ij}$, choosing $\omega_{ij} = \log \frac{\theta_{ij}}{1 - \theta_{ij}}$ leads to $\lim_{\tau \to 0} \hat{G}_s = G_s$ and justifies the approximation of the Bernoulli distribution with the binary concrete distribution. During training, when $\tau > 0$, the objective function in (4.3) is smoothed with a well-defined gradient $\frac{\partial \hat{e}_{i,j}}{\partial \omega_{ij}}$ and becomes:

$$\min_{\Omega} \mathbb{E}_{\epsilon \sim \text{Uniform}(0,1)} H(Y_o|G = \hat{G}_s) \tag{4.5}$$

The authors follow the approach of GNNExplainer [**ying2019gnnexplainer**] and modify the objective by replacing the conditional entropy with cross entropy between the label class and the prediction of the target model. This is justified by the greater importance

of understanding the model's prediction of a certain class, rather than providing an explanation based solely on its confidence.
With the modification to cross-entropy $H(Y_o, \hat{Y}_s)$, where $\hat{Y}_s$ is the prediction of the target model when $\hat{G}_s$ is given as input, as well as the adaption of Monte Carlo sampling, the learning objective becomes:

$$\begin{aligned}\min_{\Omega} \mathbb{E}_{\epsilon \sim \text{Uniform}(0,1)} H(Y_o, \hat{Y}_s) &\approx \min_{\Omega} -\frac{1}{K} \sum_{k=1}^{K} \sum_{c=1}^{C} P(Y_o = c) \log P(\hat{Y}_s = c) \\ &= \min_{\Omega} -\frac{1}{K} \sum_{k=1}^{K} \sum_{c=1}^{C} P_{\Phi}(Y_o = c | G = G_o) \log P_{\Phi}(\hat{Y}_s = c | G = \hat{G}_s^{(k)}).\end{aligned} \tag{4.6}$$

Φ denotes the parameters in the target model, K is the number of total sampled graphs, C is the number of class labels, and $\hat{G}_s^{(k)}$ denotes the k-th graph sampled with Equation 4.4, parameterized by Ω.

4.1.3 Global Explanations

The novelty of PGExplainer lies in the ability to generate explanations for graph data with a global perspective, that allow for understanding the general picture of a model across a population. This saves resources when analyzing large graph datasets, as new instances can be explained without retraining the model, and can also be helpful for establishing the users' trust in these explanations. To achieve this the authors propose the use of a parameterized network that learns to generate explanations from the target model, which also apply to not yet explained instances.
GNN models generally apply two functions to first learn node representations and utilize these in downstream tasks such as graph classification or node classification (TODO: SOURCE!). Therefore, Luo et al. [**luo2020parameterized**] define two general functions $\text{GNNE}_{\Phi_0}(\cdot)$ and $\text{GNNC}_{\Phi_1}(\cdot)$ for any GNN in the context of PGExplainer. $\text{GNNE}_{\Phi_0}(\cdot)$ denotes a model of L stacked GNN layers that returns higher dimensional node representations of the input graph and its node features. These are used as input for both the downstream task $\text{GNNC}_{\Phi_1}(\cdot)$ and the explainer network (see Equation 4.8). For models without explicit classification layers the last layer is used instead. It follows

$$\mathbf{Z} = \text{GNNE}_{\Phi_0}(G_o, \mathbf{X}), \qquad Y = \text{GNNC}_{\Phi_1}(\mathbf{Z}), \tag{4.7}$$

where $\mathbf{Z}$ denotes a matrix of node representations $\mathbf{z}$, referred to as node embeddings. $\mathbf{X} \in \mathbb{R}^{|V(G_o)| \times d}$ denotes the d-dimensional node features of G_o. The general downstream GNN model is visualized in Figure 4.1. Note that $\mathbf{Z}$ encapsulates both features and structure of the input graph and therefore serves as input for the explainer network g, defined as: TODO: Rename g?

$$\Omega = g_{\Psi}(G_o, \mathbf{Z}). \tag{4.8}$$

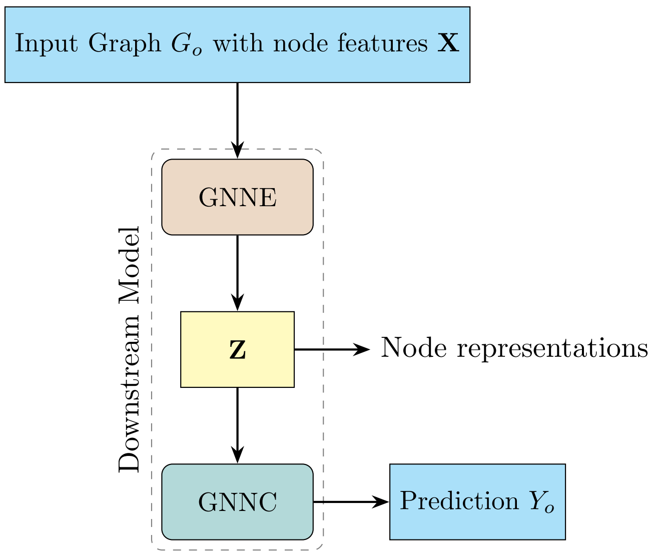


Figure 4.1: A "black-box" downstream model in the context of PGExplainer.

Ψ denotes the parameters in the explanation network and the output Ω is treated as parameter in Equation 4.6. Since Ψ is shared by all edges among the population, PG-Explainer collectively provides explanations for multiple instances. Thus, the learning objective in a collective setting with $\mathcal{I}$ being the set of instances becomes:

$$\min_{\Psi} -\frac{1}{K}\sum_{i\in\mathcal{I}}\sum_{k=1}^{K}\sum_{c=1}^{C} P_\Phi(Y_o = c|G = G_o^{(i)}) \log P_\Phi(\hat{Y}_s = c|G = \hat{G}_s^{(i,k)}). \tag{4.9}$$

Consequently, $G_o^{(i)}$ and $G_s^{(i,k)}$ denote the input graph and the k-th graph sampled with Equation 4.4, respectively for instance i.

The authors propose two slightly different instantiations of Ω for node classification and graph classification tasks.

Explanation Network for Graph Classification For graph level tasks, the authors consider each graph to be an instance, independent of specific nodes. The output Ω of the network (see Equation 4.8) is thus specified as:

$$\omega_{ij} = \text{MLP}_\Psi([\mathbf{z}_i \oplus \mathbf{z}_j]), \tag{4.10}$$

where MLP_Ψ is an MLP (see Section 4.3.1 for implementation details) parameterized with Ψ and $\oplus$ denotes the concatenation operation. Effectively, for each edge in G_o a concatenation of both its nodes is fed through the MLP. For the MLP used in both the original and this work the output ω_{ij} is an edge logit, which serves as a parameter in the sampling process.

Explanation Network for Node Classification

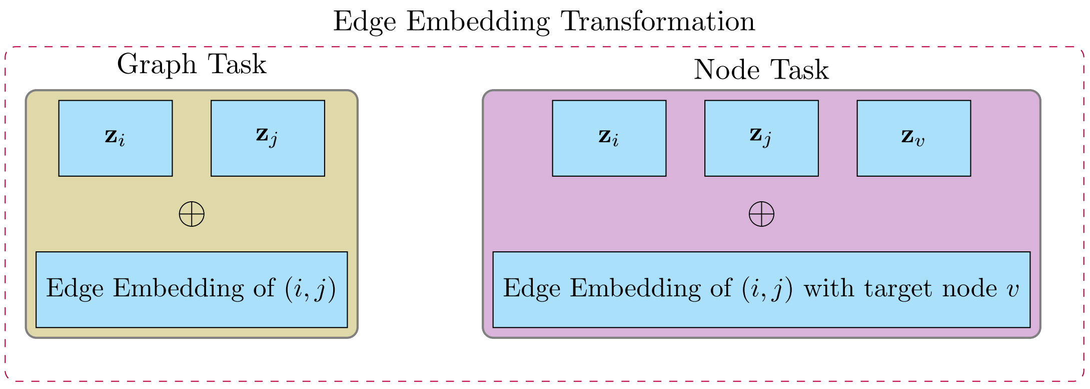


Figure 4.2: TODO: FOR ALL EDGES IN G_o Visualization of the edge embedding transformation used to create inputs for the explainer network. Depending on the downstream task used in the target model the created edge embeddings differ slightly.

For node level tasks on the other hand, each prediction node is considered as an instance. Let an edge (i, j) be considered relevant for the prediction of a node u, but irrelevant for the prediction of a node v. To explain the prediction of node v the authors specify the output Ω of the network in Equation 4.8 as:

$$\omega_{ij} = \text{MLP}_\Psi([\mathbf{z}_i \oplus \mathbf{z}_j \oplus \mathbf{z}_v]). \tag{4.11}$$

Thus, a concatenation of the node embeddings of nodes i, j and v respectively is fed through the network. We denote this preprocessing step of creating edge embeddings from node embeddings as edge embedding transformation (see Figure 4.2). The full pipeline of the PGExplainer can be observed in figure 4.3.

It is important to note, that for L-layer target GNNs utilizing a message passing mechanism, the prediction at node v is fully determined by its local computation graph. The local computation graph of v is defined as its $L-$hop neighborhood $\mathcal{N}_L(v)$ [**ying2019gnnexplainer**] (see Figure 2.4).

Collective and Inductive Setting

Due to the nature of its predecessor GNNExplainer, the authors main focus was the application in a collective setting, where the goal is to explain a full population of instances by training on all these and thus being able to provide explanations for every single one. However, since the PGExplainer utilizes a deep neural network to parameterize the generation process of explanations for a population, it can be utilized in an inductive setting, unlike its predecessor.
This means, that explanations can be generated for instances from the same population, that have not been seen during training. Thus, it is not necessary to retrain the explainer for new instances of the same population, effectively reducing the computational complexity by the training time when compared to the GNNExplainer.

This leads to an improved computational complexity when compared to their baseline GNNExplainer, since for one the number of parameters in the explainer does no longer depend on the size of the input graph and since for another the explainer does not have to be retrained for every unexplained instance.

4.1.4 Regularization Terms

To enhance the preservation of desired properties of explanations the authors propose various regularization terms. These are added to the learning objective, depending on the specific downstream task at hand.

Size and Entropy Constraints
Inspired by GNNExplainer [**ying2019gnnexplainer**], to obtain compact and precise explanations, a constraint on the size of the explanations is added in the form of $||\Omega||_1$, the l_1 norm on latent variables Ω. Additionally, to encourage the discreteness of edge weights, element-wise entropy is added as a constraint:

$$H_{\hat{G}_s} = -\frac{1}{|\varepsilon|} \sum_{(i,j)\in\varepsilon} (\hat{e}_{i,j} \log \hat{e}_{i,j} + (1-\hat{e}_{i,j}) \log(1-\hat{e}_{i,j})), \tag{4.12}$$

for one explanatory graph $\hat{G}_s$ with ε edges. For the collective setting, this is added as a mean over all instances in $\mathcal{I}$.

Note that the following two constraints are not used in the original experimental setup, but serve as inspiration for constraints introduced in our NeuroSAT application (see Section 4.2) and are therefore included.

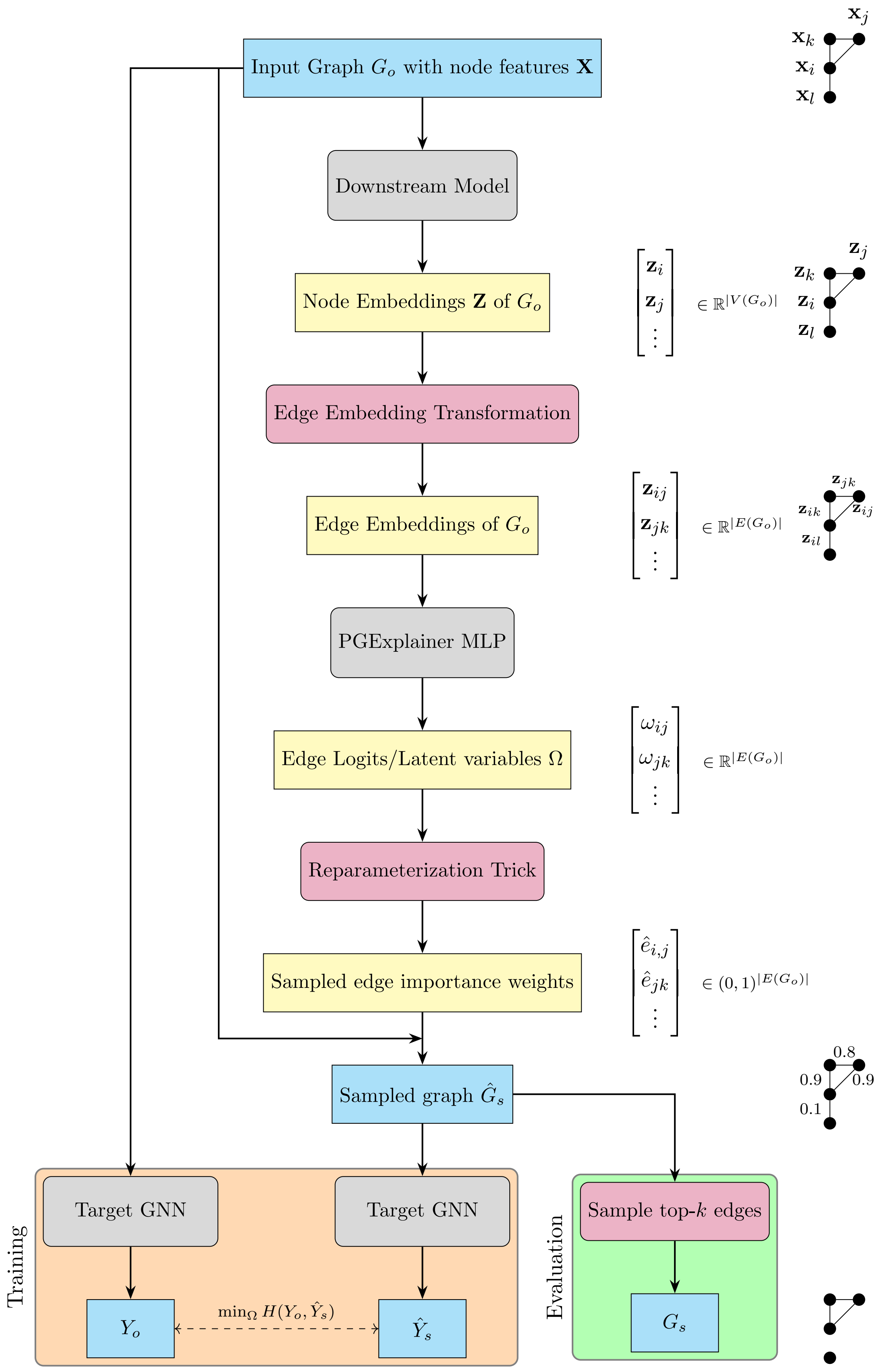
Budget Constraint
The authors propose the modification of the size constraint to a budget constraint, for a predefined available budget B. Let $|\hat{G}_s| \leq B$, then the budget regularization is defined as:

$$R_b = \text{ReLU}(\sum_{(i,j)\in\varepsilon} \hat{e}_{i,j} - B). \tag{4.13}$$

Note that $R_b = 0$ when the explanatory graph is smaller than the budget. When out of budget, the regularization is similar to that of the size constraint.

Connectivity Constraint
To enhance the effect of the explainer detecting a small, connected subgraph, motivated through real-life motifs being inherently connected, the authors suggest adding the cross-entropy of adjacent edges. Let (i,j) and (i,k) be two edges that both connect to the node i, then (i,k) should rather be included in the explanatory graph if the edge (i,j) is selected to be included. This is formally defined as:

$$H(\hat{e}_{i,j}, \hat{e}_{i,k}) = -[1-\hat{e}_{i,j} \log(1-\hat{e}_{i,k}) + \hat{e}_{i,j} \log \hat{e}_{i,k}]. \tag{4.14}$$



Input Graph G_o with node features $\mathbf{X}$
$\mathbf{x}_j$
$\mathbf{x}_k$
$\mathbf{x}_i$
$\mathbf{x}_l$
Downstream Model
Node Embeddings $\mathbf{Z}$ of G_o
$\begin{bmatrix} \mathbf{z}_i \\ \mathbf{z}_j \\ \vdots \end{bmatrix} \in \mathbb{R}^{|V(G_o)|}$
$\mathbf{z}_j$
$\mathbf{z}_k$
$\mathbf{z}_i$
$\mathbf{z}_l$
Edge Embedding Transformation
Edge Embeddings of G_o
$\begin{bmatrix} \mathbf{z}_{ij} \\ \mathbf{z}_{jk} \\ \vdots \end{bmatrix} \in \mathbb{R}^{|E(G_o)|}$
$\mathbf{z}_{jk}$
$\mathbf{z}_{ik}$
$\mathbf{z}_{ij}$
$\mathbf{z}_{il}$
PGExplainer MLP
Edge Logits/Latent variables Ω
$\begin{bmatrix} \omega_{ij} \\ \omega_{jk} \\ \vdots \end{bmatrix} \in \mathbb{R}^{|E(G_o)|}$
Reparameterization Trick
Sampled edge importance weights
$\begin{bmatrix} \hat{e}_{i,j} \\ \hat{e}_{jk} \\ \vdots \end{bmatrix} \in (0,1)^{|E(G_o)|}$
Sampled graph $\hat{G}_s$
0.8
0.9
0.9
0.1
Training
Target GNN
Target GNN
Y_o
$\min_\Omega H(Y_o, \hat{Y}_s)$
$\hat{Y}_s$
Evaluation
Sample top-k edges
G_s

Figure 4.3: TODO: The complete pipeline of PGExplainer.

