dortmund university

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

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

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 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. [40] in section 2.4, as well as two succeeding models. After introducing the most relevant concepts, in section 3.1 we briefly discuss the field of explainability in graph neural networks to motivate the selection of this approach for our work. Lastly, in section 2.5 we describe the boolean satisfiability problem that motivates this work.

2.1 Deep learning

TODO: Generally more examples, sources etc., to convey broad knowledge in the field
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. [15].
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?

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. [24] achieved record-breaking results by using a large, deep convolutional network for image classification.

TODO: LAYER AND PARAMETER NOT YET DEFINED HERE 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 [35]. 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 $f(x - \epsilon \operatorname{sign}(f'(x))) < f(x)$ 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[5]).

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 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 et al. [27]. 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

TODO: consisting of multiple layers of hidden units and nonlinear activation functions A classical DL model is the multilayer perceptron (MLP), first proposed by Rosenblatt [39], with the general goal of approximating a function f^*. In the case of classification we could define a function $y = f^*(x)$ that maps an input x to a label y. The MLP then defines the mapping $y = f(x; \boldsymbol{\theta})$ 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 [14]. These models are also referred to as feedforward neural networks, as they process information from 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 $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$ that consists of input layer $f^{(1)}$, hidden layer $f^{(2)}$ and output layer $f^{(3)}$. 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 $f^*(x)$ at different points in the training and labels $y \approx f^*(x)$. These training examples dictate the output layer to generate a value close to y for each input 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 θ 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. [28]). Examples are the Sigmoid function

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

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. 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

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 [18] 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 [21] 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. [15].

Let $\mathbf{B}$ be a minibatch of activations of the layer to normalize, in the form of a design matrix with rows of activations. To normalize $\mathbf{B}$, it is replaced by

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

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 $\mathbf{B}'$ as functionally equivalent to $\mathbf{B}$. During training,

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

and

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

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 minibatches.

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. [15][p.590]. 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 will loosely follow Liu et al. [28]. 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 [9][p.2] 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

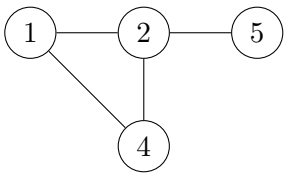
to v and denoted by $d(v) = |\{(u, v) \in E(G) \mid u \in V(G)\}|$. G can be described by an adjacency matrix $A \in \mathbb{R}^{n \times n}$, where

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

If G is an undirected Graph the adjacency matrix will be symmetrical, as seen in figure 2.1.
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 $H \subseteq G$. The number of nodes in a graph $|V|$ is its order and the number of edges $|E|$ is its size (see Diestel [9]). We additionally define bipartite graphs according to Asratian et al. [2]: 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 colour 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 coloured by at most two colours so that no two adjacent nodes share the same colour.



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

Figure 2.1: 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 A (right).

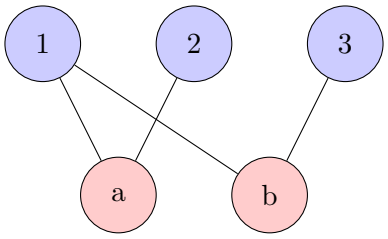


Figure 2.2: A bipartite graph with color classes $V_1 = \{1, 2, 3\}$ (blue) and $V_2 = \{a, b\}$ (red).

Random Graphs

Gilbert [13] 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.
A random graph is further described by Diestel [9][p.323] 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 et al. [7][p.13] if not stated otherwise.

2.3.1 Entropy

Entropy is used to describe the uncertainty of a random variable. It measures the amount of information required on average to describe a random variable. Let X be a discrete random variable with alphabet $\mathcal{X}$ and probability mass function $p(x) = Pr\{X = x\}$ for $x \in X$. TODO: EINHEITLICHE DEFINTION! 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 nats in our case. We will use the convention from Cover et al. [7] that $0 \log 0 = 0$, as terms of zero probability does 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 et al. [7] 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. [15][p.74]):

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

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 $\mathcal{X}$:

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

$$= -\sum_{x \in \mathcal{X}} p(x) \log p(x) + \sum_{x \in \mathcal{X}} p(x) \log p(x) - \sum_{x \in \mathcal{X}} p(x) \log q(x) \tag{2.22}$$

$$= -\sum_{x \in \mathcal{X}} p(x) \log q(x) \tag{2.23}$$

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 $p(x)p(y)$:

$$I(X; Y) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log \frac{p(x, y)}{p(x)p(y)} \tag{2.24}$$

$$= H(X) - H(X|Y) \tag{2.25}$$

2.4 Graph Neural Networks

TODO: DOES THIS CONVEY MESSAGE PASSING???
Graph Neural Networks(GNNs) [40] 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 [12], graph classification [45] and link prediction tasks [50]. 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 [25], such as local connection, shared weights, and multi-layer usage, with the concept of graph embeddings [4] to leverage the power of feature extraction and representation as low-dimensional vectors for graphs (see Liu et al. [28]).

TODO: CONCRETELY DEFINE NODE AND GRAPH TASKS! 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 [16] and refined in [40]. Thus, we describe the supervised GNN model by Scarselli et al. [40] 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 $\mathbf{h}_v \in \mathbb{R}^S$ 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. [40] 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

$$\mathbf{h}_v = f(\mathbf{x}_v, \mathbf{x}_{E(v)}, \mathbf{h}_{\mathcal{N}(v)}, \mathbf{x}_{\mathcal{N}(v)}), \tag{2.26}$$

$$\mathbf{o}_v = g(\mathbf{h}_v, \mathbf{x}_v), \tag{2.27}$$

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:

$$\mathbf{H} = F(\mathbf{H}, \mathbf{X}), \tag{2.28}$$

$$\mathbf{O} = G(\mathbf{H}, \mathbf{X}_N). \tag{2.29}$$

Note that F is assumed to be a contraction map and the value of $\mathbf{H}$ is the fixed point of equation (2.28). To compute the state the iterative scheme

$$\mathbf{H}^{t+1} = F(\mathbf{H}^t, \mathbf{X}) \tag{2.30}$$

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

$$loss = \sum_{i=1}^{p} (t_i - \mathbf{o},) \tag{2.31}$$

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 h_v^t are updated iteratively using equation (2.26) until time step T. We then obtain an approximate fixed point solution of equation (2.28): $\mathbf{H}(T) \approx \mathbf{H}$. 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 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. [40] 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 2.3, 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 [28].

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 [26]. 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.

