Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

How to combine Augmentations for Graph Contrastive Learning

Christian Stegmaier

Course of Study:	Informatik
Examiner:	Prof. Dr. Christian Becker
Supervisor:	Michael Schramm, M.Sc.
Commenced:	June 15, 2022
Completed:	December 15, 2022

Abstract

Graph neural networks (GNNs) are a topic of increasing interest in recent years. They have the potential to handle irregularly structured data, which is of great interest for graph-structured data. Like other areas, graph-structured data suffers from a lack of data. The available data consists of only a small portion of labeled data, while the majority is unlabeled. Semi-supervised learning with contrastive learning has been successfully applied in image representation learning to solve this problem. Contrastive learning relies on good augmentations, which is more complicated for graph-structured data and still needs further exploration.

In this thesis, we attempt to solve the problem by creating multiple new augmentations and comparing them to existing ones. We evaluate the performance of these single augmentations and test the combination of augmentations with different augmentation ratios. Additionally, we further improve the results by testing different loss functions. Eventually, we test transfer learning and warm-starting the neural network with the same and different datasets. Ultimately, we give an outlook into improved ideas to individually select augmentations for each graph and connect them to recent research that uses generators to create augmentations.

Kurzfassung

Graph Neural Networks (GNNs) sind in den letzten Jahren ein immer wichtigeres Thema geworden. Sie haben das Potenzial, unregelmäßig strukturierte Daten zu verarbeiten, was bei Graphen-basierten Daten von großem Interesse ist. Wie auch in anderen Bereichen leiden Graphen-basierte Daten unter einem Mangel an Daten. Die verfügbaren Daten bestehen nur zu einem kleinen Teil aus Daten mit Labeln, während der größte Teil ohne Label ist. Semi-supervised Learning nutzt Contrastive Learning, um dieses Problem zu lösen und wird beim Lernen von Bild-Repräsentationen erfolgreich eingesetzt. Die Qualität von Contrastive Learning hängt davon ab, gute Augmentations zu finden, ein Thema, das bei Graphen-basierten Daten komplizierter ist und weiter erforscht werden muss.

In dieser Thesis versuchen wir, das Problem zu lösen, indem wir mehrere neue Augmentations erstellen und sie mit bestehenden vergleichen. Wir evaluieren die Leistung dieser einzelnen Augmentations und testen die Kombination von unterschiedlichen Augmentations-Ratios. Darüber hinaus verbessern wir die Ergebnisse weiter, indem wir verschiedene Loss-Functions testen. Schließlich testen wir das Transfer-Learning und das Warmstarting des neuronalen Netzes mit denselben und verschiedenen Datensätzen. Abschließend geben wir einen Ausblick auf verbesserte Ideen zur individuellen Auswahl von Augmentations für jeden Graphen und stellen eine Verbindung zu neueren Forschungsergebnissen her, die Generatoren zur Erstellung von Augmentations verwenden.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

CNN convolutional neural network. 15

GCN graph convolution network. 17

GIN graph isomorphism network. 17

GNN graph neural network. 15

HITS Hyperlink-Induced Topic Search. 37

InfoMax mutual information maximization. 20

MLP multilayer perceptron. 17

MST minimum spanning tree. 44

NN neural network. 23

NT-Xent loss normalized temperature-scaled cross entropy loss. 20

PCA principal component analysis. 22

ReLU rectified linear unit. 18

SS sector area similarity. 32

TS triangle area similarity. 32

TS-SS Triangle Area Similarity – Sector Area Similarity. 28

t-SNE t-distributed stochastic neighbor embedding. 22

UMAP uniform manifold approximation and projection. 22

VGAE variational graph auto-encoder. 25

WL-test Weisfeiler-Lehman graph isomorphism test. 19

1 Introduction

The interest in Deep Learning has continuously increased in the last years. The most well-known examples are convolutional neural networks (CNNs) from visual representation learning. Other fields like graph representation learning are less explored but are catching up recently. Compared to CNNs, the graph-structured data introduces additional problems, which increase the difficulty of successful learning. Mainly the fact that graph-structured data has very diverse fields and is generally non-euclidian structured like social networks, citation networks, or biochemical networks with additional possible parameters like edge weights. These different and complex setups make CNNs ineffective for graph-structured data. The current solutions for graph representation learning are graph neural networks (GNNs) [1], which have shown to handle these datatypes while using neural networks and solve different tasks like node classification and link classification [2] or graph classification [3]. Many real-world applications like Protein-Protein Interactions [4], Drug Discovery [5] or Traffic Forecasting [6] depend on GNNs. One key element of GNNs is the neighborhood aggregation scheme (message passing), in which nodes exchange information with their neighbors.

Currently, GNNs have been used primarily in the context of end-to-end training tasks in the supervised context, where large amounts of labeled data are available for general and effective results. In contrast, most real-world tasks lack enough labeled data since areas like biochemistry require expensive and extensive testing to obtain labels.

Semi-supervised learning is an approach to solve this problem, where a small part of the data is labeled. In contrast, most of the data is unlabeled and used in a pre-training step in combination with contrastive learning.

Contrastive learning for visual representation learning uses different views (augmentations) of the same image as positive samples. Augmentations of different images are negative samples. This way, different representations of the same image are pushed together in the embedding space. Prominent examples from visual representation learning are SimCLR [7] and Moco [8]. Images have many suitable augmentations, like crop, resize, flip, rotate, cutout, color distortion, noise, or blur. As previously mentioned, graph-structured data introduces additional difficulties because the data is of a non-Euclidian structure.

In a recent paper by You et al. [9], the authors tested the performance of four augmentations and their combination: Node Dropping, Attribute Masking, Edge Perturbation, and Subgraph. They conclude that pre-training with augmentations is beneficial, especially combining two different augmentations.

While they show that this approach works and that the contrastive learning approach increases the accuracy of graph classification, some problems remain. One problem is that this requires costly testing and handpicking augmentations specifically for each dataset. It also needs to be investigated if other and better augmentations exist and how much of this augmentation should be applied, i.e.,

the augmentation ratio. The second problem is that different datasets perform well with different augmentations, while some combinations perform worse than those using no pre-training. The selection of good augmentations is, therefore, crucial.

An improved idea of their contrastive learning approach is not to select the augmentation but instead let the model decide which augmentations are best for the dataset. The follow-up paper by You et al. [10] still considers the previous four augmentations. However, instead of manually picking a specific augmentation, the model estimates probabilities for each augmentation depending on how high the contrastive loss for these augmentations is. This approach partly solves the problem of manually deciding which augmentation to use for the dataset. At the same time, what remains still unknown is what additional augmentations may exist and how they perform.

The essential idea behind the augmentations is to reduce the graphs to their important parts by removing unimportant details. As previously seen, different datasets have different optimal augmentations and augmentation rates. A likely assumption is that every graph inside the same dataset has a different augmentation which is optimal. Big social graphs might benefit from higher augmentation ratios, while smaller molecular graphs tolerate less. This assumption leads to another promising approach, not to specify explicit augmentations beforehand but to define a generator that learns to modify graphs individually. This way, we do not limit the training by a selected number of augmentations, and the generator can decide for each graph how many nodes get dropped or how many node attributes get masked. Different approaches to building generators exist.

You et al. [11] implement a generator to obtain the views for contrastive learning by random walks. Suresh et al. [12] use a generator that samples edges, while Yin et al. [13] use a generator that decides for each node if it should be dropped or kept and also offers the possibility to mask nodes. For some datasets, these generators reach promising results, while for other datasets, their generators still need to be optimized.

In this thesis, we test the influence of augmentations on different graph datasets. We introduce 15 new augmentations and eight existing approaches and evaluate their performance on 12 different datasets. Afterward, we test the combination of the most promising augmentations under different augmentation ratios. We also propose multiple improvements to the contrastive learning pipeline. First, we modify state-of-the-art contrastive loss functions by lowering the number of negative samples and increasing the number of positive samples. Additionally, we introduce four loss functions that use different distance metrics and weights. We also test the effect of transfer learning with improvements to warm-starting pre-trained models on different datasets. Finally, we introduce our implementation of picking individual augmentations from the embedding space.

Chapter 2 explains the technological background used in this thesis. In Chapter 3, we define the fundamental problems we will solve. Chapter 4 briefly explains related work in pre-training visual models and the first approaches that adapt them for graph models. Chapter 5 explains the assumptions of our underlying system model, and in Chapter 6, we define our augmentations. Chapter 7 evaluates our augmentations and approaches before we present our conclusion in Chapter 8 and give an outlook for future work.

2 Background

In this chapter, we describe the theoretical approaches that currently exist to handle the task of graph representation learning. We start by looking into the different neural networks that exist for graph data, mainly GNNs, graph convolution networks (GCNs), and graph isomorphism networks (GINs). Additionally, we describe the theoretic concept behind semi-supervised learning in combination with contrastive learning and augmentations. We also introduce the concept of transfer learning and warm-starting neural networks. We end the chapter with a short description of dimensionality reduction techniques to visualize pre-training results.

2.1 GNN (Graph Neural Network)

Compared to image representation learning, graph-structured data introduces additional difficulties. The main problem is the irregular data structure. While CNNs are very successful at handling regular image data [14], they can not handle graph-structured data. GNNs solve these problems and are currently the de facto standard for graph representation learning. We denote a graph as G = (V, E), where V is the set of nodes and E is the set of edges. Additionally, we have node features x_v for $v \in V$. The goal is to create a representation h_v for $v \in V$.

The key concept of GNNs is neighborhood aggregation, also called message passing. With message passing, the GNN can calculate the representation by collecting information about the graph structure and the node features from their neighborhood. Figure 2.1 shows the concept of message passing. We show the process for two nodes of the graph. In practice, all nodes in the graph do message passing. The current node (yellow) aggregates more information in each layer k.

We define the detailed propagation rules similarly to You et al. [9]. Equation (2.1) shows a K-layer GNN propagation rule.

$$a_v^{(k)} = \text{AGGREGATE}^{(k)}\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right) \tag{2.1}$$

$$h_v^{(k)} = \text{COMBINE}^{(k)}\left(h_v^{(k-1)}, a_v^{(k)}\right) \tag{2.2}$$

We denote $h_v^{(k)}$ as the feature vector in the k-th layer of node v and the embedding of node v. We initialize h with the node features $h_v^{(0)} = x_v$. $\mathcal{N}(v)$ represents the neighbors of node v.

For the downstream tasks like classification, we are interested in the graph-level representation z_g, created by pooling and a multilayer perceptron (MLP) as shown in Equation 2.3.

$$F(g) = \text{POOL}\left(\left\{h_n^{(k)} : v_n \in V\right\}\right) \tag{2.3}$$

$$z_g = \text{MLP}\left(F(g)\right) \tag{2.4}$$

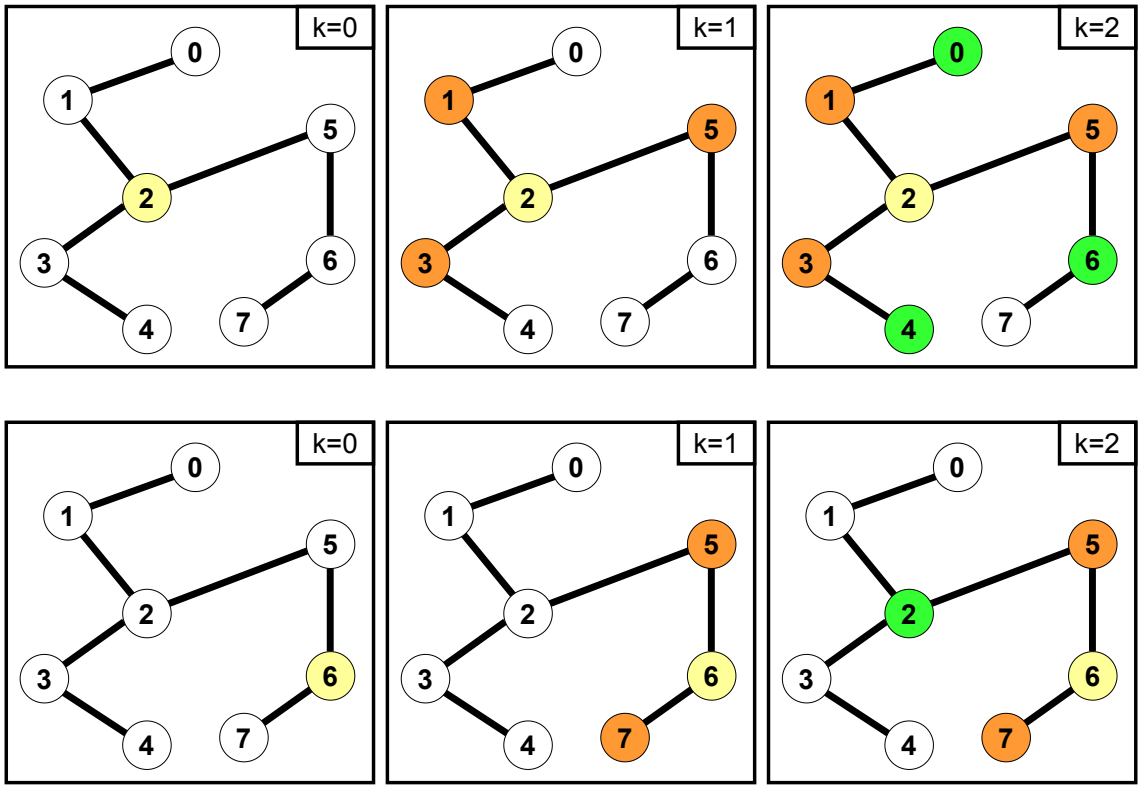

Figure 2.1: Message Passing for two nodes (yellow). Each layer k increases the depth of neighbor information.

Multiple different implementations of GNNs exist. We focus on GCNs [2] and GINs [3]

2.2 GCN (Graph Convolution Network)

GCNs are defined by Kipf et al. [2], and extend GNNs by the idea of convolution. Convolutions extract features like shapes and edges from CNNs. Pooling is applied after convolutions and reduces the size of the image while preserving these key features. Similarly, GCNs use convolutions to extract spatial information from the graphs. Pooling in GNNs takes multiple nodes and aggregates the information. In their paper, Kipf et al. state the layer-wise propagation as shown in Equation 2.5.

$$H^{(k+1)} = \sigma\left(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}H^{(k)}W^{(k)}\right) \tag{2.5}$$

With the Weight matrix W^k, a non-linear activation function σ like the rectified linear unit (ReLU), and a normalization step $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$.

They use element-wise mean pooling, which is rewritable similar to the format in Section 2.1. The final formula is Equation 2.6.

$$h_v^{(k)} = \text{ReLU}\left(\text{W} \cdot \text{MEAN}\left\{h_u^{(k-1)}, \forall u \in N(v) \cup \{v\}\right\}\right) \tag{2.6}$$

2.3 GIN (Graph Isomorphism Network)

Xu et al. [3] present a theoretical framework to analyze the expressive power of different GNNs like GCN [2] and GraphSAGE [15]. During their testing, they find graph structures they can not distinguish by these models and introduce their model, GIN. In contrast to GCNs, GINs extend GNNs with graph isomorphism layers. These layers perform operations such as relabeling or permutating nodes. GINs are closely related to the Weisfeiler-Lehman graph isomorphism test (WL-test). The graph isomorphism test asks if two graphs are topologically identical. Since there is yet to be an algorithm with a polynomial-time algorithm, the WL-test approximates this problem. The test can tell if two graphs are non-isomorphic, but it does not guarantee they are isomorphic. It distinguishes a broad class of graphs by aggregating neighborhoods similar to GNNs.

The update rule of GIN is stated in Equation 2.7, with ϵ as the importance of the target node compared to the neighbors.

$$h_v^{(k)} = \mathrm{MLP}^{(k)}\left(\left(1+\epsilon^{(k)}\right)\cdot h_v^{(k-1)} + \sum_{u\in N(v)} h_u^{(k-1)}\right) \tag{2.7}$$

2.4 Semi-Supervised Learning

Neural networks can solve a wide range of supervised tasks like image classification. The success of these approaches often relies on large amounts of labeled data. In most real-world scenarios, labeled data is often expensive or unavailable. Most practical examples have a small amount of labeled data, and a more significant amount of unlabeled data is available (e.g., 10% of the data is labeled). While it is still possible to use supervised learning on the labeled data, results would suffer from the small data. Good generalization of the model relies on large training data.