We note that in practice this is implemented only for the two highest edge weights for each edge. The definition therefore would change to (i, j) and (i, k) being the edges carrying the top two edge weights from the nodes connecting to node i.

4.2 Extension to application on NeuroSAT

In this section we propose additional restraints to fit the explanations of PGExplainer to the structure of SAT formulae. We start by giving a short introduction to NeuroSAT[**selsam2018learning**] and how it may function as a downstream model.

Proposed by Selsam et al. [**selsam2018learning**], NeuroSAT utilizes a MPNN, generally utilized to express GNNS, to solve SAT formulae. It is able to generalize in the sense that it may solve substantially larger and more difficult formulae than seen during training by running for more iterations. Though it may be used to calculate variable assignments that satisfy a formula, it is unable to provide proofs for formulae that are unsatisfiable.
In a separate experiment the authors were able to make NeuroSAT identify specific unsatisfiable cores, however this is assumed to be due to the network memorizing the subgraphs rather than learning a generic procedure that proves unsatisfiability. To verify this we aim to generate explanations for unsatisfiable formulae processed by a trained NeuroSAT model.
Since the following restraints are tailored to the application on bipartite graphs representing SAT formulae, we have to define the specifics of the input graph. A formula is encoded as an undirected bipartite graph G_b, with bipartition (L, C). It contains one node for every literal $l \in L$, one for every clause $c \in C$ and edges for all combinations (l, c) where l appears in clause c.
Additionally, connections exist between each literal and its negation, since messages are also passed along these. Note that these edges are not present in the biadjacency matrix $\mathbf{B} \in (0, 1)^{L \times C}$. $\mathbf{B}$ is used as input for the NeuroSAT model, without explicit node features. In the following definitions we let G_b be the original input graph for the PGExplainer, completely defined by its biadjacency matrix $\mathbf{B}$.

TODO: MORE FORMAL DEFINITION? Since PGExplainer generates edge wise explanations, and we want to evaluate the SAT problem evaluations with unsatisfiable cores as ground truth, we need to adapt the framework to account for the definition of unsatisfiable cores. Since a core is a subset of clauses, predicting singular edges that represent literals being present in a clause, may not provide sufficient results in the sense of human understandable explanations. Therefore, we propose a soft and a hard restraint that encourage the explainer to predict sets of edges that connect to the same node c, approximating the prediction of a complete clause.
The remainder of the explainer pipeline stays identical. The downstream task - NeuroSAT - calculates hidden node representations $\mathbf{h}^t$ for the input graph G_b, now modeling a SAT

instance, at each iteration t. The representations in the last iteration T are extracted as node embeddings $\mathbf{z}$ for clause and literal nodes respectively, and transformed into edge embeddings (see Equation 4.10). Though this is only done for node level tasks in the original, we also consider using a concatenation of multiple hidden representations $\mathbf{h}^{\frac{1}{2}T} \oplus \mathbf{h}^{\frac{3}{4}T} \oplus \mathbf{h}^{T}$ as node embeddings $\mathbf{z}$. These serve as the input of the explainer MLP and are processed as usual, with either of the following additional limitations.

Soft Modified Connectivity Constraint

To account for the definition of unsatisfiable cores - a subset of clauses in the original formula whose conjunction is still unsatisfiable - we add a constraint that reinforces the prediction of complete clauses. Therefore, if the explainer assigns a high score to an edge (l_1, c), all edges $(l_k, c) \in E(c)$ that also connect to the clause node c should receive a high score. Therefore, we introduce a soft constraint that punishes varying edge weights for the same clause. For our sampled bipartite Graph $\hat{G}_s$ with node sets L and C containing literal nodes and clause nodes respectively, we define:

$$R_C = \sum_{c \in C} \mathrm{Var}(\hat{E}_c) = \sum_{c \in C} \frac{1}{|E(c)|} \sum_{(l,c) \in E(c)} (\hat{e}_{l,c} - \bar{E}_c)^2, \tag{4.15}$$

where $\hat{E}_c = \{\hat{e}_{l,c} \mid (l,c) \in E(\hat{G}_s)\}$ is the set of edge weights corresponding to edges incident to c and $\bar{E}_c = \frac{1}{|\hat{E}_c|} \sum_{\hat{e}_{l,c} \in \hat{E}_c} \hat{e}_{l,c}$ denotes the mean of $\hat{E}_c$. This is added to our objective function during training.

Hard Constraint

Since the soft constraint only encourages the prediction of entire clauses but does not enforce it, we also propose a hard constraint that modifies the predicion process. We restrain the edge logits $\omega_{i,j}$ calculated by the MLP to be identical for all edges that connect to the same clause. For all clause nodes $c \in C$, we calculate the mean logit μ_c of all edges incident to c with

$$\mu_c = \frac{1}{|E(c)|} \sum_{(l,c) \in E(c)} \omega_{l,c}. \tag{4.16}$$

The update rule is then defined as

$$\omega'_{l,c} \leftarrow \mu_c, \tag{4.17}$$

since edges in the biadjacency matrix are from literals to clauses at all times.
The reparameterization trick is still applied for each edge, but ϵ_c is sampled per clause instead of per edge, so that all edges that connect to a clause are forced to not only bear the same logit, but also the same importance score during training. The Equation 4.4 thus changes to:

$$\epsilon_c \sim \mathrm{Uniform}(0,1), \qquad \hat{e}_{l,c} = \sigma((\log \epsilon_c - \log(1-\epsilon_c) + \omega_{l,c}/\tau). \tag{4.18}$$

4.3 Implementation details

In the following, we provide the implementation details needed to reproduce our results. This includes the general replication of the PGExplainer [**luo2020parameterized**] with the adapted downstream models in Section 4.3.1 and the specifics for the application on NeuroSAT [**selsam2018learning**] in Section 4.3.2.

4.3.1 Replication of PGExplainer

For our replication we try to implement the methods and details as close to the original paper as possible. Thus, we follow the general pseudocode algorithms presented by the authors (see Appendix A.1). Since the paper differs from the original codebase and is imprecise about certain descriptions, as found by Holdijk et al. [**holdijk2021re**], we aim to give a thorough description. This includes the tools we used, resulting changes regarding the data processing, the general architecture and hyperparameters of the downstream models, the architecture and hyperparameters of the explainer model, as well as concrete methods implemented in the model.

Libraries

To reimplement the framework, we utilize a couple of libraries that we introduce shortly. Most notably, we use PyTorch Geometric [**Fey/Lenssen/2019**], a library built upon PyTorch [**paszke2019pytorch**], that provides methods to create and train GNNs. For evaluation of the explainer model, specifically for calculating the ROC-AUC score, we use TorchEval, a model evaluation library that is part of PyTorch. Furthermore, we integrate WandB [**wandb**] to monitor model performance and allow for easy hyperparameter searches. To visualize the graphs and their explanations we employ NetworkX [**SciPyProceedings˙11**]. Lastly, we utilize seaborn [**Waskom2021**] to plot loss curves and other metrics.

Preprocessing

We use the original datasets that are provided in the PGExplainer codebase[1]. We transform the data to fit our PyTorch Geometric framework. Each graph is stored as a torch-geometric Data object. This holds the d-dimensional node features as tensors, the graph label index in the form of a long or all node labels in the form of a tensor of class indices, a ground truth edge mask that contains the edges present in the motif, as well as training-, evaluation- and test-node-masks for training the downstream model.

In PyTorch Geometric edges are stored in the edge-index format as a COO tensor - a PyTorch coordinate format that stores tuples of element indices and their corresponding values. In the context of graph edges in PyTorch Geometric, for an edge (i, j) the element

[1] `https://github.com/flyingdoog/PGExplainer`

index is the starting node i and the corresponding value its incident node j. This is computed from the adjacency matrix A as follows:

```
    edge_index = A.nonzero().t().contiguous()
```

Listing 4.1: Edge index transformation

First, the matrix indices or coordinates of the edges - non-zero elements - are extracted. These are then transposed and lastly stored in contiguous memory. The resulting shape of the edge-index is TODO: $\mathbb{N}^{2\times\#\text{edges}}$. Therefore, we only transform the data without changing its content.

Reproducibility

Inspired by Holdijk et al. [**holdijk2021re**] we implement the ability to seed the experiments performed on PGExplainer. PyTorch Geometric provides a way of seeding all modules that generate random numbers during the training process, including torch and python random. To further increase reproducibility, we utilize PyTorch's `use_deterministic_algorithms`, forcing the learning algorithm to only use deterministic algorithms. For the dataset splits we use a separate fixed seed that consistently creates the same sets across all training runs and experiments.

Downstream Model Specifications

In PGExplainer two slightly different architectures of GNNs for node classification and graph classification are introduced. We recreate these in PyTorch Geometric, while changing the exact layers used in the network to test whether the claim that the explainer does apply to any downstream GNN model holds. These models implement the same downstream classification tasks on the given datasets that achieve accuracies of at least 85%, a baseline set in GNNExplainer [**ying2019gnnexplainer**]. The datasets as well as the exact accuracies of each of the models are presented in the experimental setup (see Section 5.1).

The model for both node- and graph classification consist of 3 `GraphConv` layers, a PyTorch Geometric implementation of the Higher-order GNN by Morris et al. [**morris2019weisfeiler**] (see Equation 2.31). Since this layer allows for the passing of edge weights, weights of one are passed by default - e.g. during downstream model training - to simply maintain the adjacency matrix. Each of these layers has 20 hidden units and is followed by a ReLU activation function. The first layer processes the d-dimensional input node features, while the remaining layers retain the hidden dimensionality of 20.

Holdijk et al. [**holdijk2021re**] found that the original code wrongfully uses undocumented batch normalization layers in training mode during evaluation, which leads to a deviation in results, and thus completely omit the use of batch normalization. We choose to add batch normalization between the first two layers and their activation functions for both models, similar to the original codebase, without the training mode error. Additionally, we add an optional dropout layer after each activation function to improve the generalizability on more difficult tasks.

Note that in their codebase the authors use a concatenation of the hidden representations at each layer instead of solely the final layer representation as node embeddings for node level tasks. This leads to $\mathbf{Z} \in \mathbb{R}^{|V(G_o)| \times (Ld)}$ being the matrix of node embeddings $\mathbf{z}$ that are computed as:

$$\mathbf{z}_i = \mathbf{h}_i^{(1)} \oplus \mathbf{h}_i^{(2)} \oplus \mathbf{h}_i^{(3)}, \tag{4.19}$$

with $\mathbf{h}_i^{(L)}$ denoting the hidden representations of node i in layer $L \in \{1, 2, 3\}$.

Thus, the node embeddings used in the explainer as well as in the downstream classification task each have a shape of $\mathbb{R}^{3(20)}$. For the classification a final linear layer is added to the model that maps each 60-dimensional node embedding to C classes. A softmax is applied to the model output to get class probabilities for each node. We also adopt this in our implementation.
The softmax function is defined as follows [**Goodfellow-et-al-2016**]: Given a vector $\mathbf{z} = [z_1, z_2, \ldots, z_K] \in \mathbb{R}^K$, the softmax function maps $\mathbf{z}$ to a probability distribution over K classes:

$$\text{Softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, 2, \ldots, K, \tag{4.20}$$

where e denotes the exponential function.

The model used for graph tasks differs slightly, in the sense that only the hidden embeddings of the last "GraphConv" layer are treated as node embeddings. In addition, before the final linear layer used for classification, both a max pooling and mean pooling operation are performed on the embeddings of each graph and the results concatenated to get a representation of a complete graph. Note that the paper only states to use max pooling, while in practice a concatenation of the max pooling and sum pooling is used. We adopt the combination of mean- and max pooling from the replication study [**holdijk2021re**], as this has been used in recent graph neural networks [**simonovsky2017dynamic**], [**ma2021unsupervised**], [**zhao2023faithful**].
The max pooling operation extracts the maximum value of each feature dimension across the nodes of a graph, while the mean pooling extracts the mean value of each feature dimension across all nodes of a graph. Since the results of both pooling operations are in $\mathbb{R}^{20}$, the resulting 40-dimensional graph embedding is fed into the linear layer, again mapping to C classes, and a softmax is applied to get probabilities of a graph belonging to each class.

Furthermore, following the original paper, all layer weights are initialized with Xavier initialization [**glorot2010understanding**], while biases are initialized with 0. The datasets

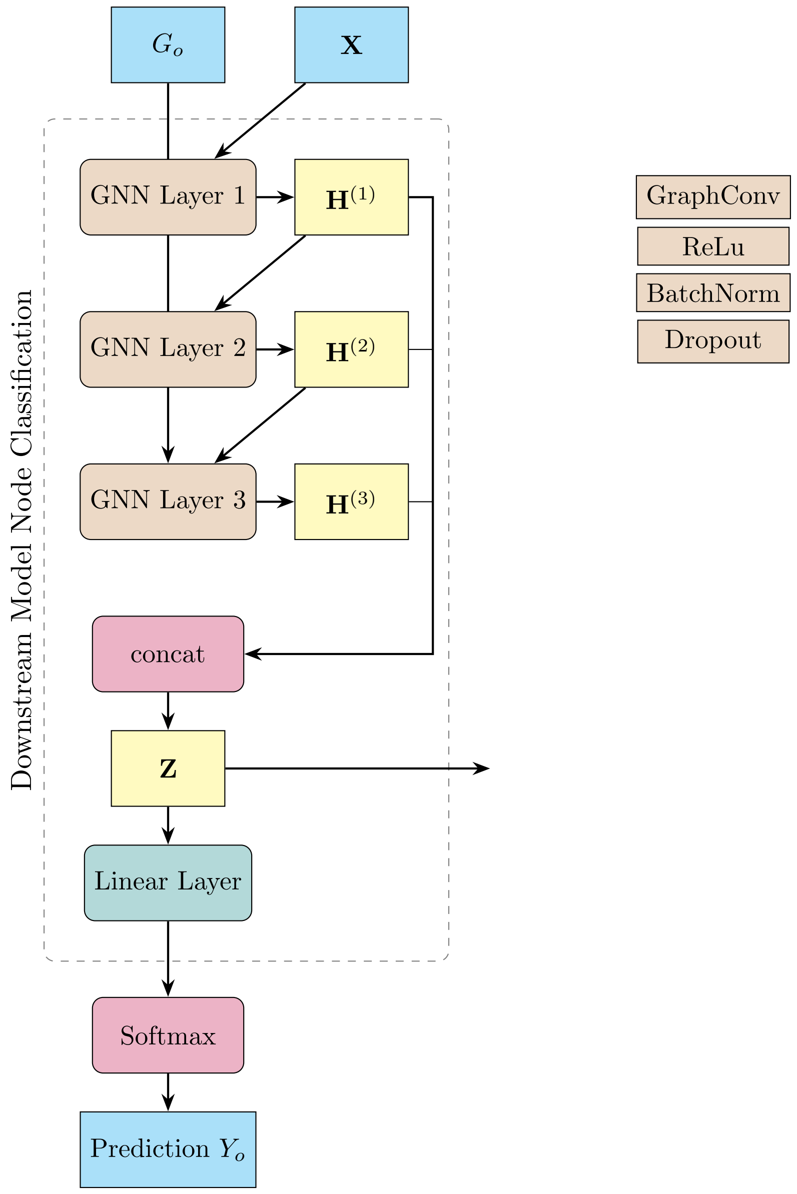


Figure 4.4: Visualization of the concrete node classification downstream model. concat denotes the row-wise concatenation along the feature dimension.