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 et al. [23] that introduces a simple, layer-wise propagation rule for multi-layer GCNs as

$$H^{l+1} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}), \tag{2.32}$$

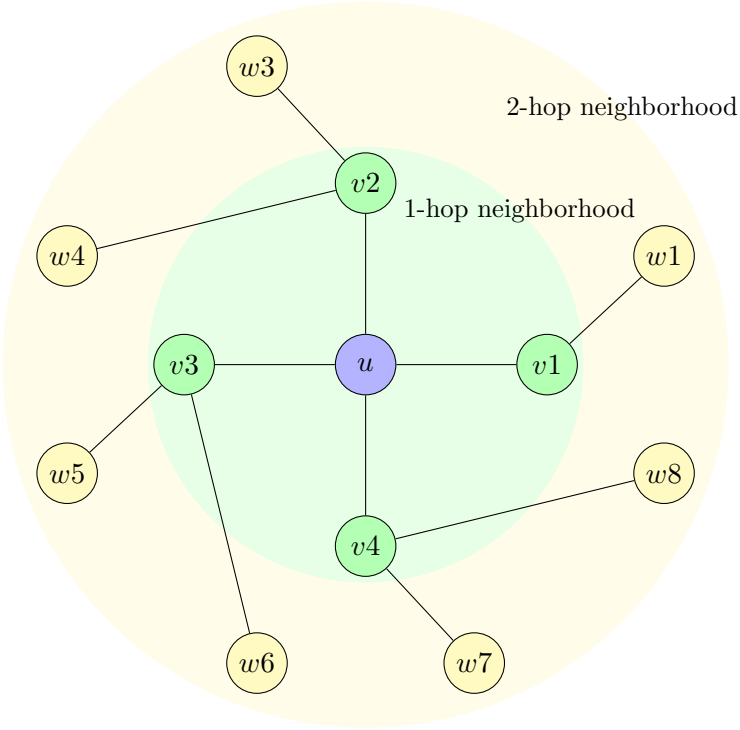


Figure 2.3: Visualization of the k-hop neighborhood of u.

where $\tilde{A} = A + I_N$ is the adjacency matrix of the undirected input graph G with added self-connections. I_N is the identity matrix, $\tilde{D}$ is the degree matrix and $W^{(l)}$ is a layer-specific trainable weight matrix. $\sigma(\cdot)$ denotes an activation function, such as ReLU. $H^{(l)} \in \mathbb{R}^{n \times d}$ denotes the matrix of node activations in the l-th layer, where $H^{(0)} = X$. Note that this definition differs slightly from the iterative formulation in equation 2.30, in the sense that each layer applies a different function F_l, rather than a shared function F. $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ 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. [34] 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:

$$x_i^{(l)} = \sigma(W_1^{(l)} x_i^{(l-1)} + W_2^{(l)} \cdot \sum_{j \in \mathcal{N}(i)} e_{j,i} \cdot x_j^{(l-1)}), \tag{2.33}$$

where $W_1^{(l)}$ and $W_2^{(l)}$ are two layer-specific trainable weight matrices.

TODO: MORE MOTIVATION FOR THIS!

2.5 Boolean Satisfiability Problem

We define the Boolean Satisfiability Problem (SAT) according to Guo et al. [17][p.641]: 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 formulas in CNF, as NeuroSAT [42] 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

$$\psi(V) = (x_1) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_2)$$

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 [6]). Current state-of-the-art SAT solvers apply searching based methods such as Conflict Driven Clause Learning [32] or Stochastic Local Seach [41] 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. [17] 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 2.4). The resulting graph G_b can formally be described by a biadjacency matrix B of shape $L \times C$, with (L, C) being a bipartition of the G_b into literals and clauses.
Following Sun et al. [43], let $A \in \mathbb{R}^{(L+C)\times(L+C)}$ 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

$$A(i,j) = \begin{bmatrix} 0_{L\times L} & B \\ B^T & 0_{C\times C} \end{bmatrix}, \tag{2.34}$$

where 0 denotes a zero matrix in the shape of their subscript.

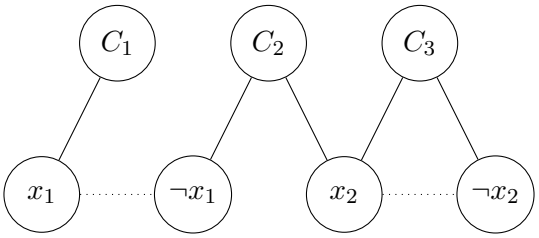


Figure 2.4: LCG representation of $\psi(V)$ with dashed lines representing the connection between complementary literals relevant for the message passing in GNNs.

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 minisat [10] 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. [44]).

Chapter 3

Related Work

Yuan et al. [48] 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 in 3.1. The authors note that the PGExplainer "is not performing as promising as its original reported results" [48]. Nevertheless, we evaluate the explainer model with regard to its applicability in the inductive setting. In 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 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 [46], [30]. However, the interpretability of these models is often limited due to their black-box design [35]. 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 [37].

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. [48]). 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 [3] 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 [47] and PGExplainer [29] 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 [49], 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 [20] 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 [3], 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. [37]). 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.

TODO: Claims to satisfy? Since the perturbation-based PGExplainer [29] claims to satisfy all the criteria, while also maintaining reasonable computational cost, we select this model

for the course of this study. It is furthermore able to generate explanations in an inductive setting, which we aim to do in **??**.
The general pipeline for different perturbation based approaches 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 and update the mask generation algorithm according to the similarity of the predictions.
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 et al. [8]). 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, 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. [38]). 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. 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, results of taxonomy propose only using PGExplainer for Node Classification as it achieves low fidelity on Graph tasks
TODO: figure of perturbation pipeline?

3.2 TODO: SECTION NAME Explainer Models

GNNExplainer

Ying et al. proposed the GNNExplainer [47]: 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 Networks

The Parameterized Explainer for Graph Neural Networks (PGExplainer) by Lou et al. [29] is the main subject of our work. The method adopts a deep neural network to parameterize the generation process of explanations, thus allowing multiple instances to be explained collectively. Furthermore, it has better generalization ability and can explicitly be utilized in an inductive setting.

We reimplement the original work using PyTorch [36] and PyTorch Geometric [11] instead of TensorFlow [33], while also emphasizing its application in an 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. This extension is motivated by the need for explainability methods that generalize across architectures (TODO: SOURCE FOR THIS). 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. [19] 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 Downstream Model

NeuroSAT

NeuroSAT by Selsam et al. [42] 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 unsat cores, revealing that NeuroUNSAT is able to detect UNSAT cores if the UNSAT problems contain a specific UNSAT core. However, this is expected to be due to the model memorizing the unsat cores, rather than generalizing to any unsat core. We want to test this by applying the PGExplainer in an inductive setting to the NeuroSAT model, and evaluate whether the generated explanations do align with unsat cores.