Semi-supervised learning combines supervised and unsupervised learning [16]. The unlabeled data still contains valuable information. For graphs, topological information is an example. We can use this information without the label during unsupervised learning. An example of this approach is contrastive learning which we describe in Section 2.6. Before the supervised training, we add an unsupervised pre-training step.

Recently different semi-supervised approaches have shown their success. Semi-supervised approaches in image representation learning like BYOL from Grill et al. [17] even outperform supervised approaches. Even though the label information is missing, the increased data size can still result in better generalizability and increased classification accuracy.

2.5 Augmentations

Augmentations are different representations of the data. The assumption is that we change the data slightly but keep the critical part of the data, and the semantic label remains the same. For images, examples would be the rotation of the image, adding noise, or cropping the image. Graph-structured data can use node dropping, edge dropping, attribute masking, and building a subgraph as the most apparent augmentations. We show examples for these four augmentations in Figure 2.2. However,

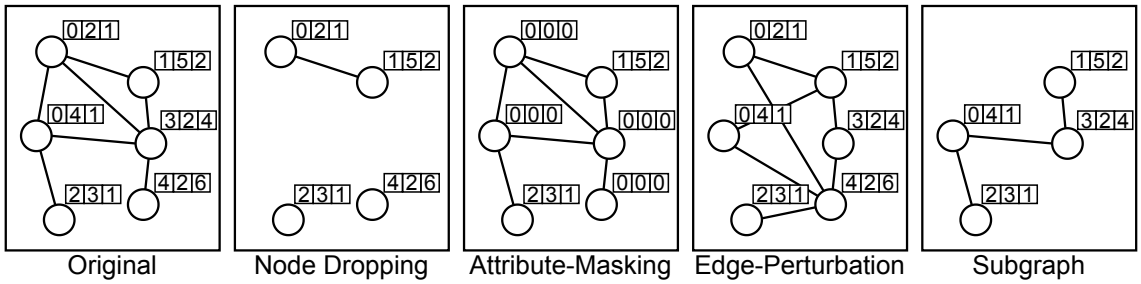

Figure 2.2: The image shows the original graph on the left. Nodes have three attributes. The four right images show the graph after applying each of the four basic augmentations.

graphs are much more challenging to augment because they consist of different geometry, and their topology is essential information. So dropping the wrong node could result in an unconnected graph, which might change the whole graph and the label. As shown in a recent paper by You et al. [9], if you find good augmentations, contrastive learning can use them, where the different augmentations represent positive examples of the same graph.

2.6 Contrastive Learning

Contrastive learning is successfully applied to visual representation learning [7] and has been of interest in the recent graph representation learning research [9]. Contrastive learning relies on the principle of mutual information maximization (InfoMax), in which we compare different augmented forms of the data. Ideally, each augmentation identifies the original data perfectly so the mutual information does not decrease. As mentioned in Section 2.4, we use contrastive learning with unlabeled data as part of semi-supervised learning. In the pre-training step, contrastive learning trains the model, and afterward, we train the downstream task on the labeled data (e.g., classification).

Hadsell et al. [18] introduce the idea of contrastive learning. We create positive samples of our data points by creating augmentations of the same data point as mentioned in Section 2.5. Other data points serve as negative samples. The contrastive loss then tries to bring the positive samples closer together while the negative ones get pushed away. Figure 2.3 shows an example of the idea. We have six different graphs in the 2-D embedding space. Each graph has two augmentations, augmentation 1=i and augmentation 2=j. We use one positive sample as the anchor, here the yellow graph n, with the first augmentation i. Green is the positive sample of graph n with the second augmentation j. Negatives are marked red, all second augmentations of the other graphs. We build the contrastive loss for each graph with positive distances (green) and negative distances (red). We sum up all six losses from the mini-batch in the end. As a result, the loss function pushes the positive samples closer together while the negatives get pushed away.

Equation (2.8) [19] shows one of the most common loss functions for contrastive learning, the normalized temperature-scaled cross entropy loss (NT-Xent loss). The distance to the single positive sample is used in the numerator and divided by the sum of all negative samples in the denominator. The loss decreases if the positive distance gets smaller or the negative distance increases. Usually, the number of negative samples depends on the batch size N and is very high. Image representation

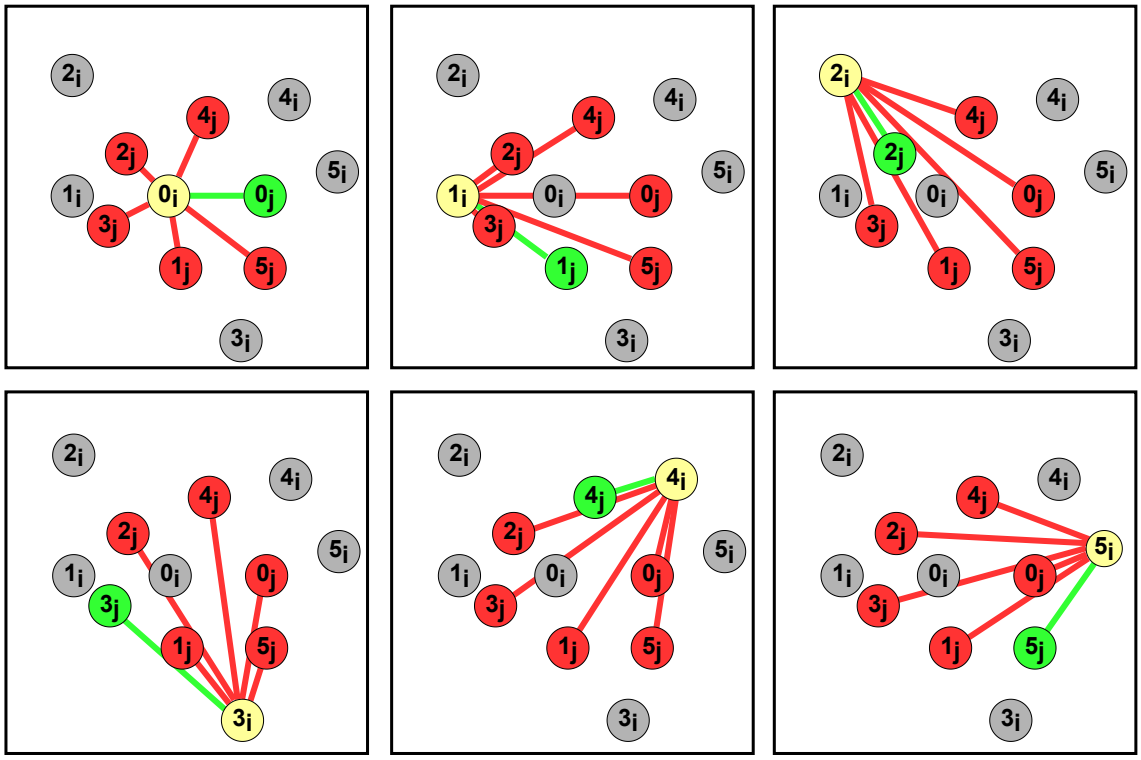

Figure 2.3: Example of the calculation of contrastive loss. We build the loss for each graph with the positive sample (green distance) and the negative samples (red distance). The loss function pushes positive samples closer together while the negative samples get pushed away.

uses a batch size of N=256 to N=4096, where all augmentations of all images except the current one serve as negative samples [7]. The total number of negative samples is 2N-1.

$$l_{i,j} = -\log \frac{\exp(\text{sim}(z_i, z_j)/\tau)}{\sum_{k=1,[k\neq i]}^{2N} \exp(\text{sim}(z_i, z_k)/\tau} \tag{2.8}$$

The similarity function $\text{sim}(z_i, z_j)$ is the cosine similarity (2.9). Other loss functions that use different distance metrics, such as the max-margin loss with the euclidian distance, are also usable and will be investigated in this thesis.

$$\text{sim}(z_{n,i}, z_{n,j}) = z_{n,i}^T z_{n,j} / \|z_{n,i}\| \|z_{n,j}\|) \tag{2.9}$$

You et al. [9] use a slightly modified version of the NT-Xent loss, shown in Equation 2.10. Instead of all 2N-1 negative samples, they only use N-1 samples. We denote the loss for graph n as l_n and sum up all l_n in the batch for the final loss.

$$l_n = -\log \frac{\exp(\text{sim}(z_{n,i}, z_{n,j})/\tau)}{\sum_{n'=1,[n'\neq n]}^{N} \exp(\text{sim}(z_{n,i}, z_{n',j})/\tau} \tag{2.10}$$

2.7 Warm-Starting and Transfer Learning

Warm-starting a neural network describes the general idea of initializing the model with previously collected data. An example would be initializing the model weights with values from a previous training session of the same task. Areas like daily financial data, where new data arrives periodically, profit from warm-starting the model with the previous weights and only train with the new data [20]. Transfer learning is a similar type of machine learning. In contrast to warm-starting, we take a pre-trained model and adapt it to a new task. The new task takes the final weights of the pre-trained model as the initialization of its model. Training the initialized model is less time-consuming than training a model from scratch. This technique can also improve the accuracy of neural networks by allowing them to leverage the additional knowledge learned from the pre-trained model. Transfer learning is especially useful when the training data for the target task is limited. Successfully using transfer learning is a challenging task. We need a deep understanding of the data and the task we want to solve. The pre-training data and model should both be similar to the new task. We also need to know our model and fine-tune the parameters for optimal results. Various areas use transfer learning, including computer vision [21] and natural language processing [22].

2.8 Dimensionality Reduction

Visualizing the results of representation learning often relies on dimensionality reduction. The high-dimensional embedding space needs to be projected onto a two-dimensional plane. Multiple dimensionality reduction methods exist. Principal component analysis (PCA) [23] is a linear dimensionality reduction that finds the direction of maximum variance on the data. The data gets transformed into a new set of variables, a linear combination of the original variables. These principal components get ranked in order of importance. PCA keeps the most important components and retains most of the information. T-distributed stochastic neighbor embedding (t-SNE) [24] is a nonlinear dimensionality reduction that preserves the local structure of the data. Points close together in the high-dimensional space will also be close together in the low-dimensional space. Uniform manifold approximation and projection (UMAP) [25] is a nonlinear dimensionality reduction that preserves the data's important global and local structures. The main difference between t-SNE and UMAP is how they measure the distance between data points. T-SNE measures the similarity between points with the probability that one data point will pick another as its neighbor. UMAP uses the distance between the points. The distance allows UMAP to capture more of the global structure.

3 Problem Statement

We introduce the main problems that we solve in this thesis. Multiple problems have to be solved. Firstly, the general problem of classifying graphs which we solve by using a neural network (NN). The neural network can not work with the graph data directly, We need to transform it into a representation that the neural network can use. This problem is called graph representation learning, and the representation should retain the property that similar graphs should have a similar representation. A prominent example of representation learning is word embeddings. They embed the word into a vector space such that similar words are close to each other in the embedding space. GNNs are the state-of-the-art solution for graph representation learning, which we use in our work. They embed the graph representation into an n-dimensional embedding space, an n-dimensional vector.

While the classification problem has been heavily researched and delivers good results in supervised learning, this approach relies on lots of labeled data [3], [26]. An additional problem in real-world examples is often the lack of data and, more importantly, the lack of labeled data. While we have many unlabeled data samples, only a few have labels. Due to the small amount of labeled data we have, supervised learning suffers from overfitting and bad generalizability. Bad generalizability reduces the classification accuracy of the model for new data. To reduce the overfitting, we add an unsupervised pre-training step, in which we train with many unlabeled data, followed by the supervised training with a few labeled data samples. We train the embedding during the pre-training, which we then use as the initialization for the supervised training. Otherwise, we would randomly initialize the embedding. The pre-training acts as a regularizer and increases the model's generalizability [27]. We can increase the classification accuracy if we find a good initialization of the embedding. We show our setup, with a simplified GNN, that combines the pre-training with fine-tuning step in Figure 3.1. We copy the weights and the embedding from the first layers of the pre-training model and use them in the fine-tuning model as initialization.

Image representation learning uses contrastive learning to find good embeddings. Prominet examples are SimCLR [7] and Moco [8]. Learning embeddings for graph-structured data is less explored and introduces additional difficulties. While image datasets like CIFAR-10 [28] have 60.000 data samples, and ImageNet [29] consists of 14.000.000 images. Most graph datasets are in the range of 1000-5000 graphs. The limited dataset size makes good pre-training strategies even more important.

Contrastive learning relies on data augmentations. Augmentations on the same data act as positive samples and should be close together in the embedding space. Different graphs are negative samples, which should be further away in the embedding space. It is critical to select good augmentations because they define the quality of the embedding. We must find different representations of the same data sample that we push closer together in the embedding space. The key factor behind this approach is that the different data representations still keep the most important information and their class label. A good embedding would place these representations next to each other in the

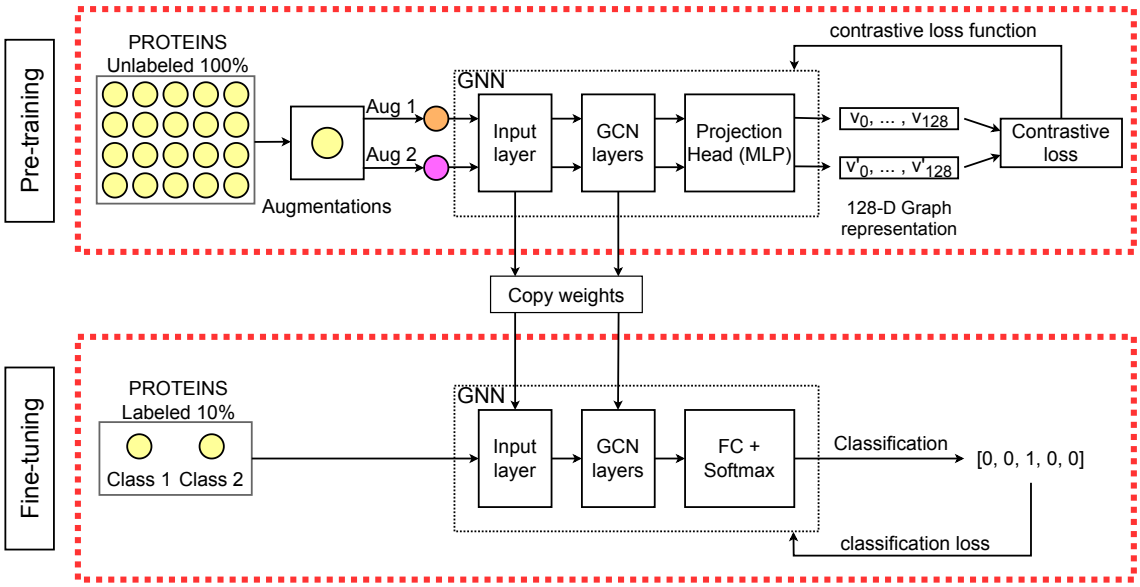

Figure 3.1: The image shows the setup we use for pre-training and fine-tuning (with a simplified GNN). We pre-train the layers of our network and use these weights as initialization during fine-tuning. Additionally, we replace the projection head with a fully connected layer that uses a softmax to classify the graphs.

embedding space, which we force by pushing them together through the contrastive loss. While images have a variety of augmentations like crop, cutout, color distortion, noise, blur, or rotation [7], graphs are more limited in the range of possible augmentations. Examples like molecules are difficult to handle because removing or changing a single atom or its connection can change the molecule's effect and, therefore, possibly change the class label. This would lead to representations with now different class labels being placed close together in the embedding space, which is the opposite of what we want and decreases the model's accuracy.

4 Related Work

In the following chapter, we address related work on using contrastive learning with augmentations as a pre-training step in visual representation learning. Afterward, we focus on the existing approaches for graph-structured data. We will describe the advantages and disadvantages and discuss still open challenges in these approaches.