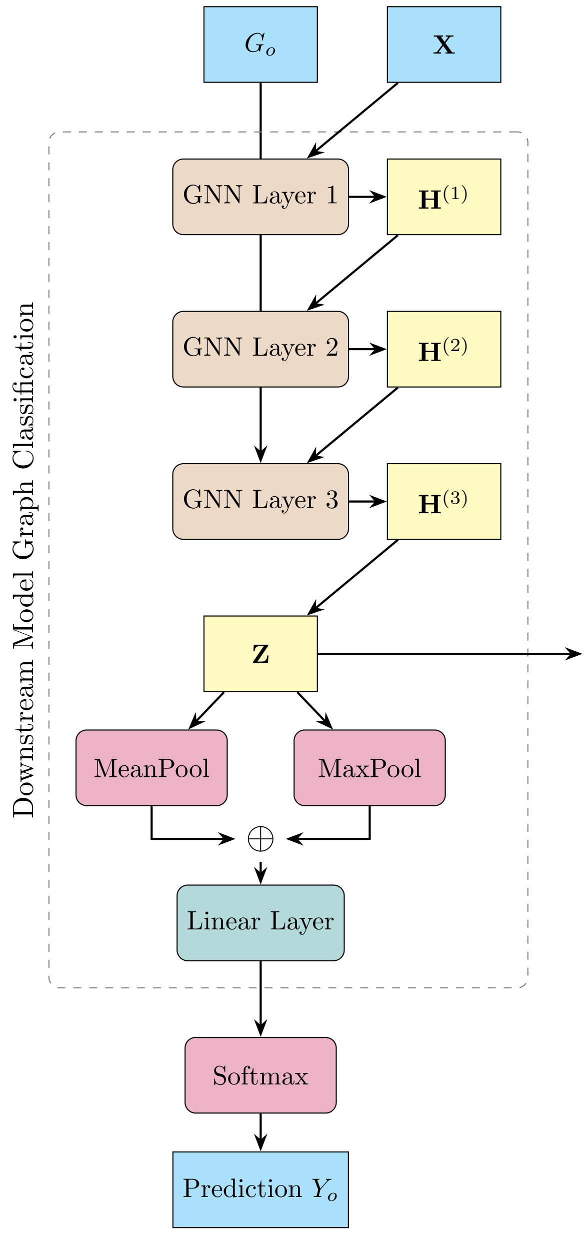
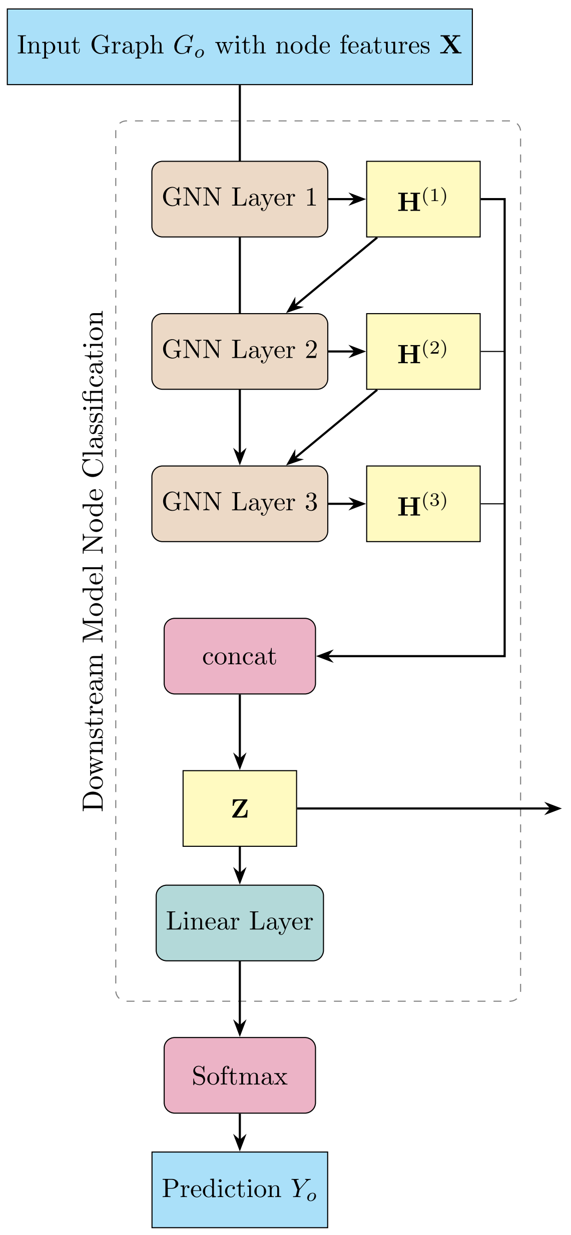
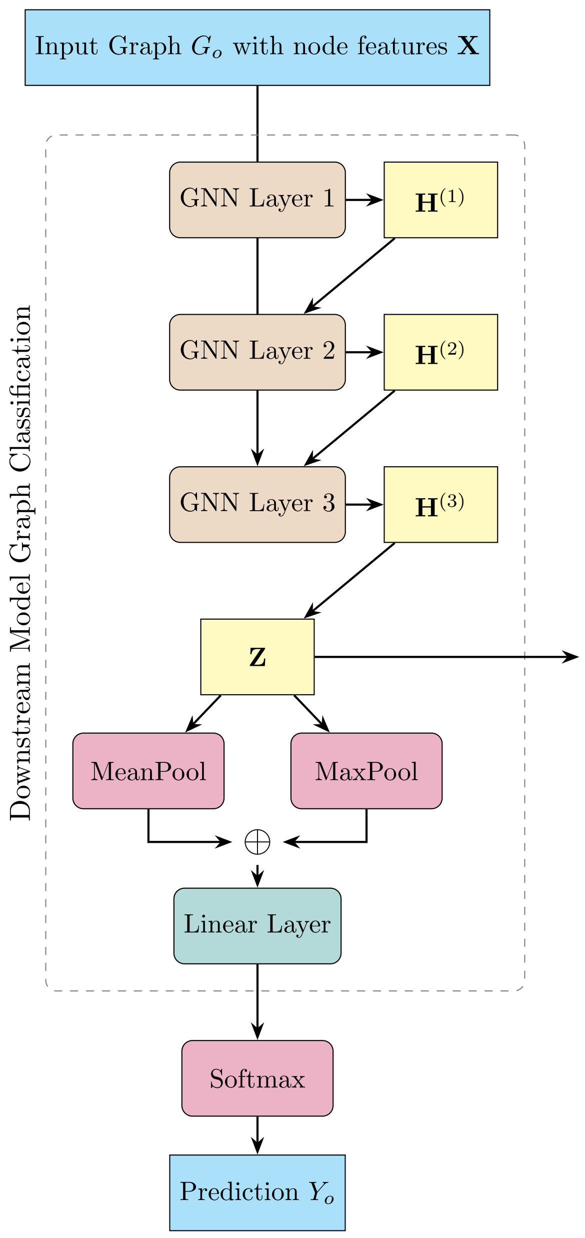


Figure 4.5: Visualization of the concrete graph classification downstream model.



(a) concat denotes row-wise concatenation along the feature dimension.



(b) Graph classification model.

Figure 4.6: Visualizations of two downstream GNN models for (a) node classification and (b) graph classification.

are split into training, testing and validation sets with an 80/10/10 split ratio. The models are trained for 1000 epochs, with a learning rate of 1.0×10^{-3} and Adam [**kingma2014adam**] is used as an optimizer. The loss is defined as the PyTorch `CrossEntropyLoss()` between the GNN prediction and the actual label of the input.

Note that the main differences between our downstream model and the model described in PGExplainer [**luo2020parameterized**] lie in the used graph layer, as well as the addition of dropout, two batch normalizations before the activations and the usage of a slightly different global pooling.
We want to stress that though not documented in the paper, early stopping is utilized in the downstream model training and the model state with the highest validation accuracy is selected. For models that achieve the highest accuracy at multiple epochs, the model state with the lowest validation loss is chosen. If the validation loss does not decrease below the minimum for 500 epochs, the training is stopped early. This is very important for a fair comparison of the models, as we found that the explanations of an overfit downstream model vary from the "best" one.

In a more recent version of the PGExplainer [**10423141**], the authors add a formal description of the downstream model layer used in their framework. The used GNN layer is defined as:

$$f(\mathbf{H}^{(l)}, \mathbf{A}) = \sigma(\mathbf{W}^{(l)}\mathbf{A}\mathbf{H}^{(l)}), \tag{4.21}$$

where A is the normalized Laplacian matrix.
Note that a PyTorch Geometric implementation of the GCN layer described in equation 2.30 is used in the replication paper [**holdijk2021re**] with a ReLU activation function. Thus, the sole difference between the two is the order of the matrix multiplication.

We use a PyTorch Geometric implementation of the Higher-order GNN layer with a ReLU activation function, as defined in Equation 2.31. On the one hand, it works on bipartite graphs by definition. On the other hand, it explicitly allows for edge weights to be passed, which is essential for "applying" the sampled, approximate discrete explanation mask to the input graph during training. This is necessary since in practice, applying a hard mask to the input graph would prevent the computation of gradients during back propagation. Thus, instead of actually removing edges of the graph, the edge importance scores that are learned by the MLP are treated as edge weights in the prediction process of the downstream model, with higher scores indicating edges being more relevant for the prediction. During training, the prediction for the original input graph is computed from the graph defined by its node features and edge-index alone, while for the sampled graph the edge weights are passed additionally.

Explainer Architecture

We implement the explainer MLP architecture described in the paper. It consists of two linear layers, with a ReLU activation function applied after the first. The first layer maps the input edge embedding dimension to 64 hidden units, and the second maps the hidden units to a single output scalar. The input edge embeddings for the MLP are calculated as described in Section 4.1.3.

For the described downstream models, the resulting MLP input - a concatenation of the node embeddings - has a shape of $\mathbb{R}^{2\cdot 20}$ for graph tasks and $\mathbb{R}^{3\cdot 60}$ for node tasks. Again, this is taken from the original codebase, as the paper does not consider the node embeddings as a concatenation of all GNN layers for node tasks.

We implement the calculation of edge embeddings as follows:

```python
    def getEdgeEmbeddings(self, modelGNN, x, edge_index, nodeToPred=None):
        emb = modelGNN.getNodeEmbeddings(x, edge_index)
        i, j = edge_index[0], edge_index[1]

        if nodeToPred is not None:
            node_emb = emb[nodeToPred].repeat(len(i), 1)
            embCat = torch.cat([emb[i], emb[j], node_emb], dim=1)
        else:
            embCat = torch.cat([emb[i], emb[j]], dim=1)
        return embCat
```

Listing 4.2: Implementation of edge embedding calculation

The trained, fixed downstream model `modelGNN` is passed to the explainer and returns its node embeddings for the input graph, defined by its node features `x` and the `edge_index`. Depending on the task at hand, the node embeddings of each two connected nodes, as well as the node embeddings of the node to be predicted `nodeToPred` in the case of a node task, are concatenated and returned as edge embeddings.

Explainer Training Specifications

We initialize all linear layer weights of the explainer with He initialization [**he2015delving**], which is used by default for linear layers with ReLU activation in PyTorch. We also test the effect of adopting the initializations from the downstream models and initializing all layer weights with Xavier [**glorot2010understanding**] (see Section 5.1.2). Biases are generally initialized with zero. During training, the Adam [**kingma2014adam**] optimizer is used to update the model parameters based on gradients. Additionally, gradients are clipped during training using PyTorch's `clip_grad_norm_` with a maximum norm of 2. The authors define the temperature used in the reparameterization trick (see Equation 4.4) with an annealing temperature schedule, as proposed by Abid et al. [**abid2019concrete**]:

$$\tau^{(t)} = \tau_0 (\frac{\tau_T}{\tau_0})^{\frac{t}{T}}, \tag{4.22}$$

where t is the current epoch and T is the total number of epochs. τ_0 and τ_T are hyper-parameters that define the initial and final temperature, respectively. The reason for this is that small temperatures tend to generate more discrete graphs, as they more closely approximate samples from the Gumbel-Softmax distribution. While this is desirable at convergence, it may hinder the optimization early in training due to a reduced gradient signal. It should be noted that in the original codebase t is initialized with 0, leading to a temperature $\tau^{(0)} = \tau_0$ in the first epoch, and $\tau^{(T-1)} = \tau_0(\frac{\tau_T}{\tau_0})^{\frac{T-1}{T}}$ in the last epoch. We initialize t with 1 to get a temperature of $\tau^{(T)} = \tau_T$ in the last epoch, as we believe this to be more inline with the proposition by Abid et al. [**abid2019concrete**].

As established in Section 4.1.3, the node prediction of an L-layer GNN is completely determined by its local computation graph, defined by its L-hop neighborhood [**ying2019gnnexplainer**]. This is capitalized on during explainer training by computing the neighborhood graphs of each input node and treating this subgraph as a base for the node predictions, rather than using the full graph (see Algorithm 1). Since the downstream models for node tasks consist of three graph layers, the 3-hop subgraph of each node is computed. We implement this using the `k_hop_subgraph()` function from PyTorch Geometric.

We compute the loss as described in Equation 4.9, with each regularization term being added according to a regularization coefficient hyperparameter. It is noteworthy that the loss function used in the original codebase can formally be defined as:

$$\min_{\Psi} -\frac{1}{K} \sum_{i \in \mathcal{I}} \sum_{k=1}^{K} -\log P_{\Phi}(\hat{Y}_s = c_i | G = \hat{G}_s^{(i,k)}), \tag{4.23}$$

where c_i denotes the class label according to the prediction on the original graph $G_o^{(i)}$ and $K = 1$. However, this is not mentioned in the paper and therefore not adopted.

A further undocumented specification is that for downstream models that perform node tasks, only node instances that belong to the "motif classes" are selected for training and evaluation, since for nodes that do not belong to these classes there does not exist a specific ground truth motif that may serve as an explanation. This is only documented for one graph classification dataset, where only one of two possible classes has a dedicated motif. The specific node sets used for each dataset vary as well and are presented in Section 5.1.

In the PGExplainer paper, Luo et al. [**luo2020parameterized**] propose the use of high temperatures, with $\tau_0 = 5.0$ and $\tau_T = 2.0$. However, Holdijk et al. [**holdijk2021re**] found that all hyperparameters described in the paper are overly simplified and in practice, different parameters are used for each dataset. Therefore, we conduct hyperparameter searches for each task in 5.1. We add that the number of sampled graphs K is not defined in the original work, and the codebase suggests the use of $K = 1$. We implement the ability to sample multiple graphs as described in the pseudocode (see Appendix A.1) and also consider this in our hyperparameter searches.

Extension description

This is not described in the original paper, but since undirected graphs contain bidirectional edges, effectively each edge carries two importance scores. The authors alleviate this in the code by symmetrizing the edge weight matrix, corresponding to the adjacency matrix of the graph. We propose meaning the logits of each pair of bidirectional edges after the MLP output has been calculated. We implement the logic of meaning each edge's logits as follows:

```
edge_pairs = edge_index.t()
# Sort node pairs so that (i, j) and (j, i) are treated the same
canonical_pairs, _ = edge_pairs.sort(dim=1)
# Find unique undirected edges and get mapping indices
unique_pairs, inverse_indices = torch.unique(canonical_pairs, dim=0)

# Average weights for duplicate edges
for i in range(unique_pairs.size(0)):
    mask = inverse_indices == i
    mean_weight = torch.mean(w_ij[mask])
    w_ij[mask] = mean_weight
```

Listing 4.3: Implementation of meaning bidirectional edge weights

All edges pairs from the edge-index that connect the same two nodes, e.g. (i, j) and (j, i), are treated as a unique pair. For each of these unique pairs the mean of the logits $\omega_{i,j}$ and $\omega_{j,i}$ corresponding to its two edges is calculated and used to update each of the two logits. To account for this in the reparameterization trick, the unique pairs are further passed there to sample ϵ for each edge pair, rather than each edge. This process guarantees edge pairs that connect the same two nodes to always carry identical importance scores.

Evaluation Implementation

To quantitatively evaluate the explanations of each prediction (see Section 5.1), the authors utilize the ROC-AUC as a metric to compare the predicted importance scores of the edges to the ground truth edges. Since the exact procedure is not further described by the authors, we extract it from the codebase as far as possible.

For graph tasks the metric is computed globally, meaning that for all graph instances the edge predictions and ground truths are gathered, and the global thresholds are computed, while for node tasks the metric is computed locally for the 3-hop subgraph of each node instance and a mean is calculated later on. Since a reason for this difference in calculation is not provided, we choose to only consider the mean of the local values, regardless of task, to get a uniform procedure. Additionally, we believe that this is more in line with the inductive setting, where individual unseen instances can be explained.

It is notable that motif nodes may be selected for explanation, where the local computation graph only consist of motif nodes and consequently only ground truth edges. Since the local ROC-AUC score is not computable for these binary cases with only one class present,

it is explicitly disregarded and skipped for these nodes in the evaluation process. Note that these nodes are not disregarded during training.
We implement this calculation using the `BinaryAUROC` from TorchEval, which operates identical to the `roc_auc_score` from scikit-learn [**pedregosa2011scikit**] in the binary case, used in PGExplainer [**luo2020parameterized**].
The qualitative explanations provided by the explainer are implemented as a NetworkX visualization of the graph instance in the case of graph tasks, or of the 3-hop subgraph of the instance node in the case of node tasks. Only the edges with the top-$2k$ importance scores are drawn, where k equals #motif-edges, since edges are bidirectional and guaranteed to carry identical weights. k can therefore be understood as a needed parameter for qualitative evaluation. TODO: Holdijk discussion about k as hyperparameter!

The inference time is measured as the time it takes to generate an explanation for any instance. This starts with the computation of node embeddings from the downstream model and ends with the reparameterization trick - the simple application of a Sigmoid to the edge logits during evaluation - that returns the edge importance scores. This process is illustrated in Figure 4.3, starting from input graph G_o and ending at the sampled graph $\hat{G}_s$.