Chapter 4

PGExplainer - Main part - Methodology

TODO: This can be understood as an extension of the theoretical background? Since the paper differs from the codebase and is imprecise about certain descriptions (see Holdijk), we aim to give a thorough introduction that regards everything.

In the following chapter, we introduce the PGExplainer by Luo et al. [29] 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" [29]. 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. [47].

The focus in this approach lies in explaining the graph structure, rather than the graph features, as feature explanations are already common in non-graph neural networks.

In Section 4.2 we present the idea of applying PGExplainer on the NeuroSAT framework to generate explanations for the machine learning SAT-solving approach and comparing these to "human-understandable" concepts like UNSAT cores and backbone variables.

We then describe our reimplementation in detail (Section 4.3), including the changes made and difficulties during the process.

In section 4.4 we describe the experimental setup used for both the replication of the PGExplainer in the inductive setting, as well the application on NeuroSAT.

4.1 Theoretical Foundations of PGExplainer

We follow the structure of the original paper [29] and start by describing the learning objective in 4.1.1, the utilized reparameterization trick in 4.1.2, the idea of global explanations in 4.1.3 and finally the applied regularization terms in 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 [47], 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 whitebox explanations (see Dabkowski et al. [8]). 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 2.2, as follows: Let $e_{ij} \in V \times V$ be the binary variable indicating whether the edge is selected, with $e_{ij} = 1$ 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

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

TODO: Inhomogeneous Erdos Renyi model? Mention that this is a generative model? (A Gilbert random graph is an example of a generative probabilistic model on graph data?) $P(e_{ij})$ is instantiated with the Bernoulli distribution $e_{ij} \sim Bern(\theta_{ij})$, where $P(e_{ij} = 1) = \theta_{ij}$ 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 [22] 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. 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 [31], also referred to as Gumbel-Softmax distribution, is utilized as an instantiation for the sampling, yielding the weight $\hat{e}_{ij} \in (0, 1)$ for edge (i, j) in $\hat{G}_s$, computed by:

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

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 $\hat{e}_{ij}$ is binarized with the sigmoid function $\lim_{\tau\to 0} P(\hat{e}_{ij} = 1) = \frac{\exp(\omega ij)}{1+\exp(\omega ij)}$. Since $P(e_{ij} = 1) = \theta_{ij}$, 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 $\frac{\partial \hat{e}_{ij}}{\partial \omega_{ij}}$ 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 [47] 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. PGExplainer hence explains predictions of the target model over multiple instances collectively (TODO: Cut this sentence?).
Since GNNs apply two functions F and G to calculate the global state embeddings and downstream task outputs respectively, we denote these two functions as $\text{GNNE}_{\Phi_0}(\cdot)$ and $\text{GNNC}_{\Phi_1}(\cdot)$ for any GNN in the context of PGExplainer. For models without explicit classification layers the last layer is used to compute the output 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 the matrix of final node representations z, referred to as node embeddings, and the initial state is G_o. TODO: (Because of focus on graph structure rather than features?) For generalizability across different GNN layers the output is only dependent on the node representation, that encapsulates both features and structure of the input graph. This representation also serves as the input for the explainer network g, defined as: TODO: Rename g?

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

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, PGExplainer 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, $G^{(i)}$ and $G_s^{(i,k)}$ denote the input graph and the k-th graph sampled with equation 4.4 in 4.8 respectively for instance i. The authors propose two slightly different instantiations for node classification and graph classification tasks.

Explanation network for node classification

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 we specify the network in 4.8 as:

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

MLP_Ψ is an MLP (TODO see **??** for implementation details) parameterized with Ψ and $\oplus$ denotes the concatenation operation. Thus, a concatenation of the node embeddings of nodes i, j and v respectively is fed through the network. The output ω_{ij} is therefore an edge logit, which serves as a parameter in the sampling process.

TODO: "In GNNs with message passing mechanisms, the prediction at a node v is fully determined by its local computation graph, which is defined by its L-hop neighborhoods, with L being the number of GNN layers." (Source GNNExplainer)

Note that in their codebase the authors use a concatenation of all hidden representations instead of final node embeddings for node level tasks. For a target GNN consisting of multiple graph layers with

$$\begin{aligned}
\mathbf{H}_1 &= F_1(G_o, \mathbf{X}),\\
\mathbf{H}_2 &= F_2(\mathbf{H}_1, \mathbf{X}),\\
&\vdots\\
\mathbf{H}_L &= F_L(\mathbf{H}_{L-1}, \mathbf{X}),
\end{aligned}$$

this leads to $Z \in \mathbb{R}^{V(G_o)\times Ld}$ being the matrix of node embeddings z that are computed as:

$$z_i = h_{1,i} \oplus h_{2,i} \oplus ... \oplus h_{L,i} \tag{4.11}$$

Explanation network for graph classification

For graph level tasks the authors consider each graph to be an instance, regardless of specific nodes. The specification for the network thus becomes:

$$\omega_{ij} = \text{MLP}_\Psi([\mathbf{z}_i \oplus \mathbf{z}_j]), \tag{4.12}$$

where for each edge in G_o a concatenation of both its nodes is fed through the MLP.

TODO: ALGORITHMS IN APPENDIX

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.
We discuss the results of the inductive performance study by the authors in **??**.
"How to explain predictions of GNNs on a set of instances collectively and easily generalize the learned explainer model to other instances in the inductive setting remains largely unexplored in the literature." (This is what PGE fills?!)
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 [47], 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:

$$H_{\hat{G}_s} = -\frac{1}{|\varepsilon|} \sum_{(i,j)\in\varepsilon} (\hat{e}_{ij} \log \hat{e}_{ij} + (1 - \hat{e}_{ij}) \log(1 - \hat{e}_{ij})), \tag{4.13}$$

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**??** and are therefore included.