The current state-of-the-art solutions for visual representation learning are SimCLR by Chen et al. [7], and MoCo by He et al. [8]. They use contrastive learning to solve the task of pre-training. In MoCo, they view contrastive learning as a dictionary look-up task. The dictionary is a dynamic queue with a moving-averaged momentum encoder. They manage to outperform previous end-to-end and memory bank approaches. SimCLR focuses on the composition of data augmentations. They test multiple augmentations like crop, cutout, color distortion, noise, blur, or rotation and their combination. As key results, they introduce a nonlinear head $g(\cdot)$ between the encoder and the contrastive loss. They highlight the augmentation combination's importance and find that large batch sizes benefit the training. The success of both approaches has led to multiple follow-up papers with improvements. SimCLRv2 by Chen et al. [30] uses deeper models, a deeper projection head, and a memory network similar to MoCo. In MoCov2 by Chen et al. [31], the authors introduce a deeper projection head and add more data augmentation.

You et al. [9] build a framework to analyze contrastive learning with augmentations on graph-structured data. They test multiple settings, from unsupervised and semi-supervised, to transfer learning. Their focus is on four essential augmentations, node dropping, attribute masking, edge perturbation, and subgraph. They show that pre-training on graphs increases end-to-end classification accuracy. Further research is necessary to increase the set of augmentations. The main problem is that testing and finding the best augmentation for each dataset is expensive.

In their follow-up paper, You et al. [10] reuse and improve their framework. They add the option to select different augmentation combinations dynamically depending on the dataset. To do that, they calculate the contrastive loss for each augmentation pair in every epoch and estimate probabilities for each augmentation. While this simplifies the selection of augmentations, they still rely on predefined augmentations, and the results are very similar to handpicking the best augmentation.

Finally, You et al. [11] try to solve the previous problem, where the set of augmentations is still predefined. They use a generator built with a combination of a variational graph auto-encoder (VGAE) and a random-walk sampler. As a result, with the information minimization principle, we get an individual augmentation for each graph. More irrelevant information gets discarded, and only the vital part of the data should remain. This approach has promising results for some datasets and does not rely on predefined augmentations. For some datasets, though, this approach is outperformed by extensive hand-picking of augmentations.

The AutoGCL framework by Yin et al. [13] further expands the idea of using a generator with contrastive learning. Their generator uses the node embeddings to create a distribution for each graph, and with the Gumbel-softmax, one of the three augmentations drop, keep, or mask gets applied to each node. They manage to outperform handpicking augmentations on most datasets, yet not all. Additionally, it remains to be seen if edges can and should be considered similar to the nodes and if this improves the results.

Another approach by Suresh et al. [12] suggests using an adversarial approach as the generator. Essentially the generator learns a probability distribution to drop edges. Their approach is in contrast to previous methods that focus on node augmentations. Compared to previous generators like AutoGCL, they outperform them on some datasets, while others still profit from different generators. It remains unclear which approach performs best for an unknown dataset and if both approaches can be combined. It is clear that all proposed generators so far have different strengths and weaknesses, and no single generator is optimal for all datasets.

5 System Model

In the following chapter, we describe the System Model of the thesis. We start with a description of the model and parameters which we use. Then we explain the different loss functions. Afterward, we describe three different methods to handle warm-starting and transfer learning. The next chapter, Chapter 6, covers the details of the augmentations we use.

5.1 Model

We build our framework on top of the framework by You et al. [9]. They follow the general approach from Chen et al. [32]. We start with an unsupervised learning pre-training step, using the unlabeled data for contrastive learning. The following end-to-end fine-tuning task is graph classification with 10% of the labeled data. During the pre-training, two augmentations get selected from the set of 24 augmentations (Identical + 23 augmentations). Pre-training and fine-tuning run for 100 episodes each with a batch size of 128. We use Adam [33] as an optimizer and set the learning rate to 0.001. The size of the hidden dimension is 128.

We give an overview of our layers in Figure 5.1. Pre-training and fine-tuning share the same model except for the last layer. Both models have a feature layer followed by three graph convolution layers. Afterward, we use sum pooling, followed by a fully connected layer. We use batch normalization layers [34] before the layers and a ReLU as an activation function after the layers. In the end, pre-training uses a two-layer perceptron (MLP) as the projection head to create the graph embeddings for contrastive learning. During fine-tuning, we switch the projection head with a fully connected layer with a softmax for classification.

We evaluate the results of the classification in two different ways. We call the first method the best-average-epoch-accuracy (mean-accuracy) and follow the approach from Xu et al. [3]. The accuracy of the single epoch with the best 10-fold cross-validation accuracy averaged over these 10 folds is selected. We call the second approach that Yin et al. [13] use max-accuracy. We use 10-fold cross-validation again. However, we use the highest accuracy for every fold and average the results. The first approach, the mean-accuracy, is generally lower than the max-accuracy but more robust regarding outliers.

5.2 Loss Functions

In the following chapter, we define multiple loss functions that we evaluate later. As a baseline loss function, we use the NT-Xent loss adaptation from You et al. [9] as described in (2.10). The first augmentation of graph n is $z_{n,i}$, while the second augmentation of the same graph n is $z_{n,j}$. Together they are the positive sample, while all other graphs in the batch $z_{n',j}[n' \neq n]$ build the

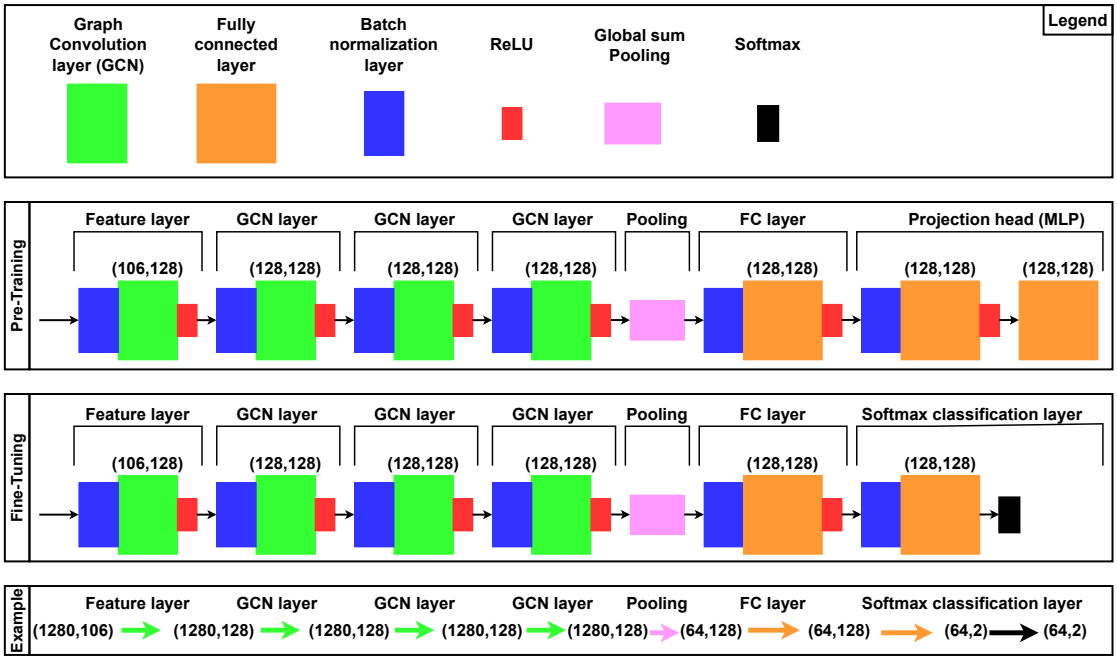

Figure 5.1: Detailed setup of the GNN we use. Both models differ in the last layer. Pre-training uses a projection head for the graph embeddings, while fine-tuning uses a fully connected layer with the softmax to classify the graphs.
Below is an example of fine-tuning. For the example, we use a batch size of 64. Each graph has 20 nodes, each node has an attribute vector of 106, and we have two classes.

|batch size|-1 number of negative samples that we sum up. We define the complete loss for the n-th graph in a mini-batch with $\text{sim}(z_{n,i}, z_{n,j})$ as the cosine similarity in Equation 5.1. For all the different loss functions we calculate the loss of the whole batch by summing up all individual graph losses as shown in Equation 5.2.

$$l_n = -\log \frac{\exp(\text{sim}(z_{n,i}, z_{n,j})/\tau)}{\sum_{n'=1,[n'\neq n]}^{N} \exp(\text{sim}(z_{n,i}, z_{n',j})/\tau)} \tag{5.1}$$

$$l_{\text{Batch}} = \sum_{n=0}^{N} l_n \tag{5.2}$$

We test multiple loss functions and compare them to the NT-Xent loss. We change the similarity measure to the euclidian distance in the first part. For our default batch size of 128, this results in 2 positive and 127 negative samples for each batch. We test a different number for the positive and negative samples in the second part, Afterward, we define a weighting function for the euclidian distance that weights the critical cases more. Additionally, we implement a combination of cosine, and euclidian distance, the Triangle Area Similarity – Sector Area Similarity (TS-SS) [35]. Ultimately, we add an implementation of Triplet loss, a similar formulation to our euclidian loss with only one positive and one negative.

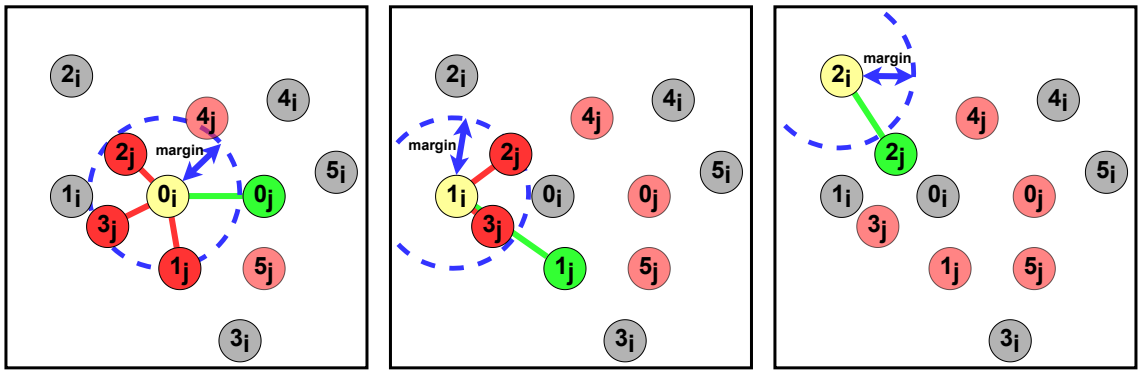

Figure 5.2: The image shows the effect of the margin parameter m (blue) for the max-margin contrastive loss. For the loss, we only consider negative samples closer to the anchor (yellow) than the margin.

5.2.1 Euclidian Distance (Max-Margin Loss)

To apply the euclidian norm as a similarity measure, we can not simply use the NT-Xent loss (5.1) formula. Cosine similarity is normed between -1 and 1, while the euclidian distance is not normed. As a result, the network would infinitely push apart all other graphs, the negative samples, to minimize the loss. This trivial solution is not practical. We need to limit the maximum distance for negative samples. One solution is the formulation of max-margin contrastive loss [18]. As shown in Equation 5.3, we ignore negative samples if they are further away than the margin m. We show the effect of the margin in Figure 5.2. Each graph has two augmentations, augmentation 1=i, and augmentation 2=j. The current anchor (yellow) is graph n, with the first augmentation i. The positive sample (green) is graph n with the second augmentation j. All second augmentations of the other graphs are negative samples (red). We count the negative distances outside the blue margin as zero.

$$l_n = \mathbb{1}_{y_i=y_j}\|z_{n,i} - z_{n,j}\|_2^2 + \mathbb{1}_{y_i\neq y_j}\max(0, m - \|z_{n,i} - z_{n',j}\|_2)^2 \tag{5.3}$$

5.2.2 Less Negative Samples

We test the effect of the number of negatives per batch. Research in image representation learning suggests large batch sizes with lots of negative samples improve the result [7]. To test if this is true for graph representation learning, we modify both formulas from the NT-Xent loss (5.1) and the max-margin loss (5.3). Instead of summing up all negative samples in the batch, we only pick |x| graphs from the batch as negative samples. We show the modified formulas in Equation 5.4 and Equation 5.5.

$$l_n = -\log\frac{\exp(\mathrm{sim}(z_{n,i}, z_{n,j})/\tau)}{\sum_{n'}^{x\subseteq(\mathrm{Batch}\setminus n)}\exp(\mathrm{sim}(z_{n,i}, z_{n',j})/\tau)} \tag{5.4}$$

$$l_n = \|z_{n,i} - z_{n,j}\|_2^2 + \sum_{n'}^{x\subseteq(\mathrm{Batch}\setminus n)}\max(0, m - \|z_{n,i} - z_{n',j}\|_2)^2 \tag{5.5}$$

5.2.3 More Positives Samples

Additionally, we test to increase the number of positive samples. For $k \geq 2$ positive samples, we update the formula as seen in Equation 5.7. For k=2, we get the same result as before, where $z_{n,0}$ is equal to $z_{n,i}$. It describes the anchor graph n under the first augmentation. Graph n under the second augmentation is $z_{n,1}$.

For k>2, we build k-1 new graphs with the second augmentation, from $z_{n,1}$ to $z_{n,k}$. Since the augmentations are not deterministic, we get k-1 similar but unequal graphs. We do the same for the negative samples. If graph n' is selected, we build $z_{n',1}$ to $z_{n',k}$. The number of positive and negative samples increases by a factor of k-1. For k=5 and |x|=1 (one negative sample), we end up with four positive pairs (anchor - positive sample) and four negative pairs (anchor - negative sample).

$$l_n = -\log \frac{\sum_{j=1}^{k} \exp(\text{sim}(z_{n,i}, z_{n,j})/\tau)}{\sum_{j=1}^{k} \sum_{n'}^{x \subseteq (\text{Batch} \setminus n)} \exp(\text{sim}(z_{n,i}, z_{n',j})/\tau)} \tag{5.6}$$

$$l_n = \sum_{j=1}^{k} \|z_{n,0} - z_{n,j}\|_2^2 + \sum_{j=1}^{k} \sum_{n'}^{x \subseteq (\text{Batch} \setminus n)} \max(0, m - \|z_{n,0} - z_{n',j}\|_2)^2 \tag{5.7}$$

The number of negative samples before only changes the loss function and does not influence the training. Increasing the number of positives, on the other hand, results in more training. The default setting, k=2, uses all graphs twice in each episode. If we increase k to 5, we use every graph five times. The training has increased by a factor of 2.5. We must keep this in mind when we evaluate the accuracy of the loss functions. Increasing the number of positives could be beneficial because of the different loss function but also due to increased training.

5.2.4 Weighted Euclidian Max-Margin Loss

We define an adaptation of the euclidian loss function that weights the graphs. We only use one positive and one negative to simplify the weight generation. The loss calculation follows the max-margin formulation we introduced before (5.3). Additionally, we create two weighting functions that give a weight to each loss l_n in the mini-batch. We give more weight to bad samples and reduce the weight of good samples. A sample is bad if the positive distance is high and the negative distance is small. We show an example of the weighting in Figure 5.3. The image on the right side is the worst case because the negative sample is closer to the anchor than the positive sample. It is most important to punish these occurrences. In the left image, we see an unimportant case. The positive distance is already much smaller than the negative distance. We do not assign much weight. Other cases in between, like the image in the middle, get an average weight. We denote the two positives for graph n as $z_{n,i}$ and $z_{n,j}$. The graph $z_{n'_n,j}$ is the randomly selected graph used as the negative sample by graph n. The first weighting function in Equation 5.8 subtracts the negative distance from the positive distance and applies a softmax over the result. The second

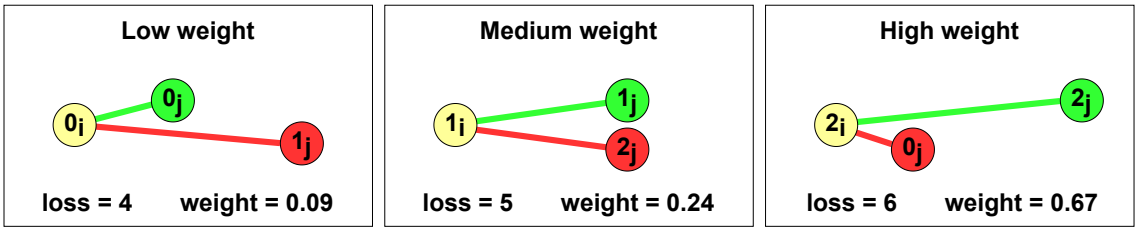

Figure 5.3: The image shows different cases with our weighting function. The left image shows the optimal case when the positive sample (green) is closer to the anchor. We assign a low weight. The right image shows the worst case. The negative sample is way closer to the anchor than the positive sample. The weighting function punishes this case.

weighting function in Equation 5.9 does the same but adds an upper bound to the negative distance to prevent negative samples from being pushed infinitely away. The upper bound is the same margin m from the max-margin loss. Equation 5.10 combines the weight with the loss.