TODO: Challenges: NECESSARY? MENTION DIFFICULTIES IN EACH SUBSECTION? Difficulties getting the code to work, as we started working with the paper only. Found that hyperparameters had to be finetuned, exact downstream model is relevant (e.g. batch norm present or not, overfit models obviously generate bad explanations, need for early stopping). Different loss used in code than in paper; we adopt the loss described in the paper. No documentation of the motif node selection/training only performed on graphs/nodes that contain gt! Imprecise about dataset description (BA-2Motif has different features than the rest of the syn datasets). No description of the AUROC calculation, had to be extracted from code (global vs local)

4.3.2 Application on NeuroSAT

In this section we describe the necessary changes for explaining predictions of NeuroSAT [**selsam2018learning**] with our PGExplainer framework. We want to stress that the NeuroSAT codebase was provided by a fellow student and only the changes mentioned in this section are part of our work. The source code for NeuroSAT is publicly available.[2]

Since NeuroSAT can be regarded as a black box MPNN in the context of our work, we only describe how it passes messages superficially. In each iteration, each clause receives messages from its neighboring literals to update its embedding. Then, each literal receives

[2]Daniel Selsam et al. “NeuroSAT” (2018). URL: `https://github.com/dselsam/neurosat`

messages not only from its neighboring clauses, but also from its complementary literal, to update its embedding. The number of iterations is set to 26 for the model we are using, which was trained in the standard manner on both satisfiable and unsatisfiable SAT problems - formulae in CNF with the goal of determining whether they are satisfiable. The downstream task of NeuroSAT can hence be understood as a graph classification task.
Besides the prediction of satisfiability, NeuroSAT returns the node embeddings of both clauses and literals at each iteration. We use these to generate edge embeddings for all edges in the biadjacency matrix as input for the explainer MLP (see Section 4.2), using the same procedure as in the previous replication study (see Section 4.3.1). Since the SAT instances are defined as biadjacency matrices, edges only exist in one direction - from literals to clauses. Thus, we dismiss the meaning of edge logits and qualitatively evaluate using the top-k edges, rather than the previous top-$2k$.

The only change we have to make is allowing the NeuroSAT forward pass to receive edge weights as a parameter. If no edge weights are passed, the forward pass behaves as usual and the biadjacency matrix contains discrete values of 0 or 1. However, when predicting sampled graphs from the explainer, we pass the sampled edge weights and multiply these with the biadjacency matrix, to receive a weighted biadjacency matrix. NeuroSAT then calculates the prediction for the weighted matrix. This change is implemented as follows:

```
    connections = torch.sparse_coo_tensor(
            indices=edges,
            values=torch.ones(problem.n_cells, device=self.device)
                    if edge_weights==None else edge_weights,
            size=torch.Size([n_literals, n_clauses])
        ).to_dense()
```

Listing 4.4: Adaptation of NeuroSAT

Usually, the `connections` tensor - a biadjacency matrix - is initialized with ones at the coordinates of the edge-index tensor `edges`, containing `problem.n_cells` edges of a SAT problem batch. However, when `edge_weights` is passed into the function, the edge weights are used as initialization, rather than ones. This can be understood as a multiplication of the edge weights with the ones representing edges.
Furthermore, the input of the explainer changes to a SAT problem instance, containing only the edge-index representation of a SAT formula, as well as a label 1 or 0, denoting satisfiability or unsatisfiability, respectively. Node features are not used in NeuroSAT and therefore irrelevant for the explainer.

The hard constraint described in 4.2 is implemented as seen in Listing 4.5.

```
    batch_clauses = torch.tensor(problem.batch_edges[:, 1])
    clauses, inverse_indices = torch.unique(batch_clauses, dim=0,
        return_inverse=True)

    for clause_id in clauses:
        mask = batch_clauses == clause_id
        clause_edge_probs = w_ij[mask]

        if len(clause_edge_probs) > 1:
            clause_mean_weight = torch.mean(clause_edge_probs)
            w_ij[mask] = clause_mean_weight

    if self.training:
        rand_vals = torch.rand(len(clauses), device=w_ij.device) + 1e-8
        epsilon = rand_vals[inverse_indices].reshape(w_ij.shape)

        trick = (torch.log(epsilon)-torch.log(1-epsilon)+w_ij)/temp
        edge_ij = nn.Sigmoid()(trick)
    else:
        edge_ij = nn.Sigmoid()(w_ij)
```

Listing 4.5: Implementation of Hard Constraint

First, all clauses are extracted from a edge index batch `problem.batch_egdes`. Since these appear in the edge index for each literal that connects to it, we need to calculate the `torch.unique` clauses. For each `clause` in these unique `clauses` we create a mask that stores its appearance in the `batch_clauses` and extract the masked edge logits `w_ij` that have the same shape as `batch_clauses`. For all clause logits `clause_edge_probs` the mean is calculated and the logits `w_ij` are updated accordingly. The unique `clauses` are further used in the reparameterization trick to sample a unique `epsilon` for each clause rather than each edge. The soft constraint uses the same idea of iterating over the unique `clauses` in a batch. Instead of calculating the `torch.mean` it calculates the `torch.var` and adds this to the loss after multiplication with a fixed connectivity coefficient hyperparameter.

To evaluate the explanations provided by the PGExplainer, we require ground truths to calculate the ROC-AUC and understand the visualizations of explanations. Since our goal is testing whether the explanations of the NeuroSAT predictions align with human-understandable concepts, we propose the use of MUSs as ground truth. Therefore, we utilize the deletion-based MUS extractor `MUSX` from PySAT [**imms-sat18**] to generate a MUS for each unsatisfiable problem. This MUS is transformed to match the corresponding edges of its graph representation and treated as expected ground truth. This allows us to calculate the local ROC-AUC for each SAT problem as described in Section 4.3.1. We visualize the provided explanations and ground truths as a pyvis [**perrone2020network**] `Network` for qualitative evaluation.

TODO: Kneed for evaluation independent of k?
We adopt the MLP architecture described in Section 4.3.1, but also perform experiments with a more complex architecture (see Section 5.2). This introduces two additional hidden layers, leading to the following architecture:

```
    self.model = nn.Sequential(
            nn.Linear(self.inputSize, 256),
            nn.ReLU(),
            nn.Linear(256, 64),
            nn.ReLU(),
            nn.Linear(64, 20),
            nn.ReLU(),
            nn.Linear(20, 1)
        )
```

Listing 4.6: Implementation of extended MLP architecture

The `inputSize` depends on whether the node embeddings $\mathbf{z}$ are used from the last iteration or if a concatenation of the states at multiple iterations shall be used. Since the embedding size of NeuroSAT is 128, this leads to an edge embedding `inputSize` of either 256 or 768, respectively. Besides these changes, the explainer operates as previously explained.

Chapter 5

Experiments and Results

In this chapter we introduce all performed experiments. This includes a general experimental setup, as well as more detailed experiment settings including their results.
Since our work is twofold, we start with the experiments regarding the replication of the PGExplainer with a changed downstream model architecture and a focus on the inductive setting in Section 5.1. Next, we describe the experiments performed on the adapted explainer framework to generate explanations for the NeuroSAT model predictions of unsatisfiable problems in Section 5.2.

5.1 Replication of PGExplainer

In this section we present all experiments regarding the replication of the presented explainer model. We first put forward the common experimental setup, including the datasets, corresponding downstream models and hyperparameter searches (see Section 5.1.1). In Section 5.1.2 we replicate the experiments in the inductive setting from the original paper with our adapted downstream models. Furthermore, in Section 5.1.3 we perform the original quantitative experiment in the collective setting for better comparability, as this was the core study of the original paper. We also include a study on the effect of using a larger set of training data, as the original proposes that only few training instances are necessary for the explainer to generalize well (see Section 5.1.4). In Section 5.1.5 we run an experiment specific to the BA-2Motif dataset, as we found that it behaves opposite to the expectations. Later, we study the effects of the specific node sets that were used in the original codebase in Section 5.1.6. Lastly, we qualitatively evaluate the explanations provided by our reimplemented model (see Section 5.1.7).

5.1.1 Common experimental setup

In this section we describe the common setup for the experiments that we perform on PGExplainer. We follow the experimental setup from the PGExplainer as closely as possible. Since the textual description refers to the setup from GNNExplainer and is lacking in some aspects, we extract the missing information from the codebase. As the hyperparameters are unclear or not comprehensible for some tasks we also draw information from the configs of the replication by Holdijk et al. [**holdijk2021re**].

Datasets

We perform the experiments on the same datasets used in the original. These were constructed by the authors similarly to the ones used in the baseline GNNExplainer. Four synthetic datasets were used for the node classification tasks. For the graph classification task the authors provide one synthetic dataset as well as the real-world dataset MUTAG. The synthetic datasets are constructed by creating a base graph and attaching motifs to random nodes of the base graph. These motifs determine the labels of the nodes or graphs, depending on the task at hand, and therefore serve as the ground truth explanations that the explainer shall detect. Statistics of each dataset can be found in Table 5.1 and a visualization in Section A.2. We will give a short description of each dataset.

Since three of the synthetic datasets use a Barabási-Albert (BA) graph as a base, we briefly introduce the BA model. The BA model generates scale-free networks that grow over time. Starting with an initialization network of $m_0 \geq m$ nodes, at each step a new node is added and connected to m of the nodes already existing in the graph. The probability for each node to be selected as a neighbor depends on its degree, leading to a higher probability for nodes that already have a high degree rather than nodes with a low degree [**albert2002statistical**].

BA-Shapes is the first node dataset that consists of a single BA-graph with 300 nodes and 80 "house" motifs - five nodes resembling the shape of a house (see 5.1a). Base graph nodes are labeled with 0 while nodes at the top/middle/bottom of the "house" are labeled with 1,2,3, respectively. The top node of each house motif is attached to a random base graph node. Additional edges are added for perturbation. Each node is assigned a 10-dimensional feature vector of 1s.
BA-Community consists of two unified BA-Shapes graphs. The features of the nodes are sampled from two Gaussian distributions. Nodes are labeled as in BA-Shapes for each community respectively, leading to 8 classes in total.
Tree-Cycles uses an 8-level balanced binary tree as a base graph. 80 cycle motifs, consisting of a 6 node cycle (see 5.1b), are attached to random nodes from the base graph. Node features are assigned as a 10-dimensional vector of 1s. A node of the base graph is labeled as 0 and a motif node is labeled as 1.
The Tree-Grid dataset is assembled in the same way as Tree-Cycles, with the difference that the motifs are 3-by-3 grids (see 5.1c). Node features and labels also follow the same procedure. BA-2Motif is the first graph dataset with 800 graphs. Each of these graphs is obtained by attaching either a "house" or a cycle as a motif to a base BA graph with 20

nodes. According to the attached motif the graphs are assigned one of two labels, with 0 or 1 implying a house or circle, respectively.
The real-world dataset MUTAG contains $4,337$ molecule graphs that are assigned to one of 2 classes, depending on the molecules mutagenic effect [**riesen2008iam**], [**ying2019gnnexplainer**]. Node features are assigned as a one-hot encoding in $\{0,1\}^{14}$, representing the chemical group of a node out of 14 possible ones. Following [**debnath1991structure**], [**ying2019gnnexplainer**], carbon rings with chemical groups NH_2 or NO_2 are known to be mutagenic, with carbon rings in general existing in both mutagenic and non-mutagenic graphs. The authors thus propose treating the carbon ring as a shared base graph and NH_2 and NO_2 as motifs for mutagenic graphs (see 5.1d). Since there are no explicit motifs for the non-mutagenic graphs, these graphs are not considered in PGExplainer.

	BA-Shapes	**BA-Community**	**Tree-Cycles**	**Tree-Grid**	**BA-2motifs**	**MUTAG**
#graphs	1	1	1	1	1,000	4,337
#nodes	700	1,400	871	1,231	25,000	131,488
#edges	4,110	8,920	1,950	3,410	51,392	266,894
#labels	4	8	2	2	2	2

Table 5.1: Dataset statistics for Node and Graph Classification tasks, reprinted from [**luo2020parameterized**].

Note that in the collective setting used in the original paper the explainer is trained and evaluated on the same data. This data is further reduced by only using graphs and nodes that contain a ground truth motif. This makes sense for evaluation, since the AUROC cannot be calculated for ground truths with only one class present. However, the authors do not specify why the training is performed only on these instances. Therefore, only the 1,015 mutagenic graphs where either NH_2 or NO_2 are present are selected for the MUTAG experiment.

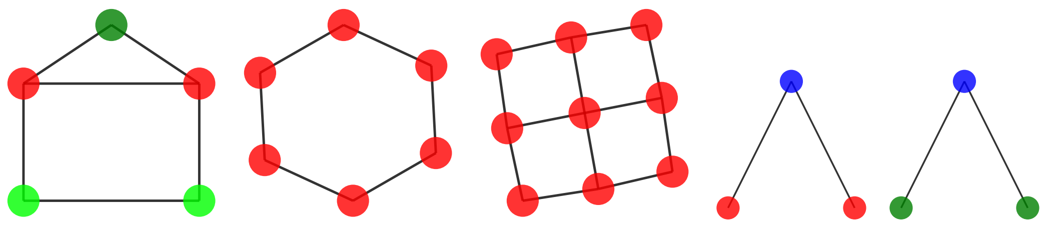

(a) House motif (b) Circle motif (c) Grid motif (d) NO_2 and NH_2 motifs

Figure 5.1: The different motifs used in the datasets.

In the node classification experiments the instance sets $\mathcal{I}$ containing the nodes to be explained, used for training and evaluation, were further finetuned per dataset. This leads to a selection of either all nodes that are part of a motif, or only one node per motif. This inconsistency is also left unexplained by the authors and extracted from the codebase. It is

likely adapted from the GNNExplainer [**ying2019gnnexplainer**], where single-instance and multi-instance explanations are differentiated. We perform an experiment on the effects of these node selections in 5.1.6.
For the instance set $\mathcal{I}$ used in BA-community multiple configurations exist in the PG-Explainer codebase. One is extracted from the GNNExplainer [**ying2019gnnexplainer**] and consists of the same nodes used for the BA-Shapes dataset, which seems unjustified and is not addressed in the paper. Another setting uses all motif nodes of both communities - effectively all nodes that do not belong to the two community base graphs: TODO: THIS IS UNECESSARY!?

```
    motifNodes = [i for i in range(data.y.shape[0])
        if data.y[i] != 0 and data.y[i] != 4]
```

Listing 5.1: Selection of BA-Community instances

`data.y` contains the labels of all nodes in the graph. We select this setting for our replication, as this makes more sense than using the arbitrary indices from BA-Shapes.

The selection of instances for the remaining dataset is as follows: In BA-Shapes one "middle" node from each motif in the graph is selected as instance. A similar selection is used in Tree-Cycles, where the "first" motif node connected to the base graph is chosen for each motif. For Tree-Grid all motif nodes in the graph are selected as instances, similar to BA-Community. For the graph dataset BA-2Motif all graphs are used as instances, since each one explicitly contains one of the two motifs. The consequential number of instances N used for each dataset are listed in Table 5.2. Note that these instance sets are only considered in the explainer and not during downstream model training.

	BA-Sh.	**BA-Co.**	**Tree-Cy.**	**Tree-Gr.**	**BA-2M.**	**MUTAG**
#total motif instances	300	800	360	289	1,000	1,015
#used motif instances N	60	800	60	289	1,000	1,015

Table 5.2: Amount of possible and actually used motif instances per dataset, as found in the original codebase.

The fixed downstream model that we use for each dataset is trained as described in Section 4.3.1. Since we use a different GNN layer, we try to achieve accuracies above 85%, similar to the original. The accuracies can be seen in Table 5.3 and are slightly higher than in the original for four of the datasets, except for BA-Community and MUTAG. We add a dropout layer with a probability of $p = 0.1$ to the latter two models to improve their generalizability and push their testing performance closer to the original. The exact downstream task accuracies achieved in PGExplainer [**luo2020parameterized**] can be seen in Table A.1.

Accuracy	BA-Shapes	BA-Community	Tree-Cycles	Tree-Grid	BA-2Motif	MUTAG
Training	1.00	0.92	1.00	0.99	1.00	0.86
Validation	1.00	0.87	1.00	0.99	1.00	0.86
Testing	0.97	0.89	0.99	0.99	1.00	0.82

Table 5.3: Accuracy table for Node and Graph Classification downstream tasks from our reimplementation using the higher-order GNN layer.

Evaluation Metrics

Following Luo et al. [**luo2020parameterized**], the explanation problem is considered a binary classification of edges for quantitative evaluation. Edges that appear in a motif are positive targets, and negative otherwise. The edge importance weights of each explained instance are treated as prediction scores.
Since the original paper does not specify the exact calculation of the AUROC score, we use the approach described in Section 4.3.1 for a standardized procedure. Thus, we conduct the mean over the local AUROC scores of all set instances as the metric for quantitative evaluation. Node instances with a local computation graph of only motif nodes are skipped in the calculation. We calculate the mean local AUROC on the training-, validation and test sets individually. The validation set is used to evaluate the hyperparameter search, while the results of the following experiments usually consider the score on the test set.

To qualitatively evaluate the explanations we visualize the top-k explanation mask edges with the highest prediction scores, following Luo et al. [**luo2020parameterized**]. The parameter k for each dataset is extracted from the original codebase, and is set to include at least the number of ground truth motif edges in the explanation. For node classification tasks the visual explanations only include the nodes inside the local computation graph of the prediction node, as only these are relevant for the prediction [**ying2019gnnexplainer**].

We perform the efficiency evaluation as described by Holdijk et al. [**holdijk2021re**]. The average inference time needed to generate an explanation for any instance is calculated over the test set for each run individually (see Section 4.3.1). We include the mean of these averages for the inductive replication experiment only.