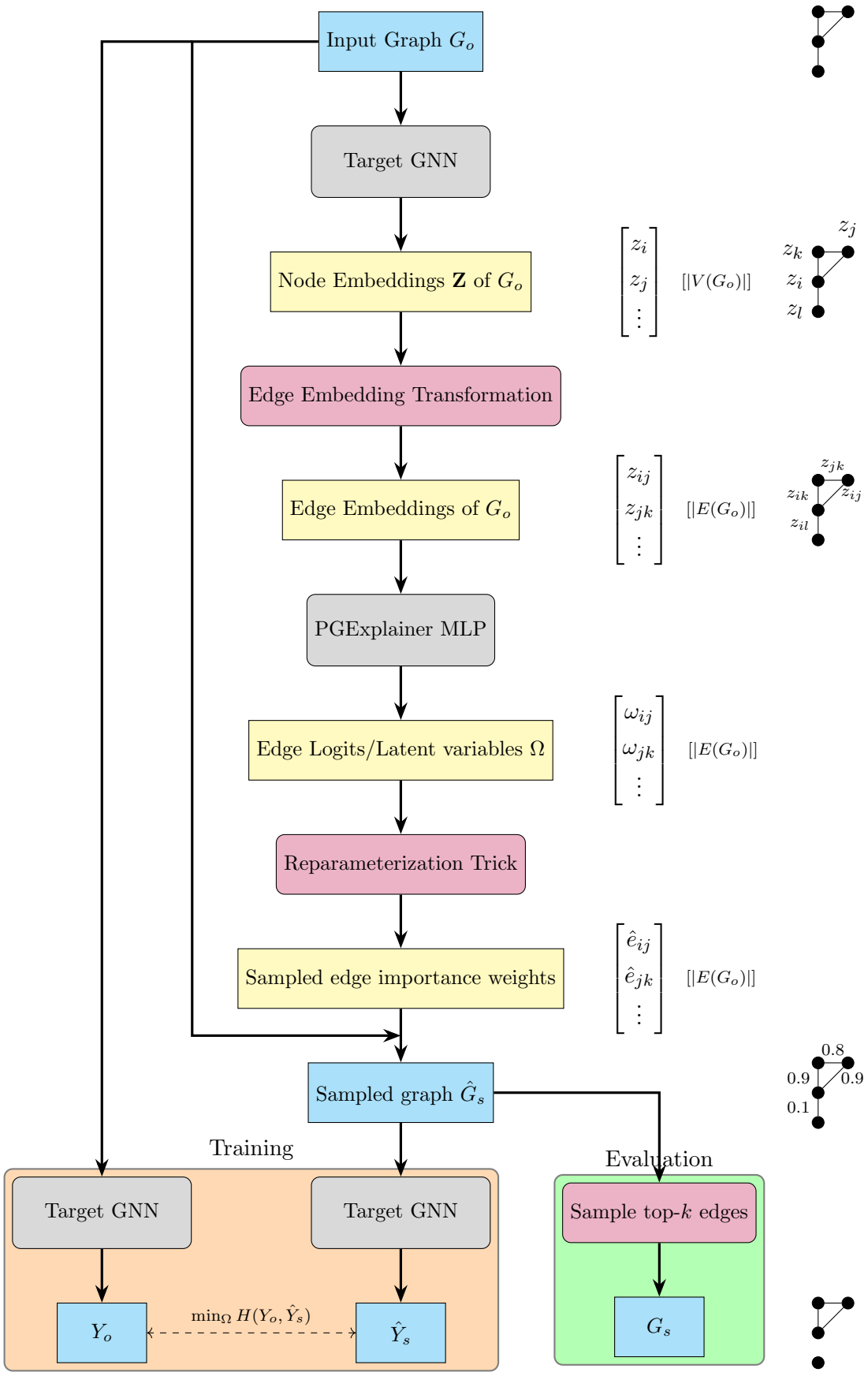


Figure 4.1: The complete pipeline of PGExplainer.

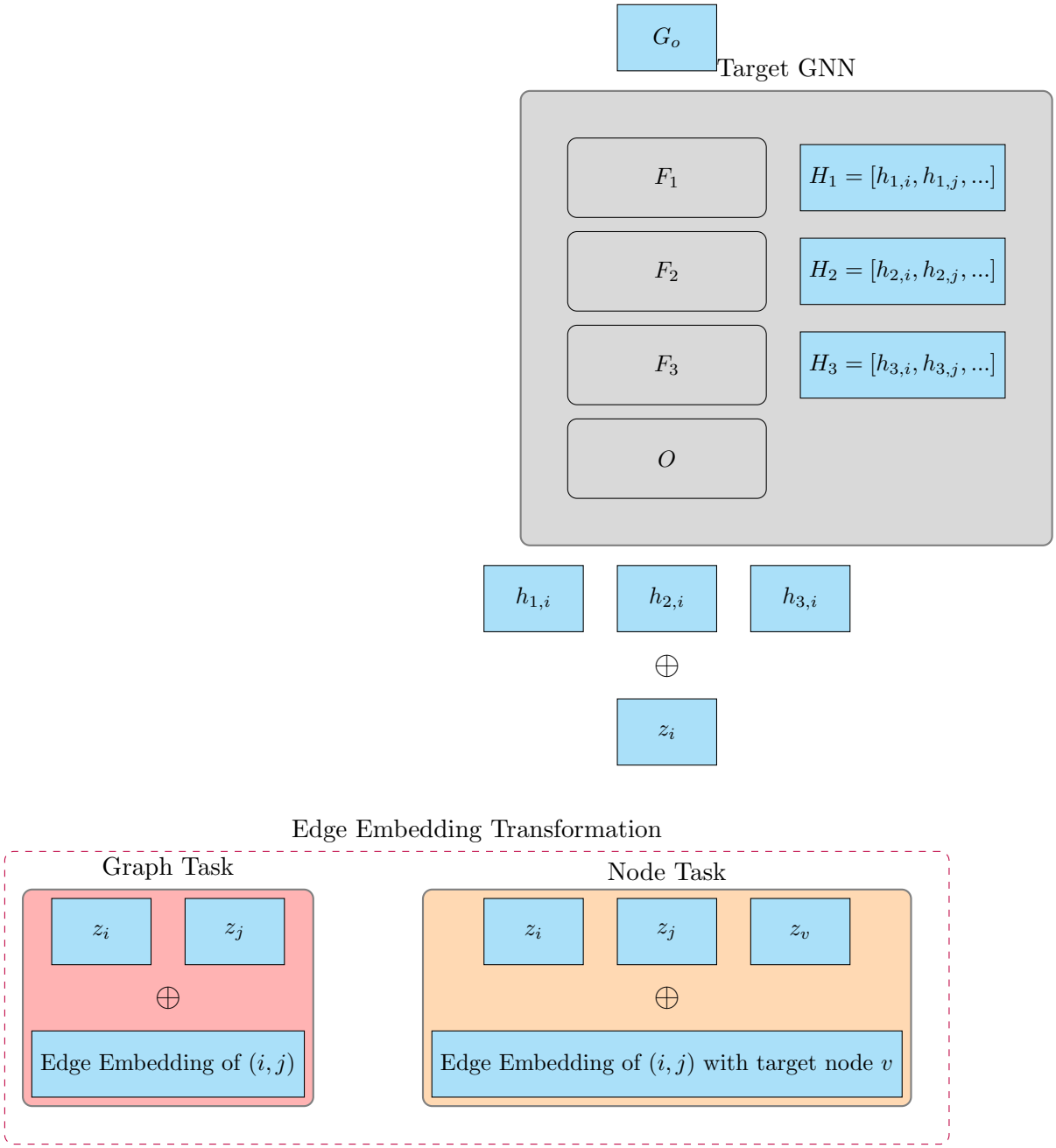


Figure 4.2: TODO: Visualization of the edge embedding transformation used to create inputs for the explainer network. Depending on the downstream task used in the target model the created edge embedding differs slightly.