(5.8) $$w = \text{softmax}(\text{pos\_distance} - \text{neg\_distance})$$

$$\begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} \|z_{0,i} - z_{0,j}\|_2 - \|z_{0,i} - z_{n'_0,j}\|_2 \\ \|z_{1,i} - z_{1,j}\|_2 - \|z_{1,i} - z_{n'_1,j}\|_2 \\ \vdots \\ \|z_{n,i} - z_{n,j}\|_2 - \|z_{n,i} - z_{n'_n,j}\|_2 \end{bmatrix} \right)$$

(5.9) $$w = \text{softmax}(\text{pos\_distance} - \min(\text{m}, \text{neg\_distance})$$

$$\begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} \|z_{0,i} - z_{0,j}\|_2 - \min(\text{m}, \|z_{n,i} - z_{n'_0,j}\|_2 \\ \|z_{1,i} - z_{1,j}\|_2 - \min(\text{m}, \|z_{n,i} - z_{n'_1,j}\|_2 \\ \vdots \\ \|z_{n,i} - z_{n,j}\|_2 - \min(\text{m}, \|z_{n,i} - z_{n'_n,j}\|_2 \end{bmatrix} \right)$$

(5.10)

$$l_n = w_n * \text{max-margin loss} = w_n * \left(\mathbb{1}_{y_i = y_j} \|z_{n,i} - z_{n,j}\|_2^2 + \mathbb{1}_{y_i \neq y_j} \max(0, m - \|z_{n,i} - z_{n',j}\|_2)^2 \right)$$

5.2.5 Triplet Loss

Triplet loss has a similar idea to contrastive loss. In the original formula, we use one positive and one negative sample as shown in Equation 5.11. As before, we minimize the distance between the original anchor and maximize the distance to the negative sample. We adopt this formula to fit the contrastive learning setup and use $z_{n,i}$ and $z_{n,j}$ as the graph n with two augmentations, the positive pair. We randomly select the negative sample $z_{n',j}$ from all other graphs in the batch. We show the final formula in Equation 5.12.

$$l(z_i, z_j, z_k) = \max(0, \|z_i - z_j\|_2^2 - \|z_i - z_k\|_2^2 + m) \tag{5.11}$$

$$l(z_{n,i}, z_{n,j}, z_{n',j}) = \max(0, \|z_{n,i} - z_{n,j}\|_2^2 - \|z_{n,i} - z_{n',j}\|_2^2 + m) \tag{5.12}$$

5.2.6 TS-SS

Cosine and euclidian distance both have disadvantages. While cosine distance does not consider the magnitude of the values and only the angle, euclidian distance is the opposite and ignores the angle and only considers the magnitude. TS-SS [35] consists of two parts, triangle area similarity (TS) and sector area similarity (SS). TS uses three components, angle, euclidian distance, and magnitude. SS uses the euclidian distance and the magnitude difference. In the end, we multiply both values TS and SS. The result combines euclidian and cosine distance and solves their problems. We show the individual formulas below, starting with Equation 5.13. For our evaluations, we use an existing implementation [36].

(5.13)
$$\mathrm{CD}(A, B) = \frac{\sum_{n=1}^{k} A(n) \cdot B(n)}{|A| \cdot |B|}$$

(5.14)
$$\theta' = \arccos(\mathrm{CD}(A, B)) + 10$$

(5.15)
$$\mathrm{ED}(A, B) = \sqrt{\sum_{n=1}^{k} \big(A(n) - B(n)\big)^2}$$

(5.16)
$$\mathrm{MD}(A, B) = \left| \sqrt{\sum_{n=1}^{k} A_n^2} - \sqrt{\sum_{n=1}^{k} B_n^2} \right|$$

(5.17)
$$\mathrm{TS}(A,B) = \frac{|A| \cdot |B| \cdot \sin(\theta')}{2}$$

(5.18)
$$\mathrm{SS}(A,B) = \pi \cdot \big(\mathrm{ED}(A,B) + \mathrm{MD}(A,B)\big)^2 \cdot \left(\frac{\theta'}{360}\right)$$

(5.19)
$$\text{TS-SS}(A,B) = \mathrm{TS}(A,B) \cdot \mathrm{SS}(A,B)$$

We use two different ways to use the TS-SS. First, we keep the max-margin loss formulation from the euclidian loss and replace the euclidian distance with the TS-SS distance. We call this formulation from Equation 5.20 the TS-SS sum. Secondly, we define a ratio loss, similar to the NT-Xent loss, where we build the ratio between the positive sample distance divided by the negative sample distance. We call this formula in Equation 5.21 the TS-SS ratio.

(5.20) $$l_n = \mathbb{1}_{y_i=y_j}\text{TS-SS}(z_{n,i}, z_{n,j})^2 + \mathbb{1}_{y_i\neq y_j}\max\big(0, m - \text{TS-SS}(z_{n,i}, z_{n',j})\big)^2$$

(5.21) $$l_n = \frac{\text{TS-SS}(z_{n,i}, z_{n,j})^2}{\sum_{n'=1,[n'\neq n]}^{N}\text{TS-SS}(z_{n,i}, z_{n',j})^2}$$

5.3 Shrink & Perturb

We define three strategies to use the model's weights in transfer learning and warm-starting. The first method is the default method and does not apply any modification to the weights. The second method applies the shrink & perturb technique from Ash et al. [20] to increase the model's generalizability. Equation 5.22 shows the formula. For each weight, we apply two steps. In the first step, the shrinking phase, all weights are multiplied with the shrinking parameter $\lambda = 0.2$. The second step, the perturbation step, adds gaussian distributed noise with the standard deviation σ=0.01 to the weights.

(5.22) $$\forall w : w = w * \lambda + \mathcal{N}(0, \sigma)$$

In the third method, we use the same shrinking parameter $\lambda = 0.2$. We modify the formula, so the gaussian noise has no static σ, but instead, we use the average weight of the current layer to compute the gaussian noise for all weights in the layer as shown in Equation 5.23. Since different layers have different magnitudes of weights, we propose this method to find a better fit for σ.

(5.23) $$\forall w : w = w * \lambda + \mathcal{N}(0, \max(1\text{e}{-10}, \mathrm{AVG}(L_w)))$$

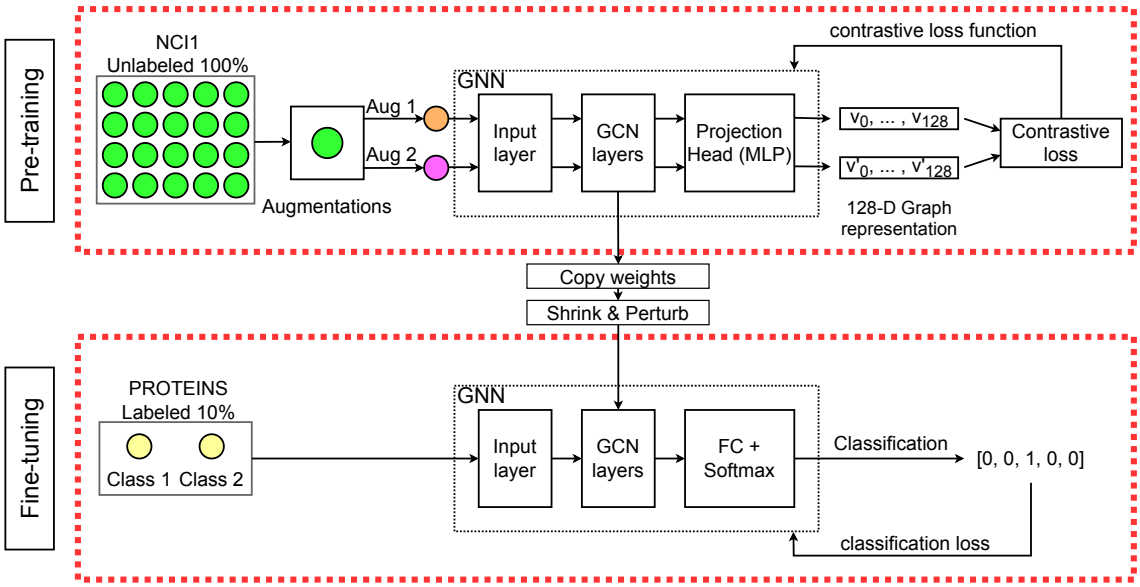

Figure 5.4: The image shows the setup for transfer learning. In this example, we pre-train with the unlabeled NCI1 data and fine-tune with the labeled PROTEINS data. We can not keep the pre-trained input layers, like in our standard setup, because different datasets usually have different input sizes.

5.4 Transfer Learning

We describe the transfer learning setup in Figure 5.4. As previously mentioned, we use different datasets for pre-training and fine-tuning. In the example, we pre-train with the complete unlabeled NCI1 and fine-tune with the labeled PROTEINS data. The goal is to gain additional training through different and potentially bigger datasets. Consequently, as a drawback, we can not use the pre-trained input layer since different datasets have different attribute sizes for the nodes. We add an optional shrink & perturb step between pre-training and fine-tuning.

5.5 Warm-Starting

We describe our setup of warm-starting in Figure 5.4. We add a pre-pre-training step to our default setup. The pre-pre-training step uses a different dataset, similar to transfer learning. In the example, we pre-pre-train with the complete unlabeled NCI1. Afterward, we pre-train with the unlabeled PROTEINS, followed by fine-tuning with the labeled PROTEINS data. We plan to profit from the general idea of transfer learning. At the same time, we still pre-train on the same dataset before the fine-tuning step to reduce the downside of transfer learning. In this setup, we can use a pre-trained input layer in the fine-tuning step.

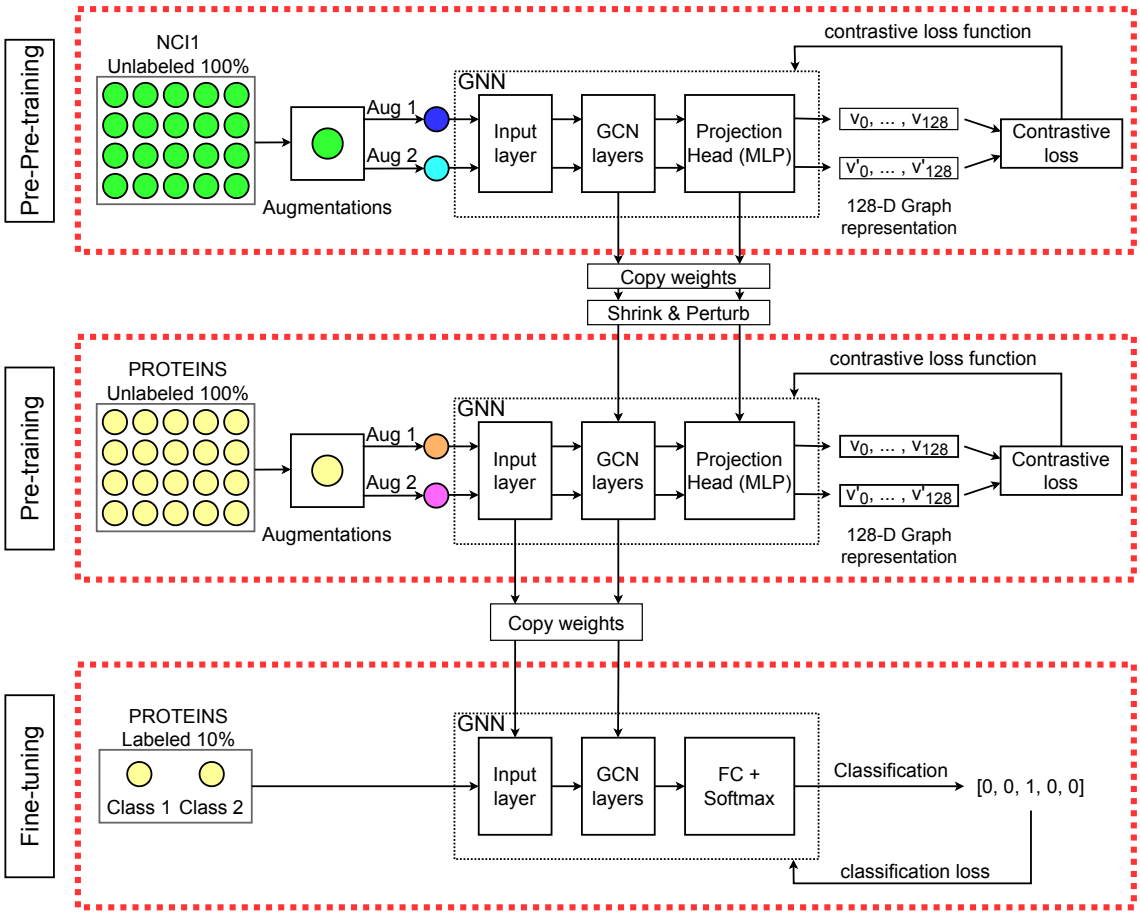

Figure 5.5: The image shows the setup for warm-starting. In this example, we pre-pre-train with the unlabeled NCI1 data. We follow with a pre-training step with the unlabeled PROTEINS data. We fine-tune with the labeled PROTEINS data. Compared to transfer learning, we can keep the input layers between the pre-training and fine-tuning since we use the same dataset.

6 Augmentations

This chapter describes the details of the augmentations we evaluate in this thesis. We construct 23 different augmentations of different types. The three basic operations for nodes are add, remove and mask, while edges have two options, add or remove. Simple augmentations start by randomly editing the graphs. More advanced augmentations focus on using an order or strategy to remove unimportant nodes and edges that carry few crucial details. As motivated in 2.6, the success of contrastive learning relies on the fact that we remove nodes and edges without changing the label of the graph. We can measure the importance of a node by centrality, which ranks the nodes in the graph. Many centralities like degree, eigenvector, pagerank [37], and Hyperlink-Induced Topic Search (HITS) [38] exist. Since it is yet unknown which centrality gives the best results for contrastive learning, and we expect that different datasets have different centralities that will perform the best, we test all previously mentioned. We present an overview of all the augmentations with a short description in Table 6.1.

6.0.1 Augmentation Ratio

The augmentation ratio defines how many nodes or edges get edited. Since the augmentation ratio is an important parameter, we construct most methods to result in the specified ratio. For example, the width-search subgraph method has been slightly modified compared to the typical implementation. As a default value, we use 20%, similar to other research [9], although we later analyze different augmentation ratios because, depending on the dataset, different datasets can tolerate higher augmentation rates and achieve better results.

6.0.2 Node Dropping

Algorithm 6.1 NodeDropping augmentation

Require: G = {V, E}, aug_ratio=k

1: V_{drop} = select_nodes_to_drop(V, |V|*k)
2: $V_{\text{nodedrop}} = G \setminus V_{\text{drop}}$
3: $E_{\text{nodedrop}} = \{e | e \in E \text{ and } e[0] \in V_{\text{nodedrop}} \text{ and } e[1] \in V_{\text{nodedrop}}\}$
4: $G_{\text{nodedrop}} = \{V_{\text{nodedrop}}, E_{\text{nodedrop}}\}$
5: **return** G_{nodedrop}

Augmentations 1-4 define different variations of node dropping. 20% node dropping means that we drop |V|*0.2 nodes. Even though only nodes are directly affected by this method, we also drop edges because all the edges which connect a dropped node are now redundant.