Hyperparameter Search

Most details on the training procedure of the explainer have been established in Section 4.3.1. As found by Holdijk et al. [**holdijk2021re**] the PGExplainer is very sensitive to hyperparameter settings on each dataset. Therefore, we conduct hyperparameter searches for the explainer model on each of the datasets to obtain the best performing explainers. We follow Liashchynskyi and Liashchynskyi [**liashchynskyi2019grid**] to perform grid searches over the parameter space that we define as an extended combination of the

setting used in the original [**luo2020parameterized**], as well as the configs provided in Replication study [**holdijk2021re**].
For our hyperparameters $\lambda_1, \lambda_2, ..., \lambda_n$ we define corresponding sets $S_1, S_2, ..., S_n$ of possible values. The grid search finds the best model with respect to the mean of the local ROC-AUC scores of all validation instances over all combinations $(\lambda_1, \lambda_2, ..., \lambda_n) \in S_1 \times S_2 \times ... \times S_n$. Additionally, we consider that the predicted edge importance scores shall not be exactly 0 or 1 for all edges. Since we found that some datasets behave unexpectedly regarding the evaluation metric, we may choose to optimize in the opposite direction, which we will further discuss in Section 5.1.2. All experiments and training were conducted on an AMD Ryzen 7 2700X processor.

The hyperparameters tested consist of the learning rate $\eta \in \mathbb{R}^+$, the number of epochs $E \in \mathbb{N}$ used to train the explainer, the number of sampled graphs $K \in \mathbb{N}$, the initial and final temperatures $\tau_0, \tau_T \in \mathbb{R}^+$, as well as two coefficients $\alpha_e \in \mathbb{R}^+$ and $\alpha_s \in \mathbb{R}^+$ to control the entropy regularization and the size regularization, respectively. For the BA-Community explainer we also test a sample bias $b = 0.5$, that restricts the ϵ in Equation 4.4 to $\epsilon \sim \text{Uniform}(0 + b, 1 - b)$. This is also extracted from the original codebase and leads to a constant $\epsilon = 0.5$. We further define a set of fixed seeds used during each grid search as $S = \{74, 75, 76\}$, since we found that the performance of the explainer is highly dependent on the seed for some of the datasets.

Since we care about the inductive performance and Luo et al. [**luo2020parameterized**] demonstrate that the explainer performs well on few training instances, we set the number of training instances $a = 30$ for graph tasks and $a = 0.08$ for node tasks during the searches. We choose a percental split for the node tasks, since the node sets used in the original experiment highly vary in size and generally contain fewer instances than the graph sets. The resulting absolute number of training instances for the node tasks, as well as the specifics of each search can be seen in Section A.3. Following PGExplainer [**luo2020parameterized**], the resulting validation set and test set each have a size of $\frac{(N-a)}{2}$, where N denotes the number of used instances in each dataset. The optimal hyperparameter values used for the following experiments are listed in Table 5.4.

Dataset	K	b	E	η	α_e	α_s	τ_0	τ_T
BA-Shapes	1	0.0	10	0.003	0.1	0.05	5.0	1.0
BA-Community	5	0.0	20	0.003	1.0	0.1	1.0	5.0
Tree-Cycles	5	0.0	20	0.0003	1.0	0.0001	1.0	1.0
Tree-Grid	5	0.0	30	0.003	1.0	0.5	5.0	2.0
BA-2Motif	10	0.0	20	0.01	0.1	0.03	5.0	1.0
MUTAG	10	0.0	20	0.01	1.0	0.005	5.0	1.0

Table 5.4: The best-performing parameter values for each dataset based on the performed grid searches (see Section A.3).

Baselines

We compare our work to both the collective and inductive results from the original PGExplainer paper, as well as the results from the collective PyTorch replication study by Holdijk et al. [**holdijk2021re**] (see Table 5.5). Since the inductive results of PGExplainer are only provided as plots, we approximate the numeric values.

	Node Classification				**Graph Classification**	
Method	BA-Shapes	BA-Community	Tree-Cycles	Tree-Grid	BA-2Motif	MUTAG
			Explanation AUROC			
PGExplainer	0.963±0.011	0.945±0.019	0.987±0.007	0.907±0.014	0.926±0.021	0.873±0.013
RE-PGExplainer	0.999±0.000	0.825±0.040	0.760±0.014	0.679±0.008	0.133±0.046	0.843±0.084
PGExplainer (inductive)	∼0.98	∼0.99	∼0.99	∼0.88	∼0.84	-
			Inference Time (ms)			
PGExplainer	10.92	24.07	6.36	6.72	80.13	9.68
RE-PGExplainer	3.58	5.23	0.45	0.54	0.33	2.05

Table 5.5: PGExplainer performance baselines.

Experimental Protocol

We follow the general experimental setup used in the baselines [**luo2020parameterized**] [**holdijk2021re**], originally proposed in [**ying2019gnnexplainer**]. We train a PGExplainer model on a fixed downstream model with a specified seed, with the hyperparameter values listed in Table 5.4. This process is repeated 10 times for each dataset. The models are evaluated quantitatively using the AUROC metric and qualitatively for select experiments, in the sense of comparing visualizations of explanations to the ground truth motifs. We provide the quantitative results as the mean and standard deviation over 10 runs for all experiments.

5.1.2 Inductive Setting

Experimental Setup

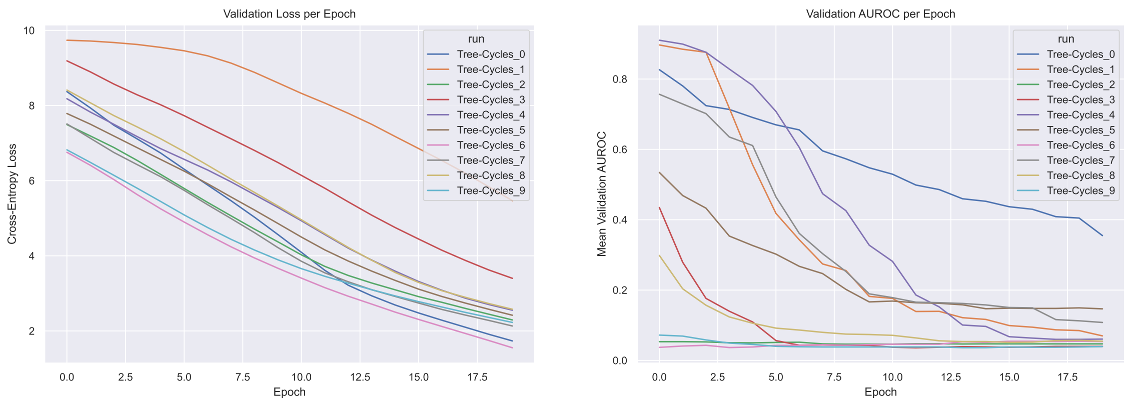
In this experiment we evaluate the quantitative performance of PGExplainer in the inductive setting on our slightly adapted downstream models. Since Luo et al. [**luo2020parameterized**] showed that the PGExplainer can achieve good results with only few training instances, we aim to replicate this for $a = 30$ training instances, one of the original experiment settings. Since the exact instances used are not further specified in the paper, we treat the motif node sets described in Section 5.1.1 as the instance sets of size N for node task.
We repeat this experiment with Xavier [**glorot2010understanding**] used as initialization for the explainer MLP weights, opposed to the He [**he2015delving**] initialization standard to PyTorch linear layers with ReLU activations. This is done since the grid searches showed

that the performance is heavily dependent on the used seed for some datasets, which mainly controls the initializations. Xavier is used by default in the original TensorFlow [**tensorflow2015-whitepaper**] implementation.

Additionally, we measure the average inference time to explain any instance and compare it to both baselines.

Results

Table 5.6 shows the quantitative results, as well as the inference time for all dataset. First, BA-Shapes is the only experiment that comes close to the originally recorded scores. It is notable, that both Tree-Cycles and BA-2Motif achieve an AUROC score far below 0.5, indicating that the opposite of the ground truth expectations is predicted. Though the variance of Tree-Cycles is quite high, the figures 5.2a and 5.2b show that this is mostly cause by a singular outlier run. Both the losses and AUROC scores are decreasing steadily, illustrating a stable learning process.



(a) Mean validation Loss **(b)** Mean validation AUROC

Figure 5.2: Validation metrics for Tree-Cycles in inductive setting.

We were unable to determine a reason for the unexpected opposite classification of select tasks, but highlight that this can also be observed for BA-2Motif in the replication study [**holdijk2021re**] (see Table 5.5). To validate this, we perform an experiment with flipped ground truth edges in Section 5.1.5.

Considering the flipped AUROC scores indicate learning, Tree-Grid is the worst performing dataset, converging to a metric score close to random guessing. We note that the loss does converge as expected in all runs, but the model seems to be unable to generalize well.

It is notable that the Xavier initialized model achieves worse results across the board. Though the mean values are higher for most of the datasets, the variance is also much higher, suggesting less stability. Since we care for stable models we only consider He initialization for the following experiments.

Method	Node Classification				Graph Classification	
	BA-Shapes	BA-Community	Tree-Cycles	Tree-Grid	BA-2Motif	MUTAG
	Explanation AUROC					
PGExplainer	∼0.98	∼0.99	∼0.99	∼0.88	∼0.84	-
Our work (He)	0.994±0.001	0.754±0.013	0.106±0.104	0.537±0.081	0.017±0.006	0.874±0.009
Our work (Xavier)	0.995±0.002	0.779±0.036	0.467±0.316	0.551±0.119	0.035±0.05	0.834±0.04
	Inference Time (ms)					
PGExplainer	10.92	24.07	6.36	6.72	80.13	9.68
RE-PGExplainer	3.58	5.23	0.45	0.54	0.33	2.05
Our work	37.0±1.4	24.8±0.1	3.0±0.2	2.7±0.1	4.0±0.3	4.0

Table 5.6: Inductive performance of PGExplainer ($a = 30$).

Our implementation improves the inference time of the explainer for 4 datasets, but is notably slower for BA-Shapes. We report considerably lower times than the replication study, which we primarily attribute to our limited hardware.

5.1.3 Collective Setting

Experimental Setup

To allow for a better comparison with both baselines we perform an experiment in the collective setting. Therefore, both the training and evaluation are performed on the complete set of motif instances. This is the setting that can be found in the original codebase. It is noteworthy that we use the hyperparameters found in an inductive setting (see Section 5.1.1), which should allow for good generalization. We also include our results obtained in the inductive setting, to evaluate the difference between the settings.

Results

We are unable to reproduce the scores achieved in the original work for most of the datasets, as seen in Table 5.7. BA-Shapes achieves a higher score than the original, while MUTAG comes close to the original baseline and even obtains a higher score in the inductive setting. The other scores do not come close when considering the scores as they are without flipped ground truths. We are however able to match the results achieved in the replication baseline in all but the Tree-Cycles experiment. This validates that the PGExplainer achieves similar results for different downstream GNN layers.

5.1.4 Inductive Setting with more training data

Experimental Setup

Luo et al. [**luo2020parameterized**] show that the inductive performance of PGExplainer improves with the number of instances used for training. To validate this we perform an

Method	**Node Classification** BA-Shapes	BA-Community	Tree-Cycles	Tree-Grid	**Graph Classification** BA-2Motif	MUTAG
	Explanation AUROC					
PGExplainer	0.963±0.011	0.945±0.019	0.987±0.007	0.907±0.014	0.926±0.021	0.873±0.013
RE-PGExplainer	0.999±0.000	0.825±0.040	0.760±0.014	0.679±0.008	0.133±0.046	0.843±0.084
Our work	0.993±0.001	0.831±0.009	0.084±0.078	0.679±0.032	0.018±0.004	0.833±0.006
Our work (inductive)	0.994±0.001	0.754±0.013	0.106±0.104	0.537±0.081	0.017±0.006	0.874±0.009

Table 5.7: Collective PGExplainer performance ($a = N$).

inductive experiment with an increased number of training instances $a = 60$. Since the motif instance sets of BA-Shapes and Tree-Cycles in the original setting only consist of 60 instances, we use a split ratio of 80/10/10. This leads to $a = 48$ for these two tasks.

Results

The results for this experiment are found in Table 5.8. The graph tasks see a slight improvement, while the node tasks all seem to perform worse with more training instances. While the variance decreases for both tree tasks, the mean moves toward the value indicating random predictions. We observe that BA-Community reaches a peak in AUROC early but steadily decreases afterward, as opposed to the setting with $a = 30$ where the AUROC rises steadily and converges strongly. This result suggests that the hyperparameters have to be retuned for this setting, which hinders the generalizability of the explainer. (TODO: THIS NEEDS PLOTS!?!?) We are unable to confirm that the PGExplainer generally achieves better results when more instances are seen during training.

Method	**Node Classification** BA-Shapes	BA-Community	Tree-Cycles	Tree-Grid	**Graph Classification** BA-2Motif	MUTAG
	Explanation AUROC					
Our work (inductive)	0.994±0.001	0.754±0.013	0.106±0.104	0.537±0.081	0.017±0.006	0.874±0.009
$a = 60$	0.987±0.002	0.641±0.045	0.144±0.096	0.495±0.052	0.017±0.004	0.895±0.009

Table 5.8: Inductive PGExplainer performance with more training instances where a is either 48 or 60.

5.1.5 BA-2Motif with Inverted Ground Truth

Experimental setup

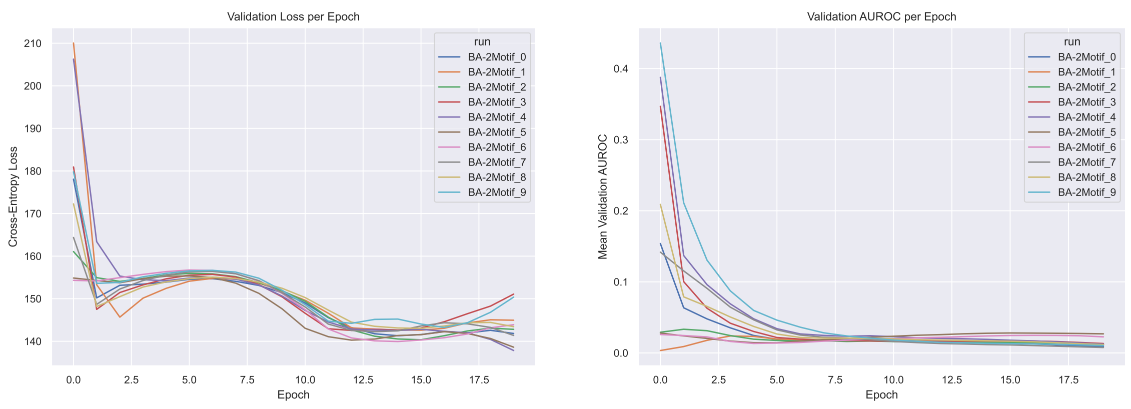
Since we found that the AUROC score for BA-2Motif decreases rather than increases over the training epochs, while converging near zero, we run an additional experiment with inverted GT. Since both the loss and AUROC curves decrease steadily and flatten in the original inductive experiment (see Figure 5.3), we consider this experiment an additional validation. The mean individual AUROC is calculated identical to before, but the ground truth mask of each graph is inverted, meaning that edges in the motif now carry a label of 0 and all other edges a label of 1. Note that the ground truth does not affect the training procedure in any way and merely changes the metric evaluation.

Method	**Explanation AUROC**
PGExplainer	0.926±0.021
RE-PGExplainer	0.133±0.046
Our work	0.017±0.006
Inverted GT	0.985±0.006

Table 5.9: Explanation AUROC for BA-2Motif with inverted GT.

Results

As expected, the AUROC score for BA-2Motif is nearly perfect in this experiment, even surpassing the original baseline, as seen in Table 5.9. It is notable that the results by Holdijk et al. [**holdijk2021re**] suggest a similar observation, however the authors do not expand on this.



(a) Mean validation Loss **(b)** Mean validation AUROC

Figure 5.3: Validation metrics for BA-2Motif in inductive setting.

5.1.6 Effects of used Motif Instances for Node Tasks

Experimental Setup

As presented in 5.1.1 the motif instances used in PGExplainer [**luo2020parameterized**] for different node level explanation tasks do not follow the same rules. This is likely related to their direct comparison to GNNExplainer [**ying2019gnnexplainer**] and the adaptation of their dataset settings. However, since the PGExplainer claims to generalize well and be able to provide explanations for any downstream GNN task, a uniform procedure for

providing explanations is necessary. Evidently, different nodes in a motif have different local compuation graphs, containing different motif- and non-motif nodes. The 3-hop graph computed from a corner node of a tree-grid may for example never contain the complete motif. Therefore, we evaluate the effect of training on all motif nodes or a singular node from each motif.
We suggest changing the instance sets $\mathcal{I}$ for training and evaluation to include either all or one select motif node, depending on the dataset. We use the same hyperparameters used for the previous experiments. Besides this change, we follow the general experimental setup in the inductive setting with $a = 30$ to evaluate whether a reasoning behind the node selection can be made out.

Since BA-Shapes and Tree-Cycles originally use a singular node from each motif, we evaluate the effect of using all nodes in each motif and refer to this as All-Motif-Nodes. Accordingly, we select one fixed node from each motif for the BA-Community and Tree-Grid tasks, similar to the original BA-Shapes and Tree-Cycles task. For BA-Community we choose one of the two "middle" nodes in the house motif, similar to BA-Shapes. For Tree-Grid we select the "second" node, starting from the corner node attached to the base graph, as the 3-hop graph from this node contains all motif nodes as well as the most nodes from the base graph. We call this experiment One-Motif-Node. The exact number of instances used in this experiment can be seen in Table 5.10.
TODO: It is noteworthy that using singular corresponding topological nodes across motifs may trivialize the problem in terms of generalizability.