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:

$$R_b = \text{ReLU}(\sum_{(i,j)\in\varepsilon} \hat{e}_{ij} - B). \tag{4.14}$$

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 subgrap, 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:

$$H(\hat{e}_{ij}, \hat{e}_{ik}) = -[1 - \hat{e}_{ij} \log(1 - \hat{e}_{ik}) + \hat{e}_{ij} \log \hat{e}_{ik}]. \tag{4.15}$$

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 TODO: Extension to application on NeuroSAT

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

TODO: BASIC INTRODUCTION/SUMMARY OF NEUROSAT, WHAT DOES IT DO.

NeuroSAT: Messages are passed between clauses and literals, as well as literals and their complement. 1. Clause receives from neighboring literals 2. Literals receive from clauses and complement.

(Define flip function that swaps literal row with row of its negation; relevant for NeuroSAT)

TODO: What changes in the context of NeuroSAT? How did we adapt PGExplainer?

Since PGExplainer generates edge wise explanations, and we want to evaluate the SAT problem evaluations with UNSAT cores as ground truth, we need to adapt the framework to account for the definition of UNSAT cores. Since a core is a subset of clauses, predicting singular edges 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 belong to the same node c, approximating a complete clause.

Changes for explainer: Edge embeddings calced by DS and passed to explainer. Calculated edge embeddings by concatenating node embeddings for connected nodes, similar to original. Embeddings fed into MLP, weights sampled with reparam. trick to get edge "probabilities", passed as "unbatched" sampled graph into NeuroSAT predictor. Visualization of SAT problems with edge weights, and gts.

Soft modified connectivity constraint

To account for the definition of UNSAT cores, that consist of sub-clauses of the original combination of clauses in the problem, we add a constraint that reinforces the prediction of complete clauses. Therefore, if the explainer assigns a high score to an edge (i, j), all edges (j, k) that also connect to the clause node j 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:

$$R_c = \sum_{c \in C} \mathrm{Var}(E_c) = \sum_{c \in C} \frac{1}{|E(c)|} \sum_{e \in E(c)} (e - \bar{E_c})^2$$

where $E_c = \{\hat{e}_{i,j} \mid j = c\}$ is the set of edge weights of edges that connect to clause c and $\bar{E_c}$ denotes the mean of E_c.

Hard constraint

Define PGE formulas over grouped clause edges rather than all edges.

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 exact same weight during training.

$$\epsilon_c \sim \mathrm{Uniform}(0, 1), \qquad \hat{e}_{l,c} = \sigma((\log \epsilon_c - \log(1 - \epsilon_c) + \omega_{ij}/\tau), \tag{4.16}$$

Additionally, we restrain the edge logits $\omega_{i,j}$ calculated by the MLP to be identical for all edges that connect to the same clause. Let $\Omega_c = \{\hat{\omega}_{i,c} \mid (i, c) \in E\}$ denote the set of logits corresponding to the edges connected to node c. We update these logits with:

$$\mu_c = \frac{1}{|\Omega_c|} \sum_{\hat{\omega}_{i,c} \in \Omega_c} \hat{\omega}_{i,c} \tag{4.17}$$

We calculate the mean logit μ_c for all edges that connect to node c with

$$\mu_c = \frac{1}{|E_c|} \sum_{(i,j) \in E_c} \omega_{i,j}. \tag{4.18}$$

The update rule is then defined as

$$\omega'_{i,j} = \mu_c, \qquad \forall (i, j) \in E_c \tag{4.19}$$

4.3 Implementation details

In the following, we provide the implementation details needed reproduce our results. This includes the general replication of the PGExplainer in section 4.3.1 and the specifics for the adaptation on NeuroSAT in section 4.3.2.

4.3.1 Replication of PGExplainer

General description: What framework (e.g., PyTorch, TensorFlow, scikit-learn) and tools you used.
Architecture specifics: Model structures, key layers, differences from the original paper.
Hyperparameters (default ones): E.g., activation functions, optimizer choice, learning rate (but not tuned values yet).
Extension description: What exactly you changed/extended compared to the original work, and how you did it technically.
Preprocessing: If you had to massage or transform input data before feeding it to the model.
Reproducibility: Seed setting, tricks to make the results reproducible.
Challenges: Brief notes if you faced and solved technical issues during reimplementation (e.g., missing details in the original paper).
TODO: WE USE PGEXPLAINER IN INDUCTIVE SETTING, OPPOSED TO COLLECTIVE SETTING USED IN ORIGINAL/GNNExplainer. INDUCTIVE SETTING EXPERIMENTS ALSO PERFORMED IN PGEXPLAINER!
Notes on Original code of target model: Not clearly specified in paper, but according to code: - No specific args set for layers etc., except hyperparameters for different datasets (see sweeps) - Therefore, assuming default settings for layers -¿ Graph conv layer uses ReLu activation (done), bias initialized with zeroes (done), no dropout(partially done), and embedding normalization! (TODO), default order 'AW'(see 4.17), weight init 'glorot' (Done with Xavier) - WE USED PyG GraphConv since it allows for bipartite+edge weights; but compare to layer close to original!!

Concrete graph layer used in original code is very similar to the semi-supervised layer**??**:

$$Z = \hat{A}XW \tag{4.20}$$

Note that a pytorch implementation of the semi-supervised layer is used in the replication paper:

$$X' = \hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}X\Theta \tag{4.21}$$

We use a pytoch implementation of the Weisfeiler and Leman Go Neural layer:

$$x_i' = W_1 x_i + W_2 \sum_{j \in \mathcal{N}(i)} e_{j,i} \cdot x_j \tag{4.22}$$

	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 4.1: Dataset statistics for Node and Graph Classification tasks.