Nr.	Data Augmentation	Type	Description
1	dropN	Nodes + Edges	Drop random nodes
2	dropN_high_degree	Nodes + Edges	Drop nodes with the lowest degree
3	WdropN_high_degree	Nodes + Edges	Drop nodes with high degree with higher probability
4	WdropN_low_degree	Nodes + Edges	Drop nodes with low degree with higher probability
5	addN	Nodes + Edges	Add nodes randomly and connect them to the graph
6	Mask_Mean	Masking	Mask nodes attributes with the mean attribute of all nodes
7	Mask_0	Masking	Mask nodes with 0
8	Switch_Attributes	Masking	Switch the node attributes of two nodes
9	permE	Edges	Randomly remove existing edges an create new ones
10	addE	Edges	Randomly add edges
11	dropE	Edges	Randomly remove edges
12	subgraph	Nodes + Edges	Subgraph from a random start node
13	subgraph_depth	Nodes + Edges	Subgraph with depth-first-search
14	subgraph_width	Nodes + Edges	Subgraph with width-first-search
15	subgraph_high_degree	Nodes + Edges	Subgraph with highest degree nodes first
16	subgraph_low_degree	Nodes + Edges	Subgraph with lowest degree nodes first
17	subgraph_eigenvector	Nodes + Edges	Subgraph with highest eigenvector value nodes first
18	subgraph_pagerank	Nodes + Edges	Subgraph with highest pagerank value nodes first
19	subgraph_HITS	Nodes + Edges	Subgraph with highest HITS value nodes first
20	subgraph_merge_nodes	Nodes + Edges	Merge Nodes and keep their edges
21	subgraph_steiner	Nodes + Edges	Steiner Tree
22	subgraph_steiner_low_degree	Nodes + Edges	Steiner Tree of the lowest degree nodes
23	subgraph_steiner_weighted	Nodes + Edges	Steiner Tree with weights depending on the degree

Table 6.1: Overview of all tested augmentations

The base algorithm selects the nodes to drop, removes them, and removes the redundant edges. Algorithm 6.1 shows the base algorithm. We test four different settings for node dropping. They differ in the select_nodes_to_drop method. The first method, augmentation 1, is the basic function, where select_nodes_to_drop selects the nodes randomly. Augmentation 2 sorts the nodes by degree and drops the nodes with the highest degree in the graph. Augmentations 3+4 are a weighted function from You et al. [9]. Both methods assign weights to the nodes depending on the degree. The weight defines the probability of dropping a node. We show the formulas for computing the weights in (6.1). A higher weight results in a higher chance of the node getting dropped. Nodes are then randomly selected given these probabilities. In contrast, augmentation 2 is a hard decision boundary.

(6.1) Augmentation 3: WdropN_high_degree: $P(\text{node}) = \frac{\text{degree(node)}^2}{\sum_{n \in N} \text{degree(n)}^2}$

(6.2) Augmentation 4: WdropN_low_degree: $P(\text{node}) = \frac{\text{degree(node)}^{-2}}{\sum_{n \in N} \text{degree(n)}^{-2}}$

6.0.3 Node Adding

Instead of removing nodes, we also test adding additional nodes in Algorithm 6.2. 20% node adding means that |V|*0.2 nodes are added. Compared to dropping nodes, this creates some problems. Mainly which edges and which attributes the new node should have. Ideally, we add meaningful nodes in the context of the dataset, but this would require in-depth domain knowledge. We stick to a simple version, in which we randomly select a node, copy the node and its attributes and add it to another random node in the graph with one edge.

Algorithm 6.2 NodeAdd augmentation

Require: G = {V, E}, aug_ratio=k
for i in range(|V|*k) **do**
    v = random_node(V)
    v_{copy} = copy_node(v)
    u = random_node(V)

    $V = V \cup v_{copy}$
    $E = E \cup (v_{copy}, u) \cup (u, v_{copy})$
end for
return G

6.0.4 Attribute Masking

Attribute Masking does not change the graph's topology but only masks the node attributes. 20% node masking means that |V|*0.2 nodes are selected, and all attributes in this node are masked. Similar to node adding, we would like to mask the node attributes with values that fit the data. Again, this is only possible with in-depth domain knowledge, so we use two approximations instead. Algorithm 6.3 shows the algorithm for masking. The first version is augmentation 6, which sets the

attributes of the selected nodes to the mean value of all node attributes in the graph. This method is used by previous papers like [9] and achieved mixed results. However, many datasets use one-hot vectors as a node attribute, while the average is usually a continuous value. These continuous values are an outlier to the model and not expected, which contradicts our assumption to mask with values that fit the data and look more natural. This observation leads us to the assumption that masking with the average might not be optimal, and we introduce the alternative, augmentation 7, where all node attributes get masked with 0. In Algorithm 6.3, we set all values in the mask to 0 instead of the average.

Algorithm 6.3 NodeMasking augmentation

Require: G = {V, E}, aug_ratio=k
1: V_{mask} = random_nodes(V, |V|*k)
2: mask = average(V['attribute'])
3: V_{mask} ['attribute'] = mask
4: **return** G

Additionally, Algorithm 6.4 shows the switching of node attributes in augmentation 8. 20% Attribute switch means that (V*0.2)/2 node pairs are selected. The node attributes of both nodes get switched, which results in some form of masking node attributes. One advantage is that we use the attribute from an actual node, not a synthetical average. A downside of this approach is that we still can not guarantee to get realistic results. In a molecule graph, the node can represent an atom, defined by the attributes. We solve the problem that the average attribute is most likely not a feasible atom. We did not solve the problem that the atom, which changed its position in the topology, now changes the molecule. Even though our graph consists of feasible nodes, we can still end up with unfeasible graphs.

Algorithm 6.4 NodeSwitch augmentation

Require: G = {V, E}, aug_ratio=k
1: **for** i in range((|V|*k)/2) **do**
2: v = random_node(V)
3: u = random_node(V)
4: attribute_v = v['attribute']
5: attribute_u = u['attribute']
6: v['attribute'] = attribute_u
7: u['attribute'] = attribute_v
8: **end for**
9: **return** G

6.0.5 Edge Perturbation

In augmentation 9, 20% Edge Perturbation means that N existing edges get removed, and additionally, E*0.2 edges are created randomly between the nodes. It is a combination of augmentation 10 and 11, as marked in Algorithm 6.5.

Algorithm 6.5 Edge Perturbation augmentation

Require: G = {V, E}, aug_ratio=k

```
1: edges = random_edges(E, |E|*k)
2: E = E \ edges
3:
4: for i in range(|E|*k) do
5:     v = random_node(V)
6:     u = random_node(V)
7:     E = E ∪ (u, v) ∪ (v, u)
8: end for
9: return G
```

Remove edges [Aug. 10]

Add edges [Aug. 11]

Edge Perturbation [Aug. 9]

6.0.6 Subgraph

20% Subgraph augmentation means that a subgraph with $|V_{\text{subgraph}}|$ = (|V| * (1-0.2)) nodes is created. Similar to node dropping, 20% of the nodes get removed, but the remaining nodes are now not random but, instead, are an ideally connected subgraph. We select the start node randomly. Neighboring nodes get added until the subgraph is big enough. The select_neigbour_node(V_{neigbors}) in line 11 is the crucial difference between the multiple subgraph methods.

In the end, the resulting graph consists of all the selected nodes V_{subgraph} and all the edges between the nodes in the subgraph. Algorithm 6.6 shows augmentation 12, the basic subgraph algorithm in pseudocode. The select_neigbour_node(V_{neigbors}) method randomly selects one of all the neighbors V_{subgraph} from as the next node.

Algorithm 6.6 Subgraph augmentation

Require: G = {V, E}, aug_ratio=k

```
1: V_subgraph = ∅
2: v = random_node(V)
3: V_subgraph = v
4:
5: while |V_subgraph| < |V| * (1-k) do
6:     V_neigbors = (V_neigbors ∪ get_neigbors(v) ) \V_subgraph
7:     if V_neigbors = ∅ then
8:         v = random_node(V \ V_subgraph)
9:     else
10:        v = select_neigbour_node(V_neigbors)
11:    end if
12:    V_subgraph = V_subgraph ∪ v
13: end while
14: E_subgraph = {e|e ∈ E and e[0] ∈ V_subgraph and e[1] ∈ V_subgraph}
15: G_subgraph = {V_subgraph, E_subgraph}
16: return G_subgraph
```

Depth-First-Search

Algorithm 6.7 shows the selection of the depth-first-search subgraph. A random start node is selected, and depth-first-search is used from this starting point until the subgraph has the correct size. Depth-first-search traverses the graph and explores all nodes along the selected branch until they are all added to the subgraph.

Algorithm 6.7 Subgraph-Depth-Search augmentation

Require: G = {V, E}, aug_ratio=k
1: $V_{\text{subgraph}} = \emptyset$
2: v = random_node(V)
3: V_{subgraph} = v
4:
5: **while** $|V_{\text{subgraph}}|$ < |V| * (1-k) **do**
6: V_{neigbors} = get_neigbors(v) $\setminus V_{\text{subgraph}}$
7: **if** $V_{\text{neigbors}} = \emptyset$ **then**
8: $V_{\text{all\_neigbors}} = \left(\cup_{v \in V_{\text{subgraph}}} \text{get\_neigbors(v)}\right) \setminus V_{\text{subgraph}}$
9: **if** $V_{\text{all\_neigbors}} = \emptyset$ **then**
10: v = random_node($V \setminus V_{\text{subgraph}}$)
11: **else**
12: v = random_node($V_{\text{all\_neigbors}}$)
13: **end if**
14: **else**
15: v = random_node(V_{neigbors})
16: **end if**
17: $V_{\text{subgraph}} = V_{\text{subgraph}} \cup$ v
18: **end while**
19: $E_{\text{subgraph}} = \{e | e \in E \text{ and } e[0] \in V_{\text{subgraph}} \text{ and } e[1] \in V_{\text{subgraph}}\}$
20: $G_{\text{subgraph}} = \{V_{\text{subgraph}}, E_{\text{subgraph}}\}$
21: **return** G_{subgraph}

Width-First-Search

Algorithm 6.8 uses width-first-search to build the subgraph. A random start node is selected, and width-first-search is used from this starting point until the subgraph has the correct size. In contrast to depth-first-search, we do not explore a single branch but all branches simultaneously. For our case, this means all neighbors of V_{subgraph} get added during every step.

Degree

Augmentations 15 and 16 decide which node gets added to the subgraph with the degree. Instead of randomly selecting neighbor nodes in the select_neigbour_node(V_{neigbors}) method, augmentation 15 sorts and selects the neighbor of V_{subgraph} which has the highest degree. Accordingly, augmentation 16 selects the neighbor with the lowest degree.

Algorithm 6.8 Subgraph-Width-Search augmentation

Require: G = {V, E}, aug_ratio=k

V_{subgraph} = ∅
v = random_node(V)
V_{subgraph} = v

while $|V_{\text{subgraph}}|$ < |V| * (1-k) **do**
    $V_{\text{neigbors}} = \left(\cup_{v \in V_{\text{subgraph}}} \text{get\_neigbors(v)}\right) \setminus V_{\text{subgraph}}$
    if $V_{\text{neigbors}} = \emptyset$ **then**
        v = random_node(V \ V_{subgraph})
        $V_{\text{subgraph}} = V_{\text{subgraph}} \cup$ v
    else
        for neig in neigbors **do**
            if $|V_{\text{subgraph}}|$ < |V| * (1-k) **then**
                $V_{\text{subgraph}} = V_{\text{subgraph}} \cup$ neig
            end if
        end for
    end if
end while
$E_{\text{subgraph}} = \{e | e \in E \text{ and } e[0] \in V_{\text{subgraph}} \text{ and } e[1] \in V_{\text{subgraph}}\}$
$G_{\text{subgraph}} = \{V_{\text{subgraph}}, E_{\text{subgraph}}\}$
return G_{subgraph}

Eigenvector

The select_neigbour_node(V_{neigbors}) method sorts the neighbors and selects the neighbor node with the highest eigenvector value. The eigenvector centrality gets calculated by solving the equation $Ax = \lambda x$, where A is the adjacency matrix of the graph with eigenvalue λ. For node i, the i-th element of x is the eigenvector centrality.

Pagerank

The select_neigbour_node(V_{neigbors}) method sorts the neighbors and selects the neighbor node with the highest Pagerank value. The Pagerank algorithm [37] is a way to measure the importance of websites and was introduced by Google. Nodes with more (incoming) edges receive a higher Pagerank. The assumption is that a node with lots of incoming edges is important. For undirected graphs, which we use, the Pagerank is close to the degree distribution. We show the equation to calculate the Pagerank for a graph in (6.3). The total number of nodes is N. We use a damping parameter d=0.85. PR(n) describes the Pagerank of node n. M(n) are all nodes with an edge to node n. L(n) is the number of outgoing edges from n. We compute the Pagerank iteratively. We start with a Pagerank of 1/N for all nodes and update the Pagerank for all nodes with the formula in (6.3). We repeat this calculation process until convergence.

$$\text{PR}(n_i) = \frac{1-d}{\text{N}} + d \sum_{n_j \in \text{M}(n_i)} \frac{\text{PR}(n_j)}{\text{L}(n_j)} \tag{6.3}$$

Hyperlink-Induced Topic Search (HITS)

The select_neigbour_node(V_{neigbors}) method sorts the neighbors and selects the neighbor node with the highest HITS value. HITS is an algorithm to rate websites with the concept of hubs and authorities [38]. Like Pagerank, it is an iterative algorithm, but it calculates two values, the authority score and the hub score. We start with an authority and hub value of 1 for all nodes. The authority value gets calculated by summing all hub values of the incoming edges, see (6.4). The hub value sums the authority values of all target nodes from the outgoing edges, see (6.5). Afterward, both values are normalized. We repeat this calculation process until convergence. Since we use directed graphs, both values have the same result, and for our calculations, we only use the hub value.

$$\text{Auth}(n_i) = \sum_{n_j \in \text{M}(n_i)} \text{Hub}(p_i) \tag{6.4}$$

$$\text{Hub}(n_i) = \sum_{n_j \in \text{M}(n_i)} \text{Auth}(p_i) \tag{6.5}$$

6.0.7 Merge Nodes

The subgraph_merge_nodes method is similar to the subgraph method, but instead of selecting neighbors and adding them to the subgraph, we select a random node, one of the neighbors, and then merge these two neighbors. Merging in this context means keeping the first node and adding edges to all neighbors of the second node. Afterward, we remove the second node and all edges from it as seen in Algorithm 6.9. The result is a mixture of node dropping and subgraph. Compared to node dropping, we keep the nodes connected when we remove a node. In the subgraph method, we keep the complete subgraph, while merging the nodes allows us to remove a node from the subgraph without splitting up the graph. The merge nodes method guarantees that if we have two connected nodes and remove some of the nodes connecting them, they remain connected.

6.0.8 Steiner Trees

Steiner trees are a problem similar to the minimum spanning tree (MST). Every edge e ∈ E has a corresponding cost C(e) ∈ N. A MST connects all the vertices V in G (without cycles) with minimal total cost C(G)

The Steiner tree has instead only a subset T ⊆ V, the terminal nodes, of the vertices V given and searches for the spanning tree that connects this subset with minimal cost. The Steiner tree problem is proven to be NP-hard [39]. Thus an approximation is needed to calculate the trees efficiently.

Algorithm 6.9 Subgraph_merge augmentation

Require: G = {V, E}, aug_ratio=k

for i in range(|V|*k) **do**
    v = random_node(V)
    N = select_neigbour_node(V_{neigbors})
    u = random_node(N)

    V = V \ u
    for edge in {e|e ∈ E and e[0] = u} **do**:
        E = E \ edge
        E = E ∪ (v, edge[1])
    end for
    for edge in {e|e ∈ E and e[1] = u} **do**:
        E = E \ edge
        E = E ∪ (edge[0], v)
    end for
end for
return G

First, the approximation solves the shortest path problem for all pairs of vertices in the graph, which results in the metric closure of the graph. The algorithm starts with one of the terminal nodes, which we add to the result set R. The terminal node with the shortest distance to one of the nodes in R will be selected, and we will add all the nodes on this shortest path to R. We repeat the previous step until no more terminal nodes are left. After all terminal nodes are in the result set R, we can calculate the MST on the resulting graph. The result is the Steiner tree, a subgraph of G.

We need to add some exceptions due to the limitations of the approximation. The graph must be connected in order for the steiner approximation to work. If the graph is not connected, we use the subgraph (12_subgraph) implementation instead. For the larger social graph datasets like COLLAB and REDDIT-BINARY, the runtime is even for the approximation very high. Therefore, we limit the three steiner methods to the five datasets PROTEINS, NCI1, FRANKENSTEIN, COIL-DEL, and COLORS-3.