	All-Motif-Nodes		**One-Motif-Node**	
	BA-Sh.	**Tree-Cy.**	**BA-Co.**	**Tree-Gr.**
#total motif instances	300	360	800	289
#used motif instances N	300	360	160	32

Table 5.10: Amount of possible and actually used motif instances in the experiments with adapted instance sets.

Results

We present the results of the All-Motif-Nodes experiment compared to our results achieved in the inductive setting in Table 5.11 and for One-Motif-Node in Table 5.12. It is most notable that the Tree-Grid experiment achieves perfect edge classification accuracy in the One-Motif-Node experiment. BA-Community also reports a significant increase in AUROC score.
The All-Motif-Nodes experiment leads to a slight deterioration in score for BA-Shapes, which is still better than the scores achieved on the other node tasks in our inductive experiment. The explainer's performance on Tree-Cycles deteriorates in both variance and mean score, particularly in terms of its distance from a random AUROC score.

	Explanation AUROC	
Method	BA-Shapes	Tree-Cycles
Our work	0.994±0.001	0.106±0.104
AllMotifNodes	0.959±0.004	0.204±0.162

Table 5.11: Explanation AUROC for All-Motif-Nodes.

	Explanation AUROC	
Method	BA-Community	Tree-Grid
Our work	0.754±0.013	0.537±0.081
OneMotifNode	0.951±0.007	1.0±0.0

Table 5.12: Explanation AUROC for One-Motif-Node .

5.1.7 Qualitative Analysis

Experimental setup

For the qualitative analysis the best performing explainer model for each downstream model is selected, and random explanations are sampled. For the explainer models with AUROC scores below 0.5, we select the model that achieves the lowest score and visualize the top-k lowest edges accordingly. Following the approach of the original, we present instance explanations that highlight the successful detection of motifs. We additionally include sample grids of 16 random graphs in the Appendix.

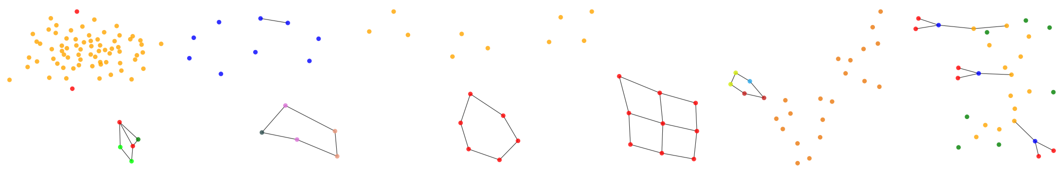
Following Luo et al. [**luo2020parameterized**], we select dataset-specific values of k: BA-Shapes, BA-Community and Tree-Cycles use $k = 6$; Tree-Grid uses $k = 12$; BA-2Motif uses $k = 5$ and MUTAG uses $k = 10$.

Results

The visualized explanations generated for each downstream model show that the ground truth motifs are detected (see Figure 5.4). We found that the MUTAG explainer detects the chemical combinations that cause the mutagenicity as the highest edges regularly, but the shared base carbon ring is not detected at all. More qualitative samples for each dataset can be seen in Appendix A.4. The qualitative explanations of BA-Community rarely contain all connected motif edges, as seen in Figure A.3. Note that due to the setting of $k = 5$ for BA-2Motif the house motif is never detect completely, but it is clearly visible that the added house edge is detected.

5.1.8 BA-2Motif with wrong features

- Effects of using ”wrong” node features for downstream task?



(a) BA-Sh. **(b)** BA-Co. **(c)** Tree-Cy. **(d)** Tree-Gr. **(e)** BA-2M. **(f)** MUTAG

Figure 5.4: Explanations for one instance of each dataset.

Comparison to orginal one: Original dataset transformed to pytorch performs way worse, for features of 0.1! Mean AUC of about 0.4!
Original dataset with features changed to ones instead of 0.1: Works good as well.

5.2 PGExplainer applied to NeuroSAT

In this section we present the experiments for generating bipartite explanations for NeuroSAT [**selsam2018learning**] predictions of unsatisfiable SAT problems. We describe the general procedure in Section **??**, followed by the results using the PGExplainer with a soft constraint in Section **??**, as well as the hard constraint experiment in Section **??**.

5.2.1 Common Experimental Setup

NeuroSAT [**selsam2018learning**] uses a distribution $\mathbf{SR}(n)$ that creates pairs of SAT problems on n variables, which are satisfiable and unsatisfiable, only differing in a singular negated literal. These are created by randomly sampling subclauses of k variables, where each variable is negated with a probability of 50%. These subclauses are added to the problem until a SAT solver [**een2003extensible**] finds it to be unsatisfiable. We adapt this to only contain singular unsatisfiable problems, rather than pairs of problems, since unsatisfiable cores only appear in unsatisfiable problems. The difficulty of the problems is slightly reduced by using $n = 8$, to allow for the explainer to run on NeuroSAT on our limited hardware. We create a dataset of 100 problems.
We create the expected ground truth with the deletion-based MUS extractor `MUSX` from PySAT [**imms-sat18**], as described in section 4.3.2.
The quantitative evaluation is adopted from Section 5.1.1, with the positive targets being the edges that appear in the corresponding ground truth MUS of each problem. Since the qualitative evaluation relies on knowing k, we calculate this dynamically during evaluation from the ground truth of the current problem.
The NeuroSAT model trained on a $\mathbf{SR}(40)$ distribution was provided by a fellow student. It achieves an accuracy of 1.0 on the dataset we created.
We apply PGExplainer in the inductive setting, using a 50/25/25 split for training, evaluating and testing the NeuroSAT explainer.

Hyperparameter Search

We perform a hyperparameter search for both the explainer with the soft constraint, and the explainer with the applied hard constraint. We follow the general grid search procedure used for the previous experiments, but introduce 4 new hyperparameters, 1 exclusive to the hard constraint and 2 exclusive to the soft constraint. $\alpha_{arch} \in \{0, 1\}$, exclusive to the hard constraint, denotes whether a more complex MLP architecture is used in the explainer. The second introduced hyperparameter $\alpha_{concat} \in \{0, 1\}$ controls whether the node embeddings $\mathbf{z}$ are used from the last iteration of NeuroSAT or if a concatenation of the states at multiple iterations shall be used (see Section 4.2). For the soft constraint, we also try using the AdamW [**loshchilov2017decoupled**] optimizer from PyTorch with an internal weight decay of 0.01, denoted by $\alpha_{AdamW} \in \{0, 1\}$. Lastly, the coefficient $\alpha_c \in \mathbb{R}^+$ controls the effect of the soft constraint R_C (see Equation 4.15) on the loss. This is disregarded in the hard constraint experiment, as the hard constraint replaces the soft constraint. The sample bias b used in PGExplainer is disregarded completely. We evaluate the search with respect to the validation AUROC score, as well as the validation loss. TODO: Also consider highest individual local AUROC

5.2.2 Soft Constraint Experiment

Experimental Setup

In this experiment we apply the soft constraint introduced in Section 4.2 to the PGExplainer to generate explanations for the predictions of NeuroSAT on unsatisfiable SAT problems. After performing a grid search specified in Appendix **??**, we utilize the optimal parameters to evaluate the test AUROC over 10 seeded runs. Additionally, we provide qualitative explanations in comparison to the ground truth MUS for the problem that results in the highest individual AUROC.

Results

We found that the grid search resulted in a maximum validation AUROC of 0.5248 and a minimal score of 0.4726. All runs achieved results in this interval, indicating completely random edge predictions. It seems that regardless of the tested hyperparameters the general explanations of NeuroSAT predictions provided by PGExplainer do not align with unsatisfiable cores.

Low size reg highest importance, followed by high consistency reg and low lr. Other parameters not as decisive. We select the parameters that maximize the validation AUROC. Seed 4 achieves the highest individual test AUROC of 0.711! (See visualization). Only 2 edegs are identified correctly, no full clauses.

5.2.3 Hard Constraint Experiment

Experimental Setup

In this experiment we apply the hard constraint introduced in Section 4.2 to the PGExplainer to generate explanations for the predictions of NeuroSAT on unsatisfiable SAT problems. The experimental setup is adopted from Section **??**.

Hyperparameter search

We found that the validation AUROC tends to decrease during training and therefore focus on minimizing the validation AUROC as well as the validation loss in the grid search. The search revealed that none of the parameter configurations achieves an AUROC score above 0.5 and below 0.44. Since 0.5 equals pure randomness and is usually caused by all edges being predicted with a score of 0, we select the model that achieves the lowest score, thus departing from the point of randomness. We found that a low learning rate is important to keep the edge predictions from reaching 0, as these steadily decrease over training time. $\alpha_s = 1.0$ further enhanced this unwanted behavior. Using a concatenation of the embeddings at multiple iterations achieved worse results than the usual graph task procedure when considering the highest individual AUROC of a validation instance. However, this does not consider whether a general procedure is learned.

Results

We found that running the training for 500 epochs resulted in a converging loss, as well as a converging metric score, close to 0.46. A local score minimum is reached in epoch 60 with a value of 0.44. Since this is very close to a score indicating randomness, we conclude that NeuroSAT does in fact not learn generalized unsatisfiable cores in the form of MUSs. However, this mostly indicates that the explainer predictions do not align with the selected ground truth (IF VARIANCE IN EXPERIMENT RUNS LOW!)
Seed 0 achieves the highest individual test AUROC of 0.776! (See visualization). Notable the 2(?) clauses are predicted correctly, but not still not good result out of (?). Show that prediction is not unsatisfiable (Results)! Besides that, mean test AUROC see Table.

Method	**AUROC**
Soft Constraint	0.512±0.023
Hard Constraint	0.528±0.013

Table 5.13: Explanation AUROC for NeuroSAT predictions of unsatisfiable SAT problems.

Chapter 6

Discussion

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.
WHAT? Maybe works well because if all motif nodes are used, subgraphs are used where only the motif or even only part of the motif is present (Tree-Grid). Thus constantly using one singular same node that contains the complete motif may be better?

6.0.1 Generalizability of PGExplainer

6.0.2 Explanations for UNSAT predictions of NeuroSAT

Chapter 7

Conclusion

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

Appendix A

Supplementary Material

A.1 Material from original PGExplainer

Algorithm 1 Training Algorithm for Explaining Node Classification from [**luo2020parameterized**].

```
Require: Input graph $G_o = (\mathcal{V}, \mathcal{E})$, node features $X$, node labels $Y$, set of instances
    to be explained $\mathcal{I}$, trained GNN model: $\mathrm{GNNE}_{\Phi_0}(\cdot)$ and $\mathrm{GNNC}_{\Phi_1}(\cdot)$, parameterized
    explainer MLP $\Psi$.
 1: for each node $i \in \mathcal{I}$ do
 2:   $G_o^{(i)} \leftarrow$ extract the computation graph for node $i$.
 3:   $Z^{(i)} \leftarrow \mathrm{GNNE}_{\Phi_0}(G_o^{(i)}, X)$.
 4:   $Y^{(i)} \leftarrow \mathrm{GNNC}_{\Phi_1}(Z^{(i)})$.
 5: end for
 6: for each epoch do
 7:   for each node $i \in \mathcal{I}$ do
 8:     $\Omega \leftarrow$ latent variables calculated with (10).
 9:     for $k \leftarrow 1$ to $K$ do
10:       $G_s^{(i,k)} \leftarrow$ sampled from (4).
11:       $\hat{Y}_s^{(i,k)} \leftarrow \mathrm{GNNC}_{\Phi_1}(\mathrm{GNNE}_{\Phi_0}(G_s^{(i,k)}, X))$.
12:     end for
13:   end for
14:   Compute loss with (9).
15:   Update parameters $\Psi$ with backpropagation.
16: end for
```

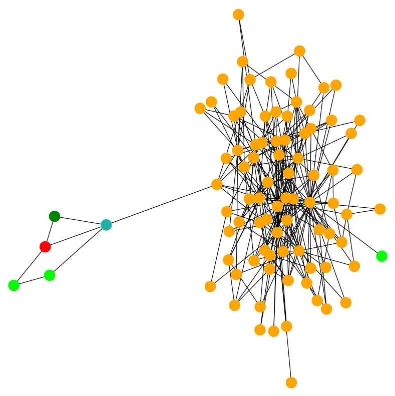
Algorithm 2 Training Algorithm for Explaining Graph Classification from [**luo2020parameterized**].

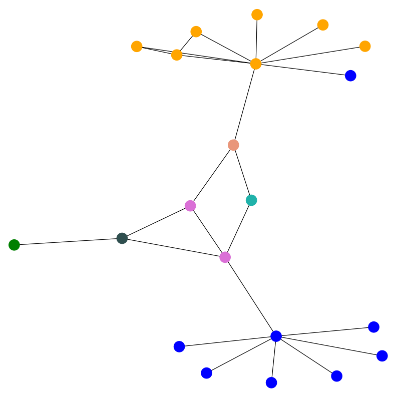
```
Require: A set of input graphs with $i$-th graph represented by $G_o^{(i)}$, node features $X^{(i)}$,
    label $Y^{(i)}$, trained GNN model: $\mathrm{GNNE}_{\Phi_0}(\cdot)$ and $\mathrm{GNNC}_{\Phi_1}(\cdot)$, parameterized explainer
    MLP $\Psi$.
 1: for each graph $G_o^{(i)}$ do
 2:     $Z^{(i)} \leftarrow \mathrm{GNNE}_{\Phi_0}(G_o^{(i)}, X^{(i)})$.
 3:     $Y^{(i)} \leftarrow \mathrm{GNNC}_{\Phi_1}(Z^{(i)})$.
 4: end for
 5: for each epoch do
 6:     for each graph $G_o^{(i)}$ do
 7:         $\Omega \leftarrow$ latent variables calculated with (11).
 8:         for $k \leftarrow 1$ to $K$ do
 9:             $G_s^{(i,k)} \leftarrow$ sampled from (4).
10:             $\hat{Y}_s^{(i,k)} \leftarrow \mathrm{GNNC}_{\Phi_1}(\mathrm{GNNE}_{\Phi_0}(G_s^{(i,k)}, X^{(i)}))$.
11:         end for
12:     end for
13:     Compute loss with (9).
14:     Update parameters $\Psi$ with backpropagation.
15: end for
```

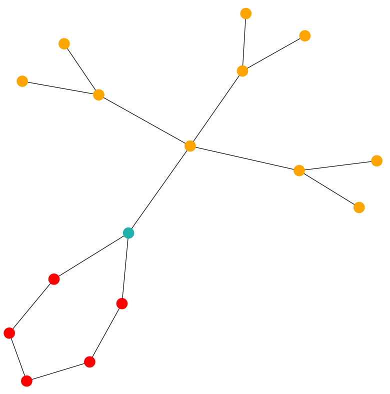
Accuracy	**BA-Shapes**	**BA-Community**	**Tree-Cycles**	**Tree-Grid**	**BA-2Motif**	**MUTAG**
Training	0.98	0.99	0.99	0.92	1.00	0.87
Validation	1.00	0.88	1.00	0.94	1.00	0.89
Testing	0.97	0.93	0.99	0.94	1.00	0.87

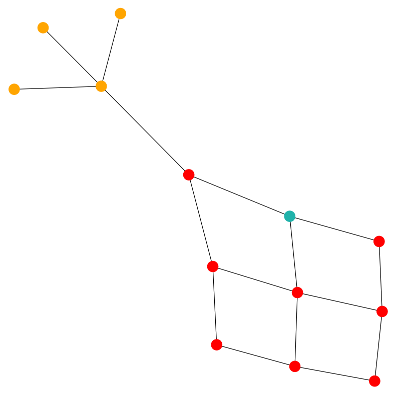
Table A.1: Compact accuracy table for Node and Graph Classification. Reprinted from [**luo2020parameterized**].

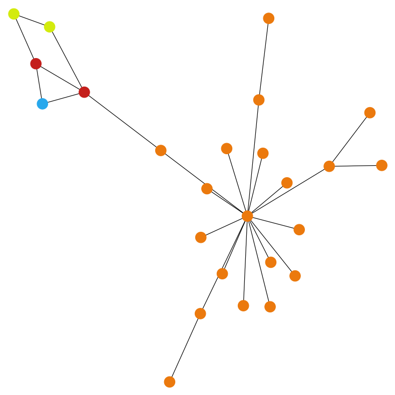
A.2 Data visualization


(a) BA-Shapes


(b) BA-Community


(c) Tree-Cycles


(d) Tree-Grid


(e) BA-2Motif

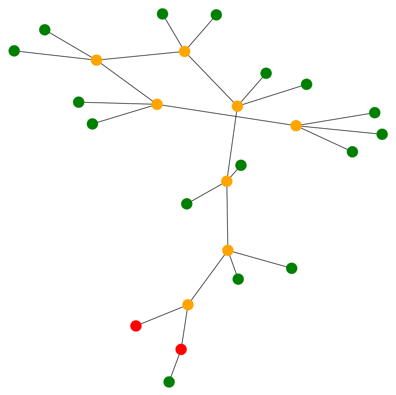

(f) MUTAG

Figure A.1: Visualization of all six datasets. For node datasets (a-d) the target prediction node where the computational graph is computed from is colored in light blue.