TODO: Maybe separate the theory of PGE from our own work more strictly? E.g. 3. PGE theory, 4. PGE reimplementation and NeuroSAT application?
Implementation details:

- Started by reimplementing the downstream tasks used in og paper for node and graph class.
- Node datasets taken from original and transformed to a "pyg format", to keep original structure and ground truths
- Graph: BA-2Motif from pyg library with self generated gt, MUTAG dataset taken from pyg and added gt-labels that were added in original dataset
- Created pytorch/pytorch geometric implementation of 3-layer graph conv networks
- architecture as described in paper: ReLu activation, Pooling for graph net
- Xavier initialization for all layers
- Same hyperparameters?(Adam, $1 * 10^-3$ lr, 1000 epochs)/experimental setup?
- ADDED Dropout(0.1) to improve performance/overfitting on node tasks
- Fully connected layer: torch geometric GraphConv to allow passing of edge weights(+ bipartite)!?
- Original references sageConv in paper, pyG impl. does not allow edge weights, verify!
- GATConv for Graph attention(referenced by original)
- Alternatives: GCNConv(edge weights, no bipartite graphs)
- 80/10/10 split
- =¿ Similar accuracies achieved

Explainer: This is irrelevant for paper, description of why reparameterization trick necessary - First approach tried passing a masked edge index to downstream task with edge weights ¿ 0.5 =¿ Unable to learn with hard cut-off (bad for gradients). (Same with TopK probably?!) - Pass calculated edge weights to downstream task for learning. If no edge weights are passed(downstream task), all edge weights are initialized with one to represent all edges beeing relevant

4.3.2 Application to NeuroSAT

and adapt the NeuroSAT model to allow passing edge weights into the adjacency matrix.
PGExplainer adapted for SAT data, NeuroSAT embeddings and evaluation.
Reimplementation of NeuroSAT provided by Rodhi. As the code used for NeuroSAT can also be found in our Repository, we stress that only the changes described in the following chapter are part of our work.

Only change in NeuroSAT: pass edge weights into adjacency matrix. Calculates ....

4.4 Experimental setup

In this section we describe the setup for the experiments that we perform on PGExplainer. We start with the utilization of PGExplainer in the inductive setting, similar to the study performed by the authors. Additionally, we include experiments on the performance in the collective setting for the sake of completeness. Lastly, we present our approach for generating bipartite explanations for NeuroSAT [**<empty citation>**] predictions of SAT problems.
Datasets used: Name, origin, train/test/validation split (or cross-validation if you use it).
Training procedures: How long you trained, on which hardware (especially if relevant), early stopping criteria, etc.
Evaluation metrics: E.g., accuracy, F1-score, BLEU, MSE, depending on your task.
Hyperparameter tuning: If you did hyperparameter search, describe how (grid search, random search, Bayesian optimization?).
Baselines: Which models/versions you are comparing against (e.g., original model, your extension, ablation studies).
Experimental protocol: How many runs you did to account for randomness (e.g., averaged over 5 seeds).
Visualization tools: If you generated plots/graphs during experiments (like loss curves), you can briefly mention this.

4.4.1 PGExplainer in the inductive setting

Datasets For our reimplementation 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 GNNExpplainer. 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. 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 [1].
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 (TODO: see xy). 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 (TODO: connected how). 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, 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. 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 and 1 implying a house and circle, respectively (TODO: CHECK). The real-world dataset MUTAG contains $4,337$ molecule graphs that are assigned to one of 2 classes, depending on the molecules mutagenic effect **??**. Following **??**, 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. Since there are no explicit motifs for the non-mutagenic graphs, these grapgs are not considered in PGExplainer.
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 mutagenic graphs where either NH_2 or NO_2 are present are selected for the MUTAG experiment. In the node classification experiments the node sets 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 is also left unexplained by the authors.

Hyperparameter search

As found by Holdijk et al.**??** the PGExplainer is very sensitive to hyperparameter setting on each dataset. Therefore, we conduct hyperparameter searches for each of the datasets to obtain best performing explainers. We follow Liashchynskyi et al. [27] to perform grid searches over the parameter space that we define as an extended combination of the setting used in the original**??**, as well as the configs provided in Replication study by Holdijk et al.**??**. More details can be found in Appendix**??**.

Parameter	**Values**
epochs	20
paper_loss	True, False
tT	1.0, 5.0
size_reg	0.03
entropy_reg	0.01
lr_mlp	0.003, 0.0003
sampled_graphs	10
sample_bias	0.0, 0.5
batch_size	16, 64
seed	74, 75, 76

Table 4.2: Grid Search Sweep Configuration

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 PYTORCH REIMPL.

Additionally, experiment in the collective setting with the configuration and setup described for inductive setting - difference lies test data seen during training. Used for fair comparison with the original paper and the replication paper.

4.4.2 PGExplainer applied to NeuroSAT

We create required data with provided methods, add unsat cores and MUSs as gt
Generated batches of unsat problems that "turned" unsat because of last added clause. 10 literals per problem. Only unsat to test for unsat cores, that only apply for unsat problems. Calculated unsat cores with solver xy by adding negative assumption literals per clause and passing these as assumption for calulation. The edges of the clauses present

in the unsat core were treated as ground truth.

For quant. eval. adapted roc auc as metric as done in PGExplainer. Results seem "good" but qual. eval. shows different result. roc auc bad metric?
For qual. eval. topk(=number of edges in gt) edges of predictions were highlighted to be compared to gt edges. For quant. eval. the edge probabilites were compared to gt with 1s for edges in gt and 0s for rest.

Chapter 5

Results

5.1 PGExplainer in the inductive setting

HERE GO THE RESULTS OF THE EXPERIMENTS, ALSO COMPARED TO THE OTHER PAPERS. INCLUDES, TABLES, PLOTS ETC.
We use normalization in our downstream models, though it is not described in the paper, since it is used in the code. We also experimented with the effects of the use of normalization since it seems to be relevant to the performance of the explainer.
The explainer is trained and evaluated on the same data. We also run experiments with added train/test splits TODO!
Not all data is fed into the explainer: BA-Shapes: BA-Community: Tree-Cycles: "First"/to base graph attached node of each motif in graph is used. Tree-Grid: All Motif nodes are used BA-2Motif: All graphs are used MUTAG: Only the graphs with an available ground truth are used. GT exist for mutagenic graphs that have either chemical groups NH2 or NO2. We later discuss if these selections make sense and run experiments with different data selections.
Quantitative: 10 explainer runs on one downstream model; Calculate ROC-AUC over ALL graphs/nodes in each run; Qualitative: Original uses a threshold; we instead take the topK nodes according to the dataset/motif as an explanation
We also discuss if treating the number of k edges as a hyperparameter dependant on the downstream task makes sense and propose having the network learn it, to improve generalizability and allow the explainer to work on data with varying size.

CPU vs GPU:
It is important to highlight that our code achieved better and way more stable results for BA-2Motif when trained on a gpu instead of cpu.
Ba-Shapes, Tree-Cycles and MUTAG results achieved were identical.
Ba-Community and Tree-Grid achieved very slightly better results on CPU.