While we previously made sure to apply the same level of augmentation to the graphs, we can not guarantee this for steiner trees. We do not know how big the steiner tree for a specific subset T will be, while the other subgraph methods construct subgraphs with an exact number of nodes. The subgraph augmentation with 20% augmentation ratio builds a subgraph with 80% of the nodes, while 20% of the nodes get removed. If we build a steiner tree with 80% of the nodes in the terminal set, the subgraph is likely considerably larger. In the best case, all nodes $v \in T$ are already a connected subgraph, and no additional nodes are necessary. Since the shortest path connects the nodes in T, the resulting augmentation ratio is, on average, considerably lower than we wanted. In our example, instead of 20%, the augmentation ratio would be closer to 5-10%. In theory, We could pick a smaller size for T, such that the steiner tree has the correct augmentation ratio. However, we only know the exact size of the steiner tree after its construction. So to find a steiner tree with a specific size, we need to construct lots of steiner trees, which is not reasonable. Therefore, we accept the limitation of a higher actual augmentation ratio.

We use three different variants of steiner trees. One thing they have in common is that we select the subset T ⊆ V with a size of T = |V| * aug_ratio nodes for our Evaluations. For the PROTEINS dataset, an augmentation ratio of 20% would result in the terminal set of size 0.2*|V|. After connecting these terminal nodes, on average, we end up with graphs that keep around 40% of the nodes. The real augmentation for the 20% input is 60%. Other datasets can achieve different results. Consequently, comparing the steiner tree performance to other augmentation ratios is hard. We only compare the steiner tree methods to the best augmentations of the dataset, independent of the augmentation ratio.

We show a general example for the steiner tree approximation in Figure 6.1. The nodes, terminal nodes, and edge weights in this example are random. They depend on the augmentation we use.

Algorithm 6.10 shows the first general version. In augmentation 21, the subset T ⊆ V is created by randomly selecting T from all nodes. The weight for all edges is the same, so only the distance is used.

Augmentation 22 does not use random nodes. We create the subset T ⊆ V by selecting the |T| nodes with the lowest degree from all nodes. This method is similar to augmentation 16, where we build the subgraph by selecting the lowest degrees. We only consider the neighbors of V_{subgraph} for the subgraph. This augmentation selects the neighbors with the lowest degree from the whole graph and afterward connects these nodes. Once again, we do not use specific weights and only consider the distance for the shortest paths.

In augmentation 23, we select the same subset T as in augmentation 22, with the lowest degree nodes. Additionally, we now introduce weights for the edges. As a cost function, we add the degree of both connected nodes, C(e) = degree(e[0]) + degree(e[1]). This way, we connect the low-degree nodes and minimize the degree of the nodes connecting them.

Algorithm 6.10 Subgraph_steiner augmentation

Require: G = {V, E}, aug_ratio=k

```
1:
2: if ! G.is_connected then
3:     return subgraph(V, k)
4: end if
5:
6: T = select_terminal_nodes(V, |V|*k)
7: G_steiner = steiner_approximation(G, T)
8: return G_steiner
```

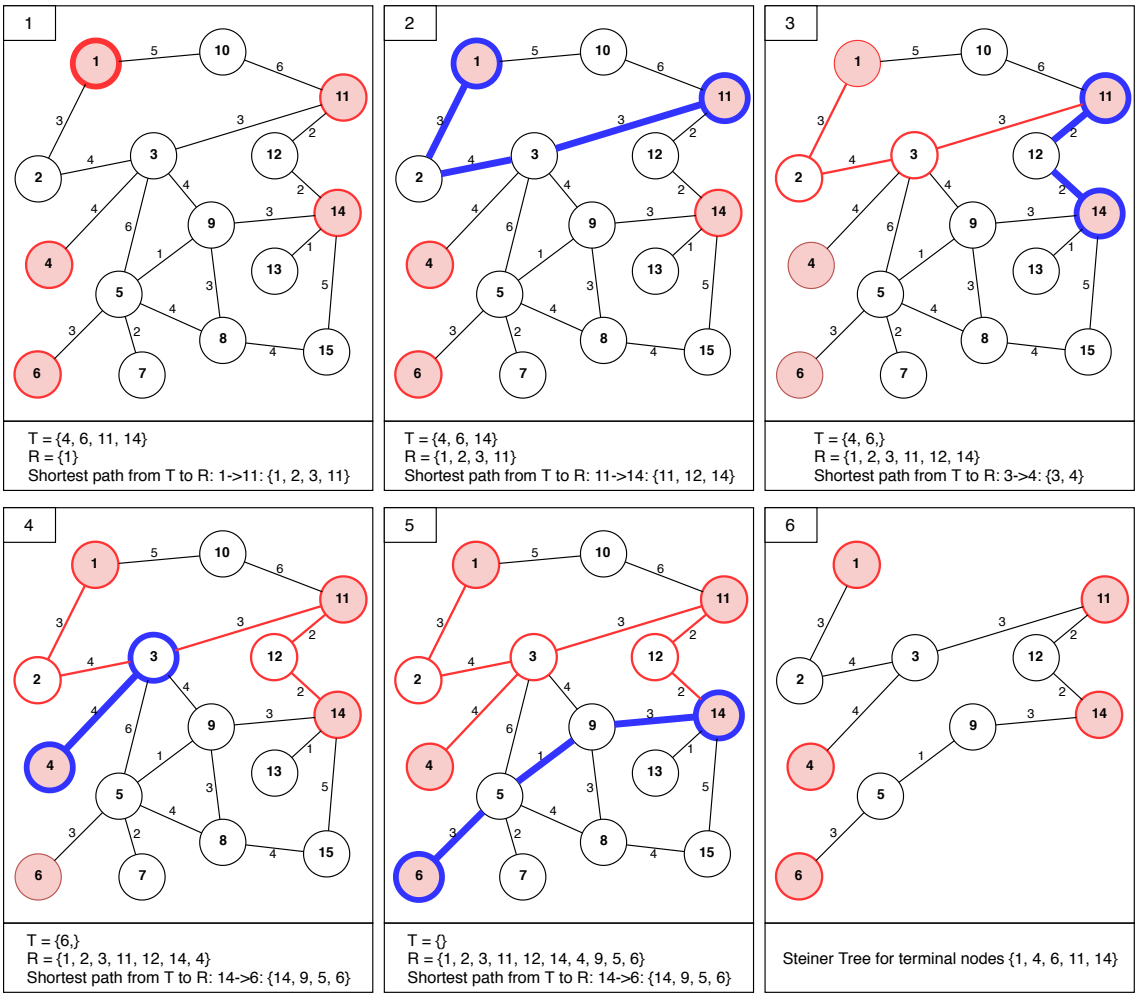

Figure 6.1: Steiner tree calculation for the red highlighted random set of terminal nodes T = 1, 4, 6, 11, 14. The blue nodes and the path are the shortest path from the current result, R, to a terminal in T. We add the blue path to R and repeat this step until all terminal nodes are in R. In the last step, we calculate an MST of R.

7 Evaluation

During our evaluations, we test multiple settings and approaches. Firstly, we briefly describe the datasets and introduce our setup for the following evaluations. In Section 7.3, we start with the first evaluation, the fundamental performance analysis of the augmentations defined in Chapter 6. We first evaluate the performance of single augmentations with the original graphs, check the influence of the augmentation ratio and finally evaluate the combination of different augmentation ratios with the best-performing augmentations. We also test different loss functions and the influence of the number of positive and negative samples for contrastive learning. We then look into two methods two improve our setup, transfer learning and warm-starting. Additionally, we add the shrink and perturb step introduced by Ash et al. [20] to both approaches and test if their approach is transferable to our setting. To improve the previous results from the augmentations, we propose different strategies of selecting augmentations for each graph individually in Section 7.9 instead of one augmentation for all graphs. We finish with Section 7.10 and the advanced approach of not using predefined augmentations but instead using a generator that creates individual augmentations for each graph by removing all unimportant nodes and edges.

7.1 Datasets

We focus on 12 graph datasets from the TUDataset collection [40], [41]. They are from different domains, Bioinformatics, Small molecules, Computer Vision, and Social networks, which differ in size, connections, and classes. See Table 7.1 for details.

Bioinformatics graphs like PROTEINS or DD describe macromolecules. Examples are protein structures, where the task is to predict if the protein is an enzyme. Nodes represent structure elements with their type and physical and chemical information. Edges connect two nodes that are neighbors along amino acid sequences.

Small molecules graphs like NCI1, MUTAG, and FRANKENSTEIN (FRANKEN.) describe molecules with nodes as atoms and edges as chemical bonds. The labels of the molecules represent different properties like toxicity or biological activity.

COIL-DEL is a graph dataset from the computer vision domain. Originally 100 objects are photographed from different angles. Corner features are extracted from the images, and a Delaunay triangulation is applied. The Delaunay triangulations are transformed into a graph where each node has a coordinate, and the edges represent the lines.

IMDB-BINARY (IMDB-B), COLLAB, github_stargazers (GITHUB), REDDIT-BINARY (RDT-B), and REDDIT-MULTI-5k (RDT-5K) are all graph datasets from social networks. REDDIT-BINARY and REDDIT-MULTI-5k represent nodes as users, and edges indicate that one user responded to another. The task is to distinguish the type of subreddit and where a thread was posted. COLLAB,

Datase	Graphs	Classes	Nodes	Edges	Node Att.	Type	Source
PROTEINS	1113	2	39.06	72.82	1	Bioinformatics	[42] [43]
DD	1178	2	284.32	715.66	0	Bioinformatics	[43] [44]
NCI1	4110	2	29.87	32.30	9	Small molecules	[45] [44]
MUTAG	188	2	17.93	19.79	0	Small molecules	[46]
FRANKENSTEIN	4337	2	16.90	17.88	780	Small molecules	[47]
COIL-DEL	3900	100	21.54	54.24	2	Computer vision	[48] [49]
IMDB-BINARY	1000	2	19.77	96.53	0	Social networks	[50]
COLLAB	5000	3	74.49	2457.78	0	Social networks	[50]
GITHUB	12725	2	113.79	234.64	0	Social networks	[50]
REDDIT-BINARY	2000	2	429.63	497.75	0	Social networks	[50]
REDDIT-MULTI-5K	4999	5	508.52	594.87	0	Social networks	[50]
COLORS-3	10500	11	61.31	91.03	4	Synthetic	[51]

Table 7.1: Datasets used from the TUDataset Collection [40], [41]

IMDB-BINARY, and GITHUB are three datasets with the same task but from different backgrounds. The COLLAB datasets are from scientific collaboration networks. Each graph represents the network of one researcher. The task is to predict the research field. In the IMDB-Binary dataset, instead of scientists, each graph represents an actor's network. We predict the genre of the actor. The GITHUB dataset uses the network of GitHub users and predicts if they starred in machine learning or web development repositories.

COLORS-3 is a synthetic dataset that was generated to show specific strengths and weaknesses of attention in GNNs. Random graphs are generated and assigned red, green, or blue. The task is to count the number of nodes with a specific color (green). The original idea is to study the influence of the model's initialization.

7.2 Setup

If not stated otherwise, our evaluations run with the model and setup described in Section 5.1 We first run a pre-training step using unsupervised contrastive learning on the entire dataset. Followed by that, we run graph classification in a supervised fine-tuning step with only 10% of the (labeled) data. The results from the fine-tuning follow a 10-fold cross-validation. To compare our results to different papers, we use both the mean-accuracy and the max-accuracy as described in Section 5.1. We split our data into 90% training data and 10% test data. We count 10% labeled data for fine-tuning as 10% of the whole data. 10% of the whole data would equal 11.11% of the training data, which means 11.11% would be a more accurate description. Since the exact label percentage is irrelevant and the papers we reference all adopt the same split and call this setup 10% labeled data, we decided to keep this notion for better comparability.

The augmentation experiments are repeated and averaged five times (DD, NCI1, COLLAB, GITHUB, RDT-B, RDT-5K, COLORS-3) or 20 times (PROTEINS, MUTAG, FRANKENSTEIN, COIL-DEL, IMDB-B), depending on the size of the dataset. The accuracy in our result is the final classification

	Dataset	PROTEINS	DD	NCI1	MUTAG	FRANKEN.	COIL-DEL
mean-acc	10% baseline	69.82±5.13	74.30±3.89	73.85±2.89	78.83±8.41	60.85±2.91	21.69±2.09
	best aug	72.30±3.77	75.97±3.77	74.22±2.76	80.80±8.09	63.42±2.12	45.54±2.06
	Full data	75.99±2.91	79.09±3.14	83.03±1.64	88.29±7.92	66.86±2.09	74.66±2.13
max-acc	10% baseline	73.42±3.93	76.75±3.20	75.70±2.71	85.86±6.53	63.84±1.90	22.71±2.06
	best aug	74.70±3.03	78.13±3.50	75.85±2.51	85.49±6.61	65.90±1.50	46.87±1.89
	Full data	78.64±3.93	81.85±3.20	84.77±2.71	94.13±6.53	69.19±1.90	76.84±2.06

	Dataset	IMDB-B	COLLAB	GITHUB	RDT-B	RDT-5K	COLORS-3
mean-acc	10% baseline	66.90±4.80	73.45±1.63	61.01±1.78	86.72±2.11	51.38±2.06	51.78±2.10
	best aug	68.52±5.55	75.81±1.79	66.22±1.50	88.70±2.05	52.81±1.95	83.68±2.61
	Full data	73.68±4.95	82.39±1.55	69.09±1.37	92.28±1.60	56.76±1.55	99.75±0.24
max-acc	10% baseline	69.30±4.56	75.14±1.25	62.57±1.39	89.06±1.54	53.08±1.60	53.34±1.89
	best aug	71.13±5.29	77.45±1.28	67.12±1.20	90.73±1.29	54.20±2.12	85.90±1.77
	Full data	78.36±3.93	84.10±3.20	70.40±2.71	94.00±1.90	58.45±6.53	99.97±2.06

Table 7.2: Baseline classification accuracy of the datasets with the variance. 10% baseline uses only fine-tuning with 10% of the labeled data. Full data uses 100% of the data with labels during fine-tuning. The best augmentation is the best single augmentation with 20% augmentation ratio.

accuracy of the fine-tuning. We state our baseline without pre-training in Table 7.2. The following results always show the change in percentage points in the classification accuracy we get when we add the specific pre-training step before fine-tuning the model.

Contrastive learning uses two augmented views of the data. Due to computation limitations, we can only test a limited number of augmentation combinations. We use a setup in which we, if not explicitly stated otherwise, pick the first augmentation as the original graph without augmentation, the so-called Identical augmentation. The second augmentation is one of our 23 defined augmentations from Chapter 6. For better readability, we do not use the full name of the augmentation. Instead, we highlight specific augmentations by adding the number of the augmentation in round brackets behind it. If we write that we build the low-degree subgraph (16), we reference augmentation 16_subgraph_lowest_degree. We list all combinations of augmentation names and numbers in Table 6.1. If not stated otherwise, we use the NT-Xent loss with one positive and 127 negatives.

All evaluations run on four NVIDIA A100-SXM4-40GB (only one GPU per evaluation) and two AMD EPYC 7763 processors with 64 cores. We run CUDA 11.3 with python 3.8, Pytorch 1.7.1, and torch-geometric 1.6.0.

7.3 Augmentations

In the first step, we test different augmentations and their performance on different datasets. We state the baselines of the datasets in Table 7.2. 10% baseline is the classification accuracy of using no pre-training and only the fine-tuning step with 10% of the labeled data. As a reference, full data is the accuracy when we use 100% of the data with labels during fine-tuning. The best augmentation is the best single augmentation with 20% augmentation ratio that we combine with the original graph. The results are only for one single combination that we test. Later evaluations show that

higher augmentation ratios and the combination of augmentations further increase the result. Even with this single test, we show that pre-training consistently improves our results throughout all the datasets. Some datasets like PROTEINS, FRANKENSTEIN, GITHUB, RDT-B, and COLORS-3 close the gap to the full data results by close to 50%. Other datasets like NCI1 and COLLAB still increase the results but have a large gap compared to the full data results.

We show the detailed results from the best augmentation evaluation in Figure 7.1. We tested the original graph (Identical) combined with the selected augmentation during our first evaluations. The reported values are the percentage point increase or decrease in classification accuracy we get by adding pre-training with this augmentation compared to the baseline with no pre-training from Table 7.2. We show improved performance in red and decreased performance in blue. The top plot shows the results when we use the mean-accuracy. The bottom plot shows the results for the max-accuracy. The y-axis specifies the dataset, and the x-axis the augmentation we use. All augmentations use 20% augmentation ratio.