A.3 Replication Hyperparameter Searches

The following tables contain the grid search configurations for the explainer on each downstream model. The first row includes the configuration used in the original codebase [**luo2020parameterized**] and the second row contains the configurations used in [**holdijk2021re**]. For BA-Community (see Table A.3) both configurations are identical, and the second row is thus omitted. The last section of each table contains the set of the values that we tested for each parameter. The optimal settings for our explainer implementation are highlighted in each column. Note that we optimize Tree-Cycles (see Table A.4) and BA-2Motif (see Table A.6) towards a minimal metric score, as discussed in Section 5.1.2.

BA-Shapes									
a	K	b	E	η	S	α_e	α_s	τ_0	τ_T
N	1	0.0	10	0.003	-	1.0	0.05	1.0	0.05
N	1	0.0	10	0.003	-	1.0	0.05	5.0	2.0
5	**1**	0.0	10	0.0003	74	**0.1**	0.005	5.0	**1**
	5			**0.003**	75	0.5	**0.05**		2
	10			0.03	76	1.0	0.1		5

Table A.2: First two rows show hyperparameter settings used in [**luo2020parameterized**] and [**holdijk2021re**], respectively. Last section contains our grid search configuration. Bolded values indicate the best performance.

BA-Community									
a	K	b	E	η	S	α_e	α_s	τ_0	τ_T
N	1	0.5	20	0.003	-	1.0	0.05	1.0	1.0
64	1	**0.0**	20	**0.003**	74	**1.0**	0.05	1.0	1.0
	5	0.5		0.0003	75	0.1	**0.1**		**5.0**
	10				76				

Table A.3: First row shows hyperparameter settings used in [**luo2020parameterized**] and [**holdijk2021re**]. Last section contains our grid search configuration. Bolded values indicate the best performance.

Tree-Cycles									
a	K	b	E	η	S	α_e	α_s	τ_0	τ_T
N	1	0.0	20	0.003	-	0.01	0.0001	5.0	5.0
N	1	0.0	20	0.003	-	10.0	0.1	1.0	5.0
5	1	0.0	20	**0.0003**	74	0.01	**0.0001**	1.0	**1.0**
	5			0.003	75	**1.0**	0.05		5.0
	10				76	10.0	0.1		

Table A.4: First two rows show hyperparameter settings used in [**luo2020parameterized**] and [**holdijk2021re**], respectively. Last section contains our grid search configuration. Bolded values indicate the best performance.

Tree-Grid									
a	K	b	E	η	S	α_e	α_s	τ_0	τ_T
N	1	0.0	30	0.01	-	1.0	0.01	5.0	5.0
N	1	0.0	30	0.003	-	1.0	1.0	5.0	2.0
24	1	0.0	30	0.0003	74	0.1	0.01	5.0	**2.0**
	5		30	**0.003**	75	**1.0**	**0.5**		5.0
	10		30	0.01	76	10	1.0		
				0.05					

Table A.5: First two rows show hyperparameter settings used in [**luo2020parameterized**] and [**holdijk2021re**], respectively. Last section contains our grid search configuration. Bolded values indicate the best performance.

BA-2Motif									
a	K	b	E	η	S	α_e	α_s	τ_0	τ_T
N	1	0.0	10	0.003	-	0.0	0.00	1.0	0.0
N	1	0.0	20	0.005	-	0.01	0.03	5.0	1.0
30	1	0.0	10	0.0003	74	0.01	0.03	5.0	**1.0**
	5		**20**	0.003	75	**0.1**			5.0
	10			0.005	76				
				0.01					

Table A.6: First two rows show hyperparameter settings used in [**luo2020parameterized**] and [**holdijk2021re**], respectively. Last section contains our grid search configuration. Bolded values indicate the best performance.

MUTAG									
a	K	b	E	η	S	α_e	α_s	τ_0	τ_T
N	1	0.0	10	0.01	-	1.0	0.01	5.0	5.0
N	1	0.0	30	0.0003	-	1.0	0.005	5.0	5.0
30	1	0.0	10	0.0003	74	0.1	0.01	5.0	**1.0**
	5		**20**	0.003	75	**1.0**	**0.005**		5.0
	10		30	**0.01**	76				

Table A.7: First two rows show hyperparameter settings used in [**luo2020parameterized**] and [**holdijk2021re**], respectively. Last section contains our grid search configuration. Bolded values indicate the best performance.

A.4 Multiple Explanation Visualizations

The following figures contain 16 randomly sampled explanations for the best explainer model of every dataset. For graph tasks all nodes of the input graph are included, while for node tasks only the nodes of the local computation graph from the prediction motif node are shown.

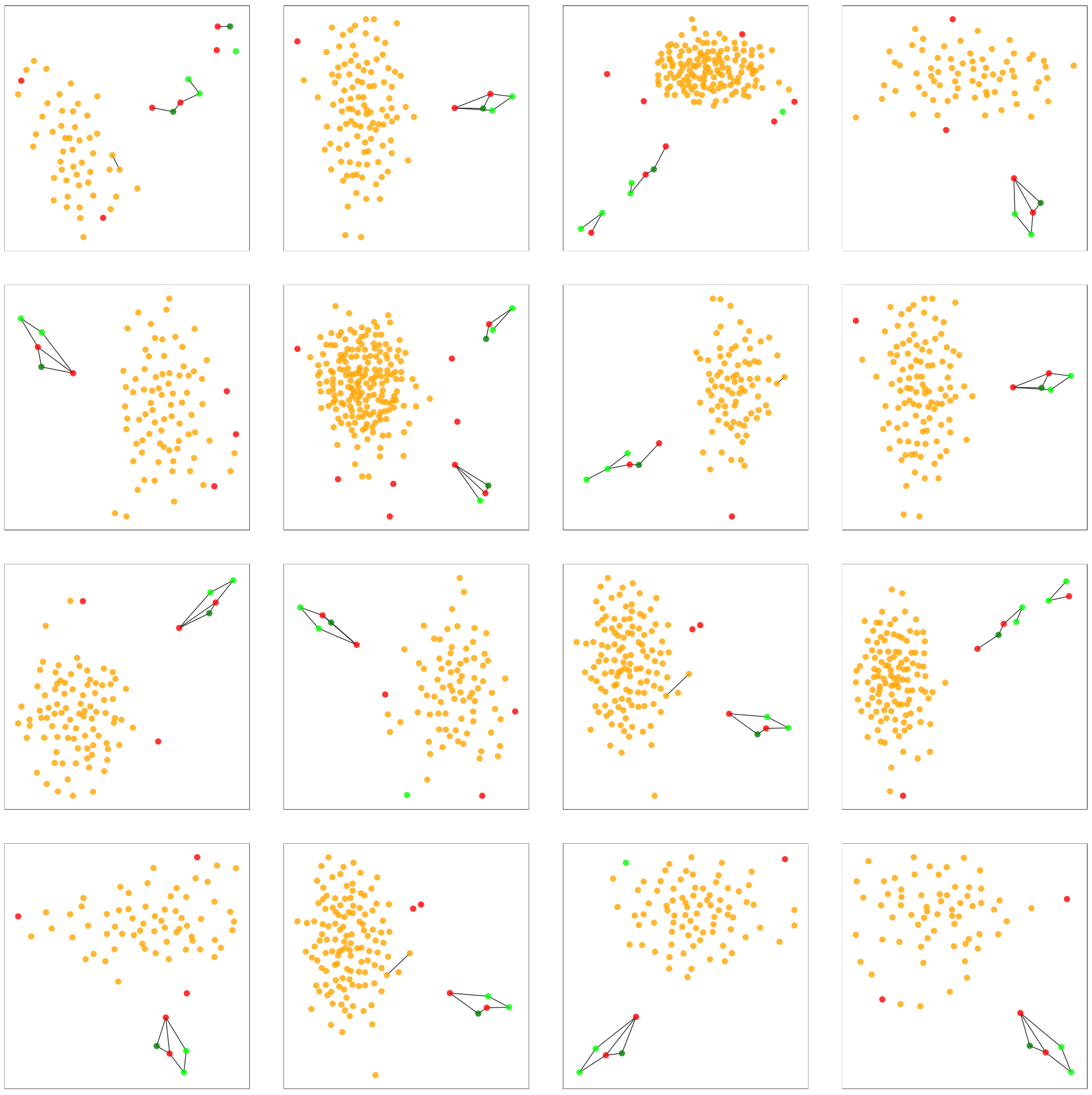


Figure A.2: Grid of BA-Shapes explanations (top-6 edges)

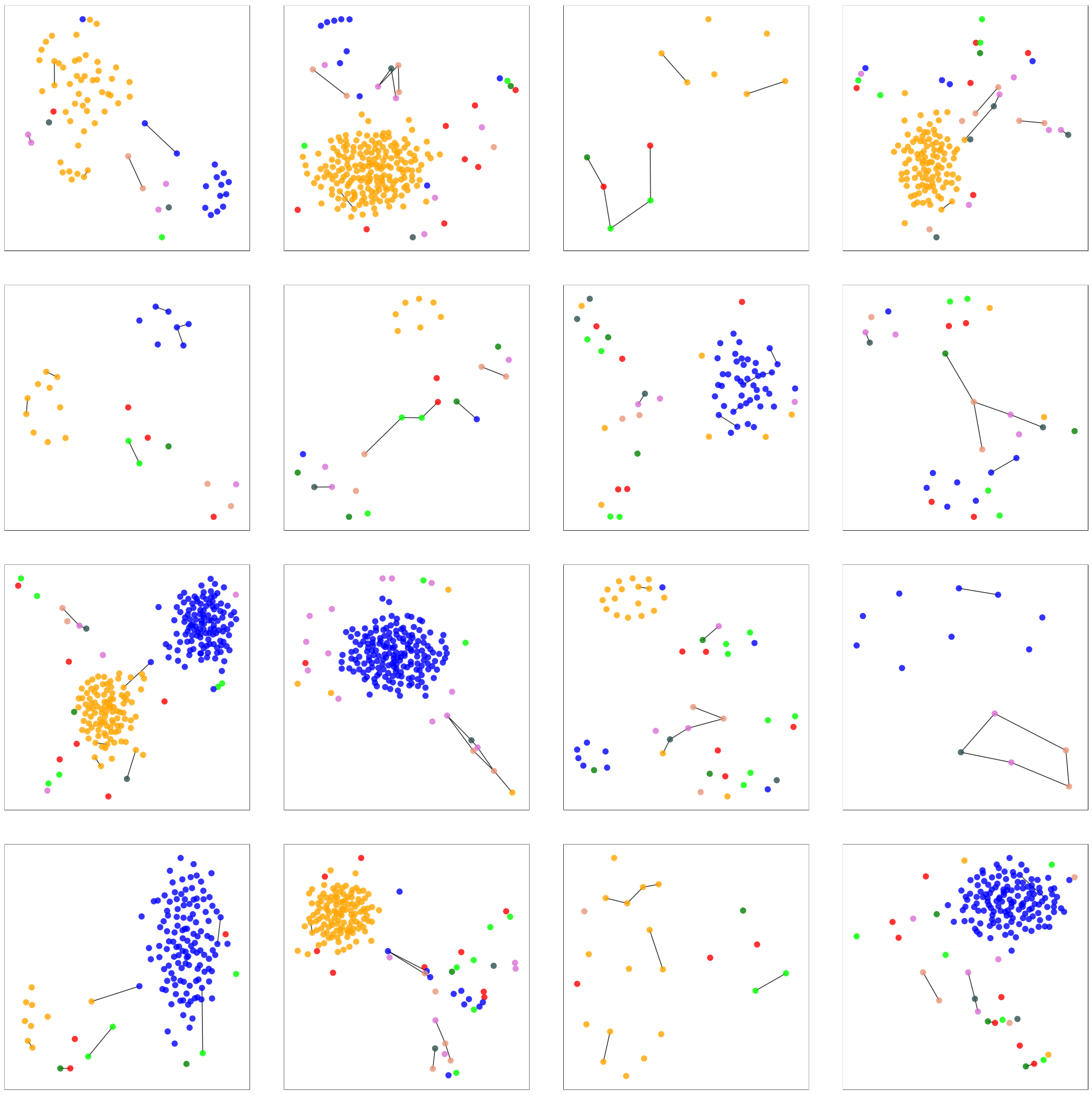


Figure A.3: Grid of BA-Community explanations (top-6 edges)

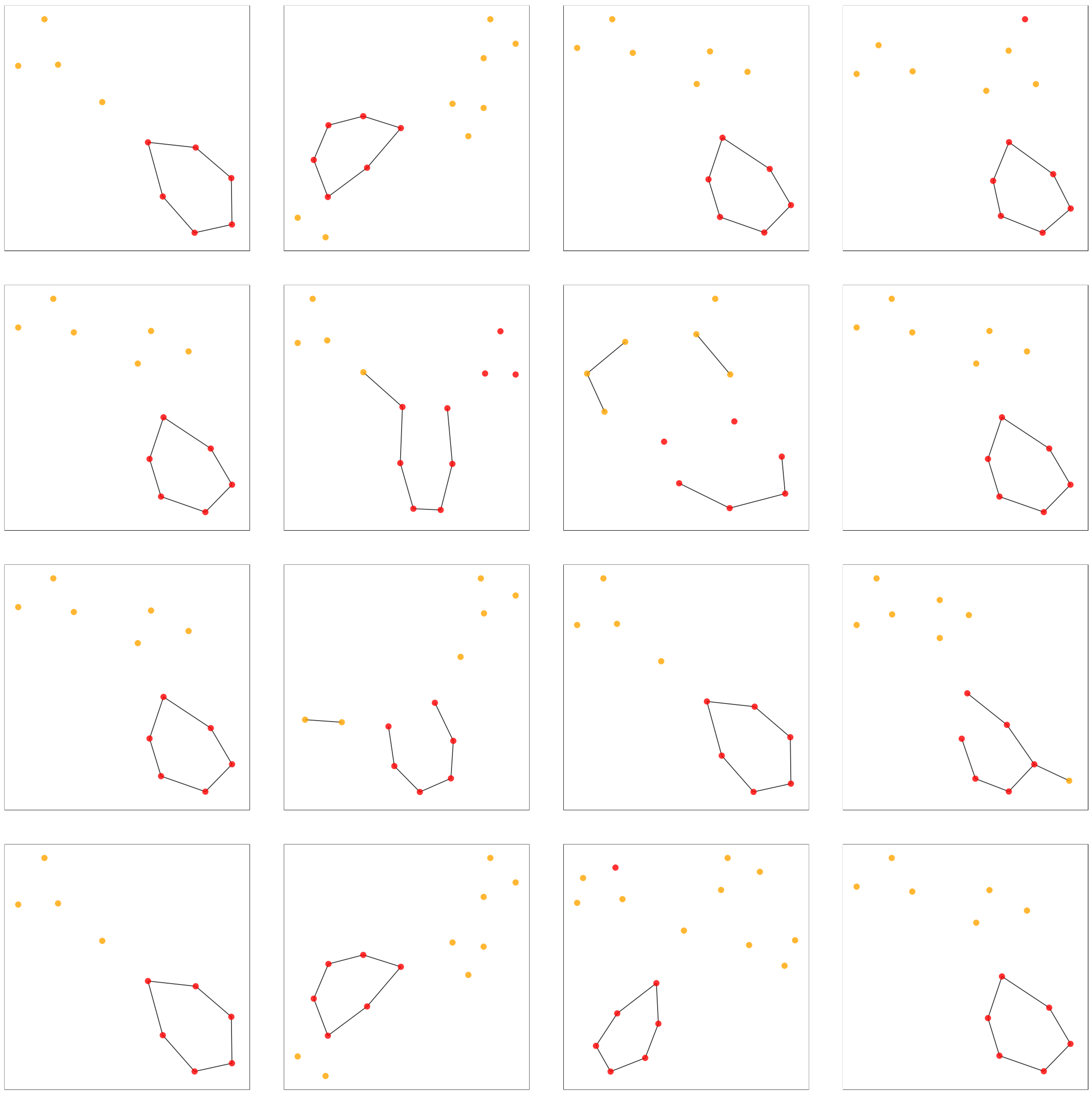


Figure A.4: Grid of Tree-Cycles explanations (top-6 edges)