Sweeps: (Params ordered by importance)
BA-Shapes: higher size reg -¿ 0.1; lower entropy reg -¿ 0.01; lr and tT very low impact but slightly higher -¿ 0.01 and 5. Note that Loss curve jumps on most runs! (logical-sweep-94 and restful-sweep-92 have clean loss) TRY HIGHER SIZE REG AND LOWER ENTROPY REG

BA-Community: lr 0.0001 too low, not working -¿ 0.003; lower entropy -¿ 0.1; higher size -¿ 0.1; TRY MORE SEEDS, LR, EPOCHS?

Tree-Cycles: high lr -¿ 0.01 ; lower entropy reg -¿ 0.1/0.01; higher size reg -¿ 0.1/0.01 ; lower tT -¿ 1. TRY WITH MORE SEEDS FOR ENT, SIZE, TEMP? Confirmed higher size reg -¿ 0.1; lower entropy reg -¿ 0.01; temp really low impact, tendency higher. TRY 30 EPOCHS???

Tree-Grid: high lr -¿ 0.01; high size reg -¿ 1; higher entropy reg? -¿ 10/1, high tT -¿ 5. TRY MORE SEEDS FOR ENTROPY REG -¿ not quite clear, tendency lower; MAYBE EVEN HIGHER LR -¿ No

BA-2Motif: RUN ON GPU - Not the cause. Cause for better results were features of 1 instead of 0.1! However, good results achieved on BA2-Motif dataset from pyg, not original one.
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.
MUTAG: Low lr -¿ 0.0003; low entropy reg(high impact, but highest AUC runs vary) -¿ 0.1; low tT -¿ 1; less epochs -¿ 20; low size reg -¿ 0.005(/0.01); Loss is messy and AUC seems to decrease over time! lr 0.0001 worse, entropy reg 0.1/0.01 has zero effect -¿ 0.1

Effects of selected motif nodes for Node task: Compare Tree-Grid/Tree-Cycles performance when using all/one node per motif...
RESULTS OBTAINED IN COLLECTIVE SETTING:

Accuracy	**BA-S**	**BA-C**	**Tree-C**	**Tree-G**	**BA-2m**	**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 5.1: Compact accuracy table for Node and Graph Classification.

5.2 PGExplainer applied to NeuroSAT

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.

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.

Bibliography

[1] Réka Albert and Albert-László Barabási. "Statistical mechanics of complex networks". In: *Reviews of modern physics* 74.1 (2002), p. 47.

[2] A.S. Asratian, T.M.J. Denley, and R. Häggkvist. *Bipartite Graphs and their Applications*. Cambridge Tracts in Mathematics. Cambridge University Press, 1998. ISBN: 9781316582688. URL: https://books.google.de/books?id=l8fLCgAAQBAJ.

[3] Federico Baldassarre and Hossein Azizpour. "Explainability techniques for graph convolutional networks". In: *arXiv preprint arXiv:1905.13686* (2019).

[4] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. "A comprehensive survey of graph embedding: Problems, techniques, and applications". In: *IEEE transactions on knowledge and data engineering* 30.9 (2018), pp. 1616–1637.

[5] Augustin Cauchy et al. "Méthode générale pour la résolution des systemes d'équations simultanées". In: *Comp. Rend. Sci. Paris* 25.1847 (1847), pp. 536–538.

[6] Stephen A Cook. "The complexity of theorem-proving procedures". In: *Logic, automata, and computational complexity: The works of Stephen A. Cook*. 2023, pp. 143–152.

[7] T.M. Cover and J.A. Thomas. "Entropy, Relative Entropy, and Mutual Information". In: *Elements of Information Theory*. John Wiley & Sons, Ltd, 2005, pp. 13–55. ISBN: 9780471748823. DOI: https://doi.org/10.1002/047174882X.ch2. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/047174882X.ch2. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/047174882X.ch2.

[8] Piotr Dabkowski and Yarin Gal. "Real time image saliency for black box classifiers". In: *Advances in neural information processing systems* 30 (2017).

[9] Reinhard Diestel. "Random Graphs". In: *Graph Theory*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 323–345. ISBN: 978-3-662-53622-3. DOI: 10.1007/978-3-662-53622-3_11. URL: https://doi.org/10.1007/978-3-662-53622-3_11.

[10] Niklas Eén and Niklas Sörensson. "An extensible SAT-solver". In: *International conference on theory and applications of satisfiability testing*. Springer. 2003, pp. 502–518.

[11] Matthias Fey and Jan E. Lenssen. "Fast Graph Representation Learning with PyTorch Geometric". In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2019.

[12] Hongyang Gao and Shuiwang Ji. "Graph u-nets". In: *international conference on machine learning*. PMLR. 2019, pp. 2083–2092.

[13] Edgar N Gilbert. "Random graphs". In: *The Annals of Mathematical Statistics* 30.4 (1959), pp. 1141–1144.

[14] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[16] Marco Gori, Gabriele Monfardini, and Franco Scarselli. "A new model for learning in graph domains". In: *Proceedings. 2005 IEEE international joint conference on neural networks, 2005.* Vol. 2. IEEE. 2005, pp. 729–734.

[17] Wenxuan Guo et al. "Machine learning methods in solving the boolean satisfiability problem". In: *Machine Intelligence Research* 20.5 (2023), pp. 640–655.

[18] Geoffrey E Hinton et al. "Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv preprint arXiv:1207.0580* (2012).

[19] Lars Holdijk et al. "[Re] Parameterized Explainer for Graph Neural Network". In: *ML Reproducibility Challenge 2020*. 2021.

[20] Qiang Huang et al. "Graphlime: Local interpretable model explanations for graph neural networks". In: *IEEE Transactions on Knowledge and Data Engineering* 35.7 (2022), pp. 6968–6972.

[21] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. pmlr. 2015, pp. 448–456.

[22] Eric Jang, Shixiang Gu, and Ben Poole. "Categorical reparameterization with gumbel-softmax". In: *arXiv preprint arXiv:1611.01144* (2016).

[23] Thomas N Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks". In: *arXiv preprint arXiv:1609.02907* (2016).

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012).

[25] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: `10.1109/5.726791`.

[26] Yujia Li et al. “Gated graph sequence neural networks”. In: *arXiv preprint arXiv:1511.05493* (2015).

[27] Petro Liashchynskyi and Pavlo Liashchynskyi. “Grid search, random search, genetic algorithm: a big comparison for NAS”. In: *arXiv preprint arXiv:1912.06059* (2019).