Eventually, we will combine two augmentations and evaluate their performance in the next chapter. However, since for our 23 augmentations, there already exist 529 possible combinations, it is not feasible to compute all possibilities for multiple datasets. Therefore, we use this first evaluation to filter out the most promising augmentations.

The most important result is that an augmentation that improves a dataset does not necessarily perform well on other datasets. We do, however, see strong similarities across datasets. We make multiple interesting key observations. Firstly, using the Identical augmentation for both augmentations, i.e., using only original graphs, usually lowers the model's accuracy. As previous research by You et al. [9] has suggested, contrastive learning with only the original graphs is generally not helpful. The positive loss is 0, which results in different graphs being pushed further away in the embedding space.

We also observe that some augmentations lower the model's accuracy even further than using only original graphs. It is crucial to select good augmentations, while bad augmentations can significantly lower the model's accuracy. We see the general effects in both the mean-accuracy and the max-accuracy evaluations. They only differ in the magnitude of improvement since the max-accuracy method profits more from outliers. This means that the improvement is bigger for the mean-accuracy because the baseline is not as strongly affected by outliers and is lower. As a result, it is easier to improve the mean-accuracy.

Across all datasets adding nodes, attribute mean-masking, switching node attributes, edge perturbation, adding edges, removing edges, and the three subgraph metrics eigenvector, Pagerank, and HITS do not perform well (5, 6, 8, 9, 10, 11, 17, 18, 19). For some datasets, these augmentations do increase the performance, but we have other similar augmentations that perform at least the same or considerably better and make the previous methods redundant.

Bioinformatic datasets like PROTEINS and DD perform best with node dropping and subgraph augmentations (1, 3, 12, 13, 16). PROTEINS additionally improves with attribute masking and merging nodes (7, 20). Keeping low-degree nodes and removing high-degree nodes is a good strategy. The random node dropping and subgraph methods (1, 12) are especially good with DD.

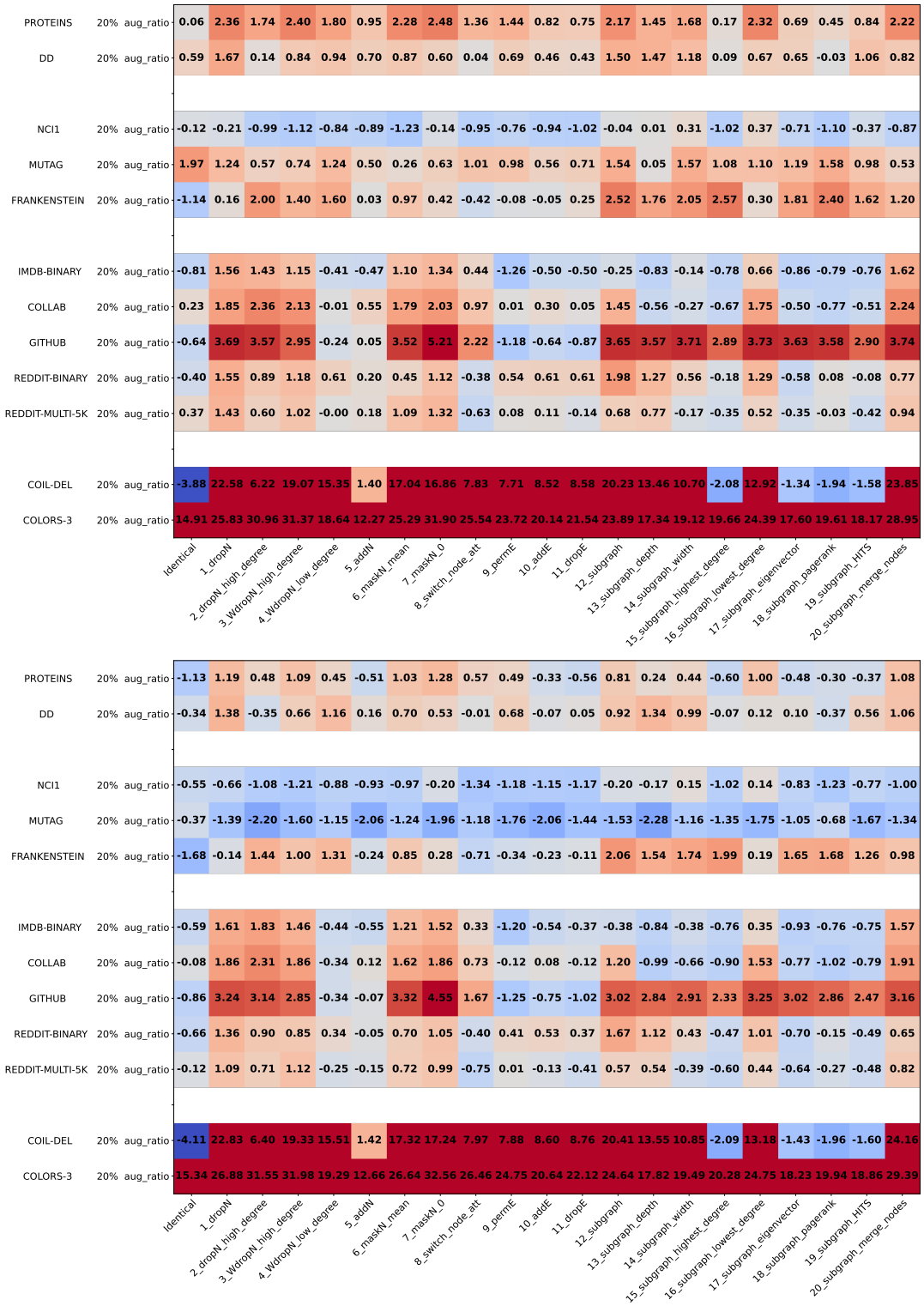

Figure 7.1: We evaluate the combination of the original graph (Identical) with all augmentations. We report the change in classification accuracy. The top plot shows the results when we use the mean-accuracy. The bottom plot shows the results for the max-accuracy. The y-axis specifies the dataset, and the x-axis the augmentation.

Small molecules datasets like NCI1, MUTAG, and FRANKENSTEIN are more difficult to improve. Especially in the max-accuracy setting, many augmentations result in decreased accuracy. The width-search subgraph (14) is generally good across all datasets in this category. MUTAG and FRANKENSTEIN also benefit from the random, high-degree, and Pagerank subgraph (12, 15, 18).

Social network datasets like IMDB-B, COLLAB, GITHUB, RDT-B, and RDT-5K perform best with node dropping and attribute masking (2, 3, 4, 7). Only random, low-degree, and merging nodes (12, 16, 20) perform well from the subgraph methods. The other subgraph methods are considerably worse. GITHUB is an outlier in this behavior. All the subgraph methods perform well.

COIL-DEL performs best with random node dropping and merging nodes (1, 20). This dataset profits a lot from pre-training. Big outliers are the four subgraph methods that keep high-degree nodes (15, 17, 18, 19) and adding nodes (5). They decrease the accuracy by two percentage points, while many other augmentations increase it by over 20 percentage points.

The COLOR-3 dataset also profits a lot from pre-training. We have to consider that this is a synthetic dataset. It is hard to judge how valuable the information from this dataset is. Node dropping, attribute masking, and merging nodes is best (2, 3, 7, 20).

COIL-DEL and COLORS-3 show a very strong increase in accuracy with pre-training compared to the other datasets. To show that the improvement is actually due to the pre-training itself and not only more training, we increase the episodes for fine-tuning and compare the results. Instead of 100, we now use 200 episodes of fine-tuning to calculate the baseline values. For the mean-accuracy, COIL-DEL increases from 21.69% to 21.72%, COLORS-3 increases from 51.78% to 52.22%. In the max-accuracy setting, COIL-DEL increases from 22.71% to 23.58%, COLORS-3 increases from 53.34% to 53.82%. Increased fine-tuning only very slightly increases the performance, while the pre-training step greatly improves the model.

Node dropping and subgraphs are good methods for all datasets. Attribute masking is also decent and especially good for social networks. The specific strategy that selects which nodes are augmented differs. Node dropping and subgraphs both do not need an advanced strategy. The random methods already improve the accuracy, yet selecting a strategy that depends on the dataset is highly beneficial. An example is the COLLAB dataset. With mean-accuracy, random node dropping (1) increases the result by 1.85 percentage points. Dropping high-degree nodes instead (2) increases the results by 2.36 percentage points. Accordingly, the increase for a random subgraph (12) is 1.45 percentage points, while a subgraph with low-degree nodes (16) improves by 1.75 percentage points. We also find examples where some node selection strategies perform badly, even though the random strategy is good. Unsurprisingly, opposite strategies like dropping high-degree and low-degree nodes usually show opposite results. If we look at the same subgraph example with COLLAB, we see that most strategies like depth-search or width-search not only perform worse than the random variant. They also result in a performance decrease compared to the baseline.

We can observe a general tendency in the different node selection strategies. Specifically, we observe that removing high-degree nodes and keeping the low-degree nodes is often beneficial. We see this effect in node dropping and subgraphs. Dropping high-degree nodes with a higher chance performs better than random node dropping while dropping low-degree nodes is significantly worse. The same goes for subgraphs. Building the subgraph with high-degree nodes performs worse than building the subgraph with low-degree nodes. Consequently, the three augmentations 17-19 also perform worse because these node centralities tend to assign a higher score to nodes that receive

		Steiner tree augmentation		
	Dataset	random (21)	low-degree(22)	weighted (23)
mean-acc	PROTEINS	1.90	2.06	1.84
	NCI1	0.35	0.25	0.03
	FRANKENSTEIN	1.98	4.26	4.29
	COIL-DEL	18.87	7.98	7.90
	COLORS-3	13.43	14.68	15.24

		Steiner tree augmentation		
	Dataset	random (21)	low-degree(22)	weighted (23)
max-acc	PROTEINS	0.79	0.82	0.46
	NCI1	-0.16	0.20	0.20
	FRANKENSTEIN	1.63	3.35	3.31
	COIL-DEL	19.10	8.00	7.93
	COLORS-3	13.74	14.64	15.44

Table 7.3: The table shows the performance of the three steiner tree methods. We report an increase in percentage points compared to the baseline. We use 20% of the nodes as the terminal nodes. The real augmentation is around 60%.

more edges. Since we have an undirected graph, higher-degree nodes usually get a high score. PROTEINS, COLLAB, GITHUB, RDT-B, and COIL-DEL show that keeping low-degree nodes is beneficial. The FRANKENSTEIN dataset is an outlier for this behavior. Building the low-degree subgraph (16) is the worst subgraph strategy for this dataset. The high-degree subgraph (15) is the best.

Another effect we see is the difference between masking node attributes with the average or with 0. Masking with 0 outperforms the mean masking in all datasets we tested. As mentioned in the augmentation Chapter 6, we assume that reason for this is the fact that the node attributes are one-hot vectors in our datasets. The continuous average value is an outlier and can not be handled well by the model.

We mentioned earlier that adding nodes, switching node attributes, edge perturbation, adding edges, and removing edges do not perform well (5, 8, 9, 10, 11). Often they are not only worse than the other augmentations, but they are often worse than the baseline. What these methods have in common is that they do not change the original nodes. It makes sense that the results are similar to using no augmentation (Identical). The key structure of the graph is unchanged. All nodes, most edges, and most attributes are still the same. Other methods, like subgraphs, remove complete parts of the graph.

Merging nodes (20) is a mixture of node dropping and building a subgraph. The results from merging nodes are close to the random variants of node-dropping and subgraph. The method does not generally improve the performance of either of them by much.

We show the results for five datasets with the three steiner tree augmentations in Table 7.3. As previously mentioned, the steiner tree methods have the disadvantage that we can not pick a specific augmentation ratio. The results use a terminal set with 20% of the nodes, which results in about 60%

augmentation ratio. The three different variants show no clear favorite. The random variant (21) performs better with COIL-DEL. Selecting the low-degree nodes (22) and the weighted approach (23) perform better with FRANKENSTEIN. For most datasets, there are better augmentations than the steiner trees. FRANKENSTEIN is an exception. Here they are significantly better than the other augmentations with 20% augmentation ratio. However, FRANKENSTEIN is similar to datasets like PROTEINS and benefits from higher augmentation ratios. Part of the increased performance for FRANKENSTEIN is because we apply a higher augmentation ratio. Steiner trees are generally an interesting approach to augmentations. Nevertheless, the results can not compensate for the disadvantages of not being able to pick exact augmentation ratios and infeasible computation times for the big datasets.

As mentioned before, we see that different datasets can show different behavior. Datasets like PROTEINS improve with most augmentations, while NCI1 shows the opposite effect. Here nearly all augmentations result in slightly lower accuracy than the baseline. A simple assumption is that similar datasets behave the same. To define a dataset as similar, multiple parameters can be compared. Examples are the source of the data and their general type, the number of graphs, the number of nodes and edges, and the average degree. As a simple example, we compare NCI1 and FRANKENSTEIN. They both are graphs of small molecules, and while NCI1 has twice as big graphs, their average degree is the same. We expect these very similar datasets to show similarities in the performance of the augmentations and augmentation ratios. As we see in Figure 7.1, this is not the case for these two datasets. While NCI1, in general, is a dataset that is improved very little by all tested augmentations, FRANKENSTEIN shows improvements in nearly all augmentations. However, not only the magnitude of improvement is different. As we previously mentioned, NCI1 shows the usual pattern of getting better performance when we remove high-degree nodes. FRANKENSTEIN performs better when we keep the high-degree nodes. This effect shows in node dropping, the subgraph method, and the three metrics from augmentation 17-19. There is no simple way to predict whether an augmentation will perform well on a dataset. Finding datasets that show the same behavior by looking only at these simple parameters is impossible. This task would require at least more profound in-depth knowledge of the datasets.

7.3.1 Augmentation Ratios

After analyzing the single augmentation performance, we now evaluate different augmentation ratios. We compare their impact in detail for PROTEINS in Figure 7.2 and for NCI1 in Figure 7.3. We evaluate the combination of the original graph (Identical) with all augmentations. The top plot shows the results when we use the mean-accuracy. The bottom plot shows the results for the max-accuracy. On the y-axis, we see the augmentation ratio. The x-axis shows the applied augmentation. Previous research [9] often uses 20% as a default augmentation ratio, and not much time has been invested into researching different augmentation ratios. We test augmentation ratios in the range of 5%, 20%, 35%, 50%, 65%, 80%, and 95%. We observe that a higher augmentation ratio often gives better results. We see this in the PROTEINS dataset in Figure 7.2. Other datasets like FRANKENSTEIN, COLLAB, and RDT-B show the same behavior. The optimum for these datasets is between 60% and 80%. In contrast to this, other datasets show the reverse behavior. For the NCI1 dataset in Figure 7.3, most augmentations get worse with higher augmentation ratios. In general, nearly all augmentations for NCI1 result in worse accuracy. We somewhat expect that a bad augmentation, which is applied more, results in a worse performance. For the PROTEINS

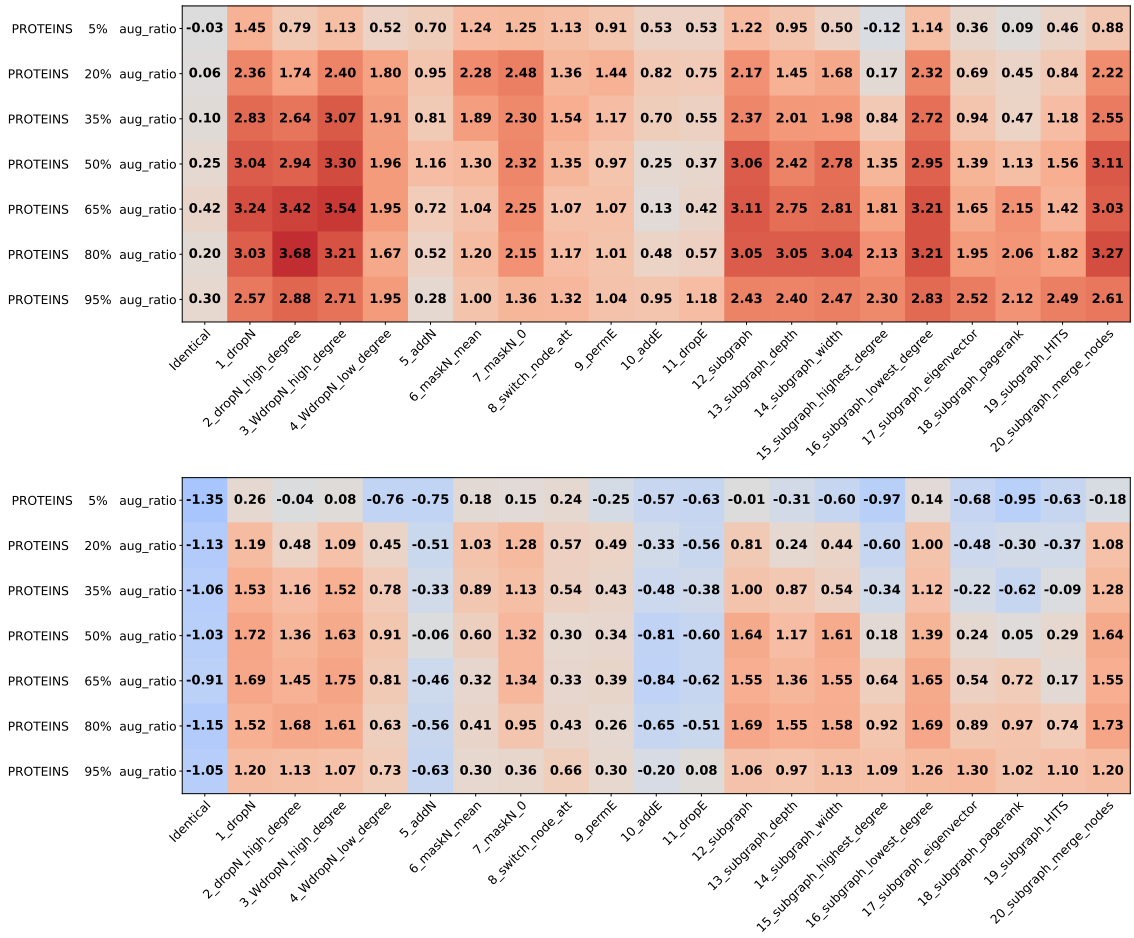

Figure 7.2: Augmentation ratio comparison for PROTEINS. The results are the change in percentage points compared to our baseline. The top plot shows the results when we use the mean-accuracy. The bottom plot shows the results for the max-accuracy. On the y-axis, we see the augmentation ratio. The x-axis shows the applied augmentation.

dataset, we manage to increase our best mean-accuracy result with 20% augmentation ratio from 1.28 to 1.75 percentage points. The max-accuracy increases from 2.48 to 3.68 percentage points With different augmentation ratios, we improved our previous results by 40%. We conclude that the augmentation ratio is a very important, so far neglected, parameter.

7.3.2 Augmentation Combinations

We test the combination of augmentations in Figure 7.4, Figure 7.5, and Figure 7.6. Due to computational limits, we can not test all possible combinations with all datasets. We use ten of the most interesting augmentations from our previous single performance evaluation and evaluate them with the PROTEINS dataset. We do see, however, similar effects on other datasets. We test three different configurations: both augmentations with 20%, both with 80%, and one augmentation with 20% and the other with 80% augmentation ratio.

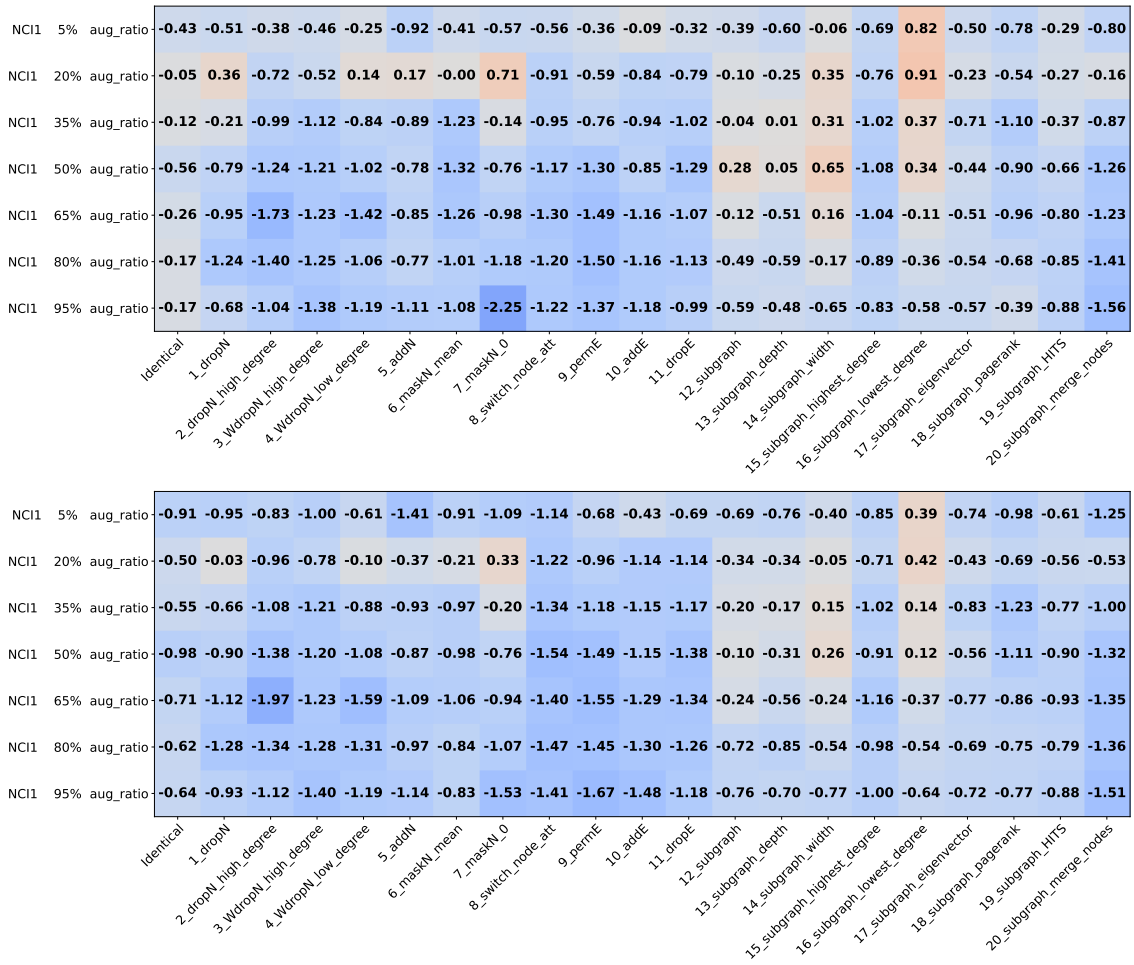

Figure 7.3: Augmentation ratio comparison for NCI1. The results are the change in percentage points compared to our baseline. The top plot shows the results when we use the mean-accuracy. The bottom plot shows the results for the max-accuracy. On the y-axis, we see the augmentation ratio. The x-axis shows the applied augmentation.

The first image, Figure 7.4, shows both augmentations with 20% augmentation ratio. Compared to our single performance before in Figure 7.2, the best mean-accuracy combination of augmentations increases from 2.48 to 3.34 percentage points. The best max-accuracy combination increases from 1.28 to 1.84 percentage points. Interestingly, the two best single augmentations from before do not yield the best augmentation combination. For mean-accuracy, dropping high-degree nodes (3) and masking nodes (7) is the best combination. They only increase the accuracy by 2.40 or 2.65 percentage points, depending on the order of both augmentations. They are slightly better than their previous single performance, 2.40 and 2.48 percentage points. Generally, most of the 20% augmentation ratio combinations seem to benefit from the combination and result in higher accuracy than both augmentations individually. Augmentations like the depth-first subgraph (13) increased noticeably throughout all tested combinations. There are a few combinations that do perform worse. Most noticeably, the combination of two times 0-masking (7) performs worse than the single version and even worse than the baseline. Another thing we notice is that the order of the augmentations does matter. We do not see perfect symmetry between the table's lower left

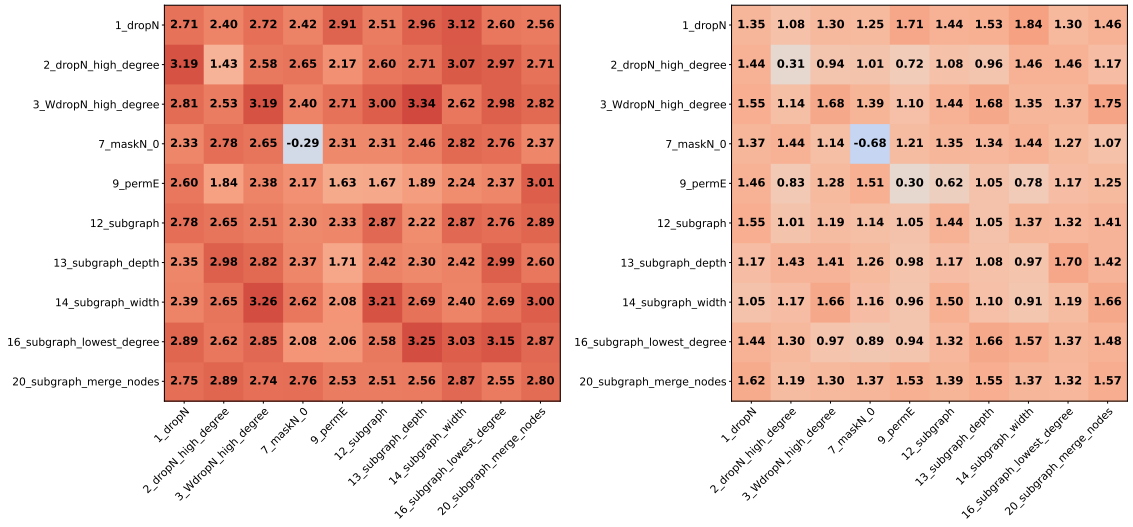

Figure 7.4: Evaluation of augmentation combinations for PROTEINS. The results are the change in percentage points compared to our baseline. The left plot shows the results when we use the mean-accuracy, the right plot for the max-accuracy. The x-axis and y-axis, define the augmentation combination. All augmentations use 20% augmentation ratio and are averaged over five runs.

and upper right sides. Some combinations show differences. For example, the mean-accuracy for subgraph (9) and edge perturbation (12) increases to 2.33 or 1.67 percentage points, depending on the order of both augmentations. Although the magnitude of the combination, if it is bad, average, good, or very good, is not dependent on the order. Especially the smaller graph datasets generally suffer from high variance. We attribute most of the difference to the variance.

The second image, Figure 7.5, shows both augmentations with 80% augmentation ratio. Similarly to the combination of 20% augmentation ratios, we see a general increase by combing the augmentations. Compared to before, we get higher accuracies for more combinations but also more combinations with lower accuracies. The 20% augmentation ratio combinations were closer to the average. For 80% we get more high and low values. However, we can say the same for the single combination with 20% augmentation ratio compared to 80% augmentation ratio. This result is, therefore, not surprising. The best augmentation combination increased the mean-accuracy from 3.68 to 3.87 percentage points. The max-accuracy increased from 1.73 to 2.23 percentage points. Compared to before, the increase from the combination for 80% augmentation ratio is noticeably lower than for 20% augmentation ratio. The worst method is once again the combination of two times 0-masking (augmentation 7). The performance has significantly decreased. For 20% to 80% augmentation ratio, the single mean-accuracy decreased from 2.48 to 2.15. At the same time, the combination of 20% with -0.29 percentage points decreased to -2.49 percentage points with the combination of 80%. While the single accuracy only slightly decreased with the higher augmentation ratio, the decrease of the combination was much higher. This supports our previous observation, in which higher augmentation ratios result in more outliers in both positive and negative directions. This result indicates that while combing augmentations is generally very beneficial, there might be a turning point for the augmentation ratio when some combinations stop benefiting from the combination and start to decrease.

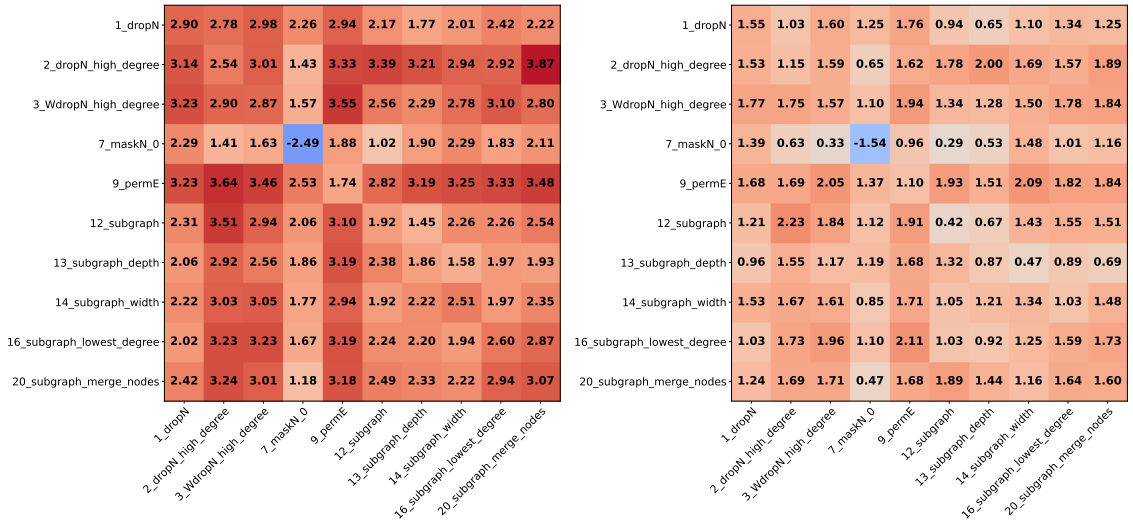

Figure 7.5: Evaluation of augmentation combinations for PROTEINS. The results are the change in percentage points compared to our baseline. The left plot shows the results when we use the mean-accuracy, the right plot for the max-accuracy. The x-axis and y-axis, define the augmentation combination. All augmentations use 80% augmentation ratio and are averaged over five runs.

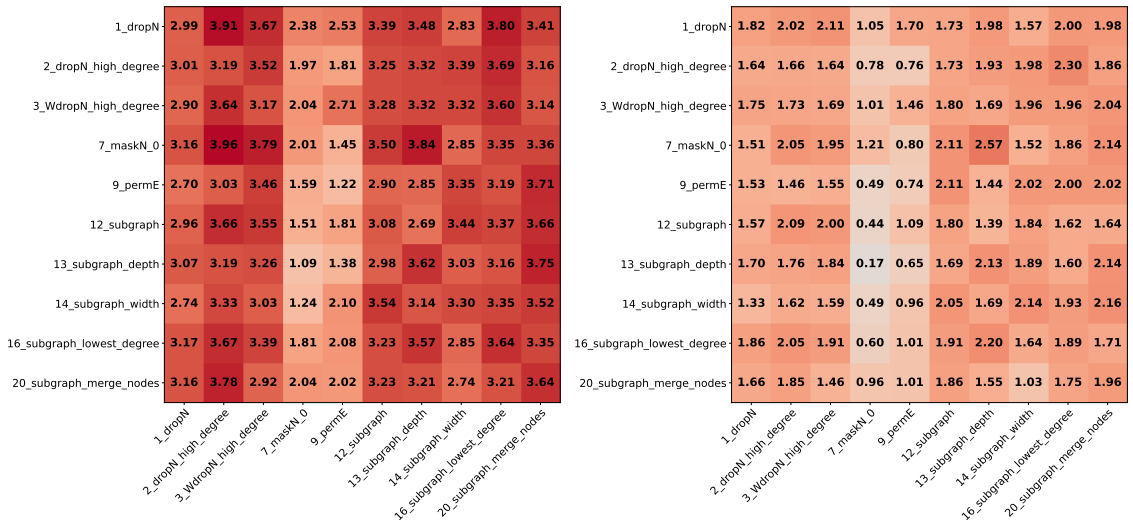

Figure 7.6: Evaluation