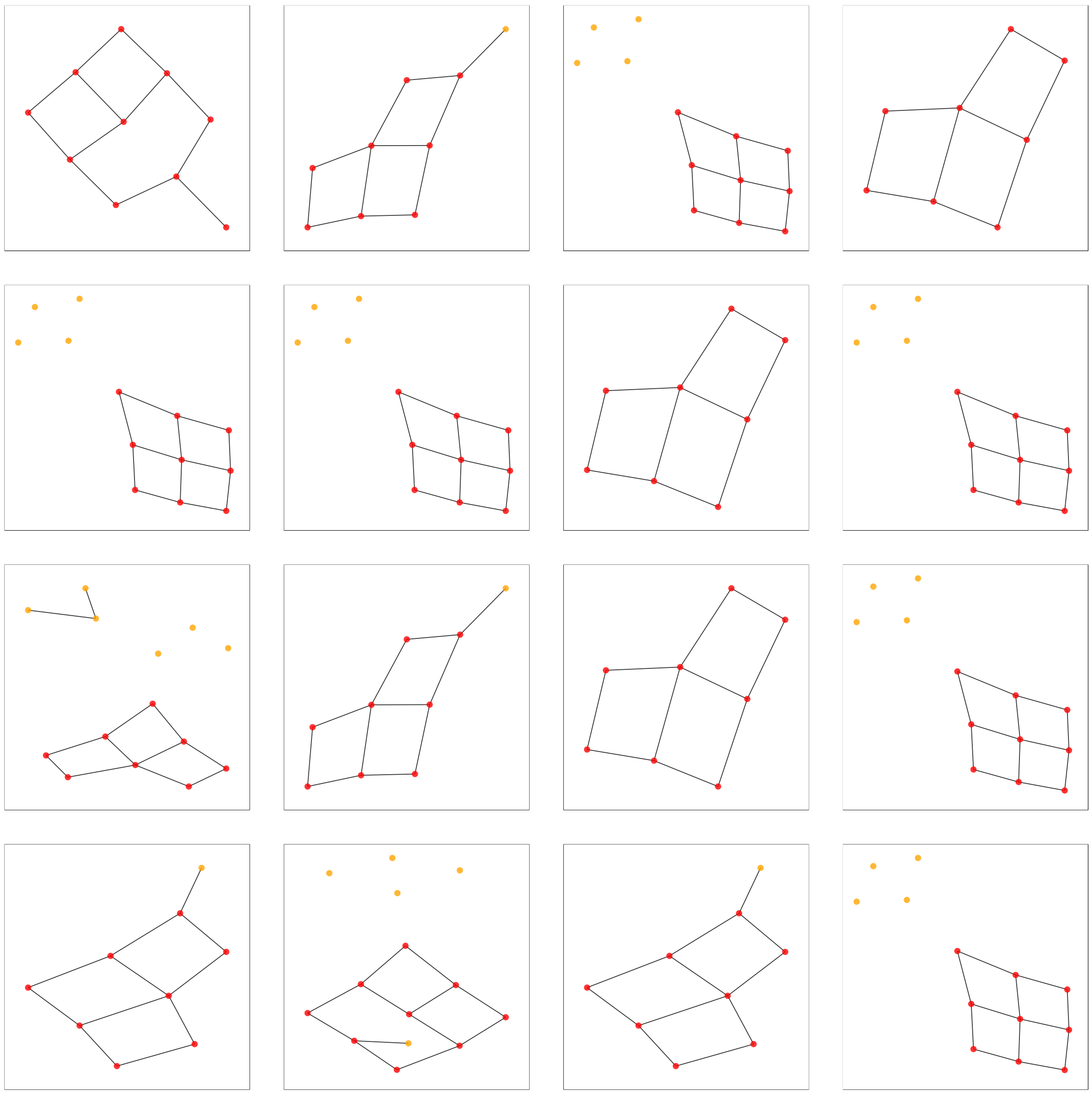


Figure A.5: Grid of Tree-Grid explanations (top-12 edges)

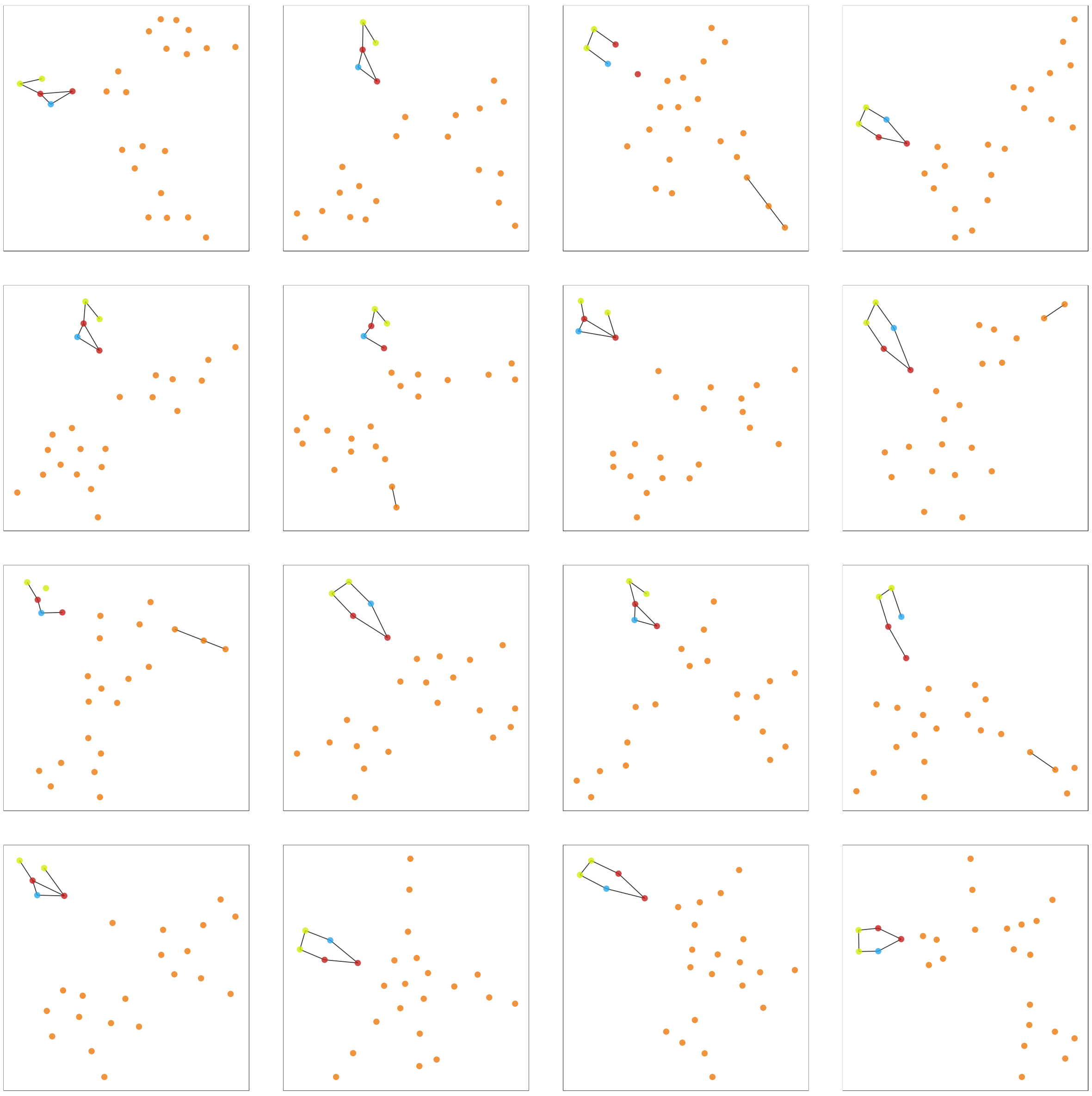


Figure A.6: Grid of BA-2Motif explanations (top-5 edges)

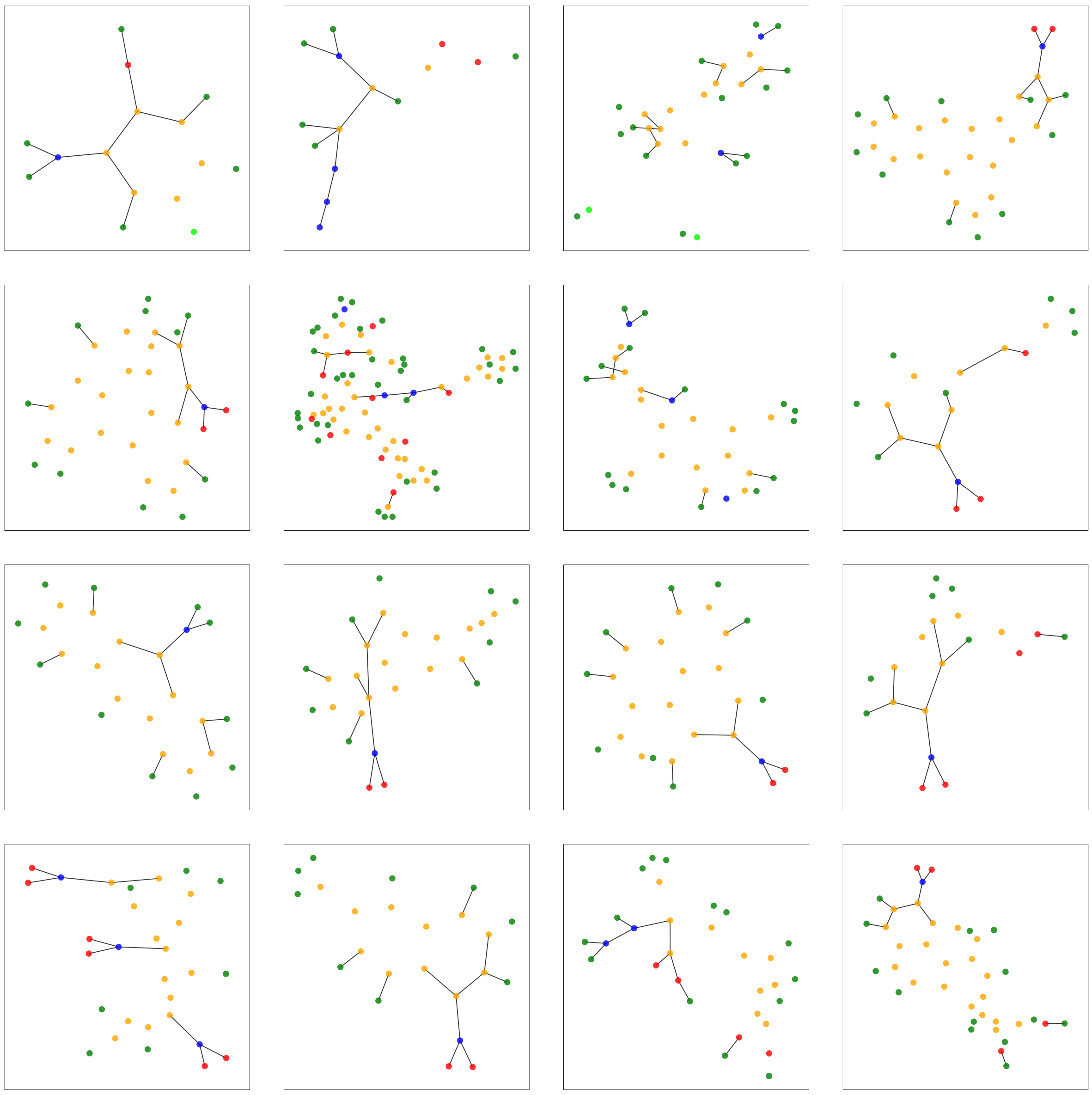


Figure A.7: Grid of MUTAG explanations (top-10 edges)

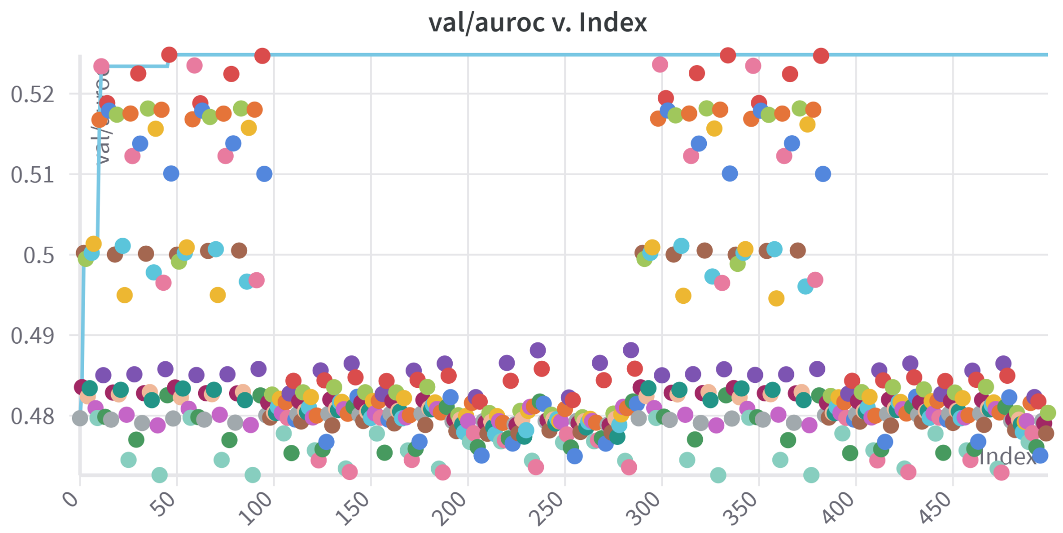


Figure A.8: Grid search results of all runs using WandB [**<empty citation>**].

A.5 NeuroSAT explainer Hyperparameter Searches

The following tables contain the grid search configurations for the NeuroSAT explainer models with either the soft constraint or the hard constraint applied. The columns contain the values tested for the corresponding hyperparameter. The optimal settings for each explainer implementation are highlighted.
For the hard constraint the mean local ROC-AUC was no higher than 0.502 and no lower than 0.44 over all runs. We select the hyperparameters that maximize the metric and minimize the validation loss while avoiding edge weights of 0. TODO:
We conclude that regardless of exact hyperparameters, the NeuroSAT model does not succinctly learn UNSAT cores.

Soft Constraint Explainer										
α_e	E	η	K	α_{concat}	S	α_s	τ_T	τ_0	α_C	α_{AdamW}
0.1	**20**	**0.0003**	1	False	75	**0.001**	**1.0**	5.0	0.1	False
1.0	30	0.003	**5**	**True**	76	0.01	5.0		1.0	**True**
	50	0.01				0.1			**10.0**	

Table A.8: Grid search results over hyperparameter space for the NeuroSAT explainer that uses a soft constraint. Bolded values indicate the best performance.

Hard Constraint Explainer									
α_e	E	η	K	α_{concat}	S	α_s	τ_T	τ_0	α_{arch}
0.1	20	**0.0003**	1	**False**	75	**0.01**	**1.0**	5.0	False
1.0	30	0.003	**5**	True	76	0.1	5.0		**True**
	50	0.01				1.0			

Table A.9: TODO: ALSO INCLUDE LOWER LR?!?! Highlighted values are the best performing ones, leading to a smaller mean validation AUROC and avoiding same weighted edges. Bolded values indicate the best performance.

A.6 REMOVE

	Explanation AUC			
Method	BA-Shapes	BA-Community	Tree-Cycles	Tree-Grid
Our work (inductive)	0.993±0.002	0.829±0.008	0.109±0.108	0.689±0.027
Inference Time (ms)	39.0±1.9	25.6±1.5	3.1±0.3	3.4±0.1

Table A.10: PGExplainer performance WITH $a = 0.08$! (4.8, 64, 4.8, 23) (USED IN SWEEP!)

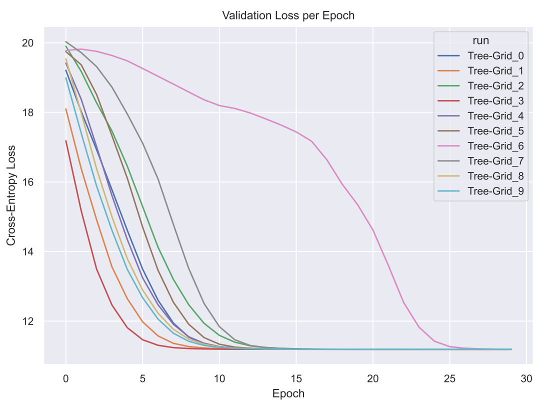


Figure A.9: Mean validation Loss per Epoch (Tree-Grid).

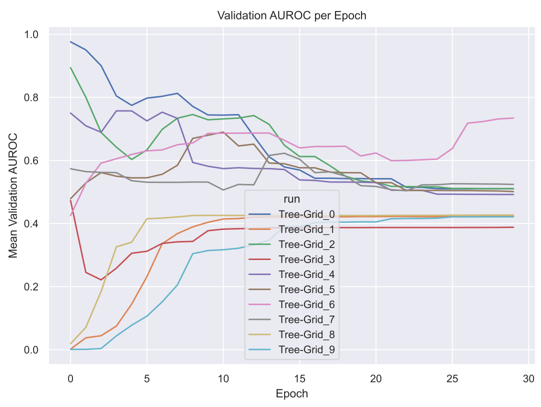


Figure A.10: Mean validation AUROC per Epoch (Tree-Grid).

	One-Motif-Node-Train			
#used motif instances N	60	60	160	32

Method	**Explanation AUROC**			
	BA-Shapes	BA-Community	Tree-Cycles	Tree-Grid
Experiment	0.85±0.005	0.731±0.016	0.224±0.160	0.65±0.026

Table A.11: TODO: THIS DOES NOT MAKE SENSE! TRAIN AND EVAL NODES SHALL BE FROM THE SAME DISTRIBUTION!!! Explanation AUROC for One-Motif-Node-Train.

Eidesstattliche Versicherung

(Affidavit)

Name, Vorname (surname, first name)	Matrikelnummer (student ID number)

☐ Bachelorarbeit
(Bachelor's thesis)

☐ Masterarbeit
(Master's thesis)

Titel
(Title)

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.	I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

Ort, Datum (place, date)	Unterschrift (signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

Ort, Datum (place, date)	Unterschrift (signature)

***Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.**