[28] Zhiyuan Liu and Jie Zhou. “Introduction”. In: *Introduction to Graph Neural Networks*. Cham: Springer International Publishing, 2020, pp. 1–3. ISBN: 978-3-031-01587-8. DOI: 10.1007/978-3-031-01587-8_1. URL: https://doi.org/10.1007/978-3-031-01587-8_1.

[29] Dongsheng Luo et al. “Parameterized explainer for graph neural network”. In: *Advances in neural information processing systems* 33 (2020), pp. 19620–19631.

[30] Yao Ma and Jiliang Tang. *Deep learning on graphs*. Cambridge University Press, 2021.

[31] Chris J Maddison, Andriy Mnih, and Yee Whye Teh. “The concrete distribution: A continuous relaxation of discrete random variables”. In: *arXiv preprint arXiv:1611.00712* (2016).

[32] Joao P Marques-Silva and Karem A Sakallah. “GRASP: A search algorithm for propositional satisfiability”. In: *IEEE Transactions on Computers* 48.5 (1999), pp. 506–521.

[33] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[34] Christopher Morris et al. “Weisfeiler and leman go neural: Higher-order graph neural networks”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 33. 01. 2019, pp. 4602–4609.

[35] Mohd Halim Mohd Noor and Ayokunle Olalekan Ige. “A Survey on State-of-the-art Deep Learning Applications and Challenges”. In: *arXiv preprint arXiv:2403.17561* (2024).

[36] A Paszke. “Pytorch: An imperative style, high-performance deep learning library”. In: *arXiv preprint arXiv:1912.01703* (2019).

[37] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. “” Why should i trust you?” Explaining the predictions of any classifier”. In: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 2016, pp. 1135–1144.

[38] Eve Richardson et al. “The receiver operating characteristic curve accurately assesses imbalanced datasets”. In: *Patterns* 5.6 (2024), p. 100994. ISSN: 2666-3899. DOI: https://doi.org/10.1016/j.patter.2024.100994. URL: https://www.sciencedirect.com/science/article/pii/S2666389924001090.

[39] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[40] Franco Scarselli et al. "The Graph Neural Network Model". In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80. DOI: 10.1109/TNN.2008.2005605.

[41] Bart Selman, Henry A Kautz, Bram Cohen, et al. "Local search strategies for satisfiability testing." In: *Cliques, coloring, and satisfiability* 26 (1993), pp. 521–532.

[42] Daniel Selsam et al. "Learning a SAT solver from single-bit supervision". In: *arXiv preprint arXiv:1802.03685* (2018).

[43] Hanqing Sun et al. "Supervised biadjacency networks for stereo matching". In: *Multimedia Tools and Applications* 83 (June 2023), pp. 1–26. DOI: 10.1007/s11042-023-15362-5.

[44] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. "Finding Minimal Unsatisfiable Cores of Declarative Specifications". In: *FM 2008: Formal Methods*. Ed. by Jorge Cuellar, Tom Maibaum, and Kaisa Sere. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 326–341. ISBN: 978-3-540-68237-0.

[45] Keyulu Xu et al. "How powerful are graph neural networks?" In: *arXiv preprint arXiv:1810.00826* (2018).

[46] Rex Ying et al. "Graph convolutional neural networks for web-scale recommender systems". In: *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 2018, pp. 974–983.

[47] Zhitao Ying et al. "Gnnexplainer: Generating explanations for graph neural networks". In: *Advances in neural information processing systems* 32 (2019).

[48] Hao Yuan et al. "Explainability in graph neural networks: A taxonomic survey". In: *IEEE transactions on pattern analysis and machine intelligence* 45.5 (2022), pp. 5782–5799.

[49] Hao Yuan et al. "On explainability of graph neural networks via subgraph explorations". In: *International conference on machine learning*. PMLR. 2021, pp. 12241–12252.

[50] Muhan Zhang and Yixin Chen. "Link prediction based on graph neural networks". In: *Advances in neural information processing systems* 31 (2018).

Appendix A

Supplementary Material

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

PGExlpainer algorithms

Algorithm 1 Training Algorithm for Explaining Node Classification.

Require: Input graph $G_0 = (\mathcal{V}, \mathcal{E})$, node features X, node labels Y, set of instances to be explained $\mathcal{I}$, trained GNN model: $\text{GNNE}_{\Phi_0}(\cdot)$ and $\text{GNNC}_{\Phi_1}(\cdot)$, parameterized explainer MLP Ψ.

for each node $i \in \mathcal{I}$ **do**
  $G_0^{(i)} \leftarrow$ extract the computation graph for node i.
  $Z^{(i)} \leftarrow \text{GNNE}_{\Phi_0}(G_0^{(i)}, X)$.
  $Y^{(i)} \leftarrow \text{GNNC}_{\Phi_1}(Z^{(i)})$.
end for
for each epoch **do**
  for each node $i \in \mathcal{I}$ **do**
    $\Omega \leftarrow$ latent variables calculated with (10).
    for $k \leftarrow 1$ to K **do**
      $G_s^{(i,k)} \leftarrow$ sampled from (4).
      $\hat{Y}_s^{(i,k)} \leftarrow \text{GNNC}_{\Phi_1}(\text{GNNE}_{\Phi_0}(G_s^{(i,k)}, X))$.
    end for
  end for
  Compute loss with (9).
  Update parameters Ψ with backpropagation.
end for

Algorithm 2 Training Algorithm for Explaining Graph Classification.

Require: A set of input graphs with i-th graph represented by $G_0^{(i)}$, node features $X^{(i)}$, label $Y^{(i)}$, trained GNN model: $\text{GNNE}_{\Phi_0}(\cdot)$ and $\text{GNNC}_{\Phi_1}(\cdot)$, parameterized explainer MLP Ψ.

for each graph $G_0^{(i)}$ **do**
  $Z^{(i)} \leftarrow \text{GNNE}_{\Phi_0}(G_0^{(i)}, X^{(i)})$.
  $Y^{(i)} \leftarrow \text{GNNC}_{\Phi_1}(Z^{(i)})$.
end for
for each epoch **do**
  for each graph $G_0^{(i)}$ **do**
    $\Omega \leftarrow$ latent variables calculated with (11).
    for $k \leftarrow 1$ to K **do**
      $G_s^{(i,k)} \leftarrow$ sampled from (4).
      $\hat{Y}_s^{(i,k)} \leftarrow \text{GNNC}_{\Phi_1}(\text{GNNE}_{\Phi_0}(G_s^{(i,k)}, X^{(i)}))$.
    end for
  end for
  Compute loss with (9).
  Update parameters Ψ with backpropagation.
end for

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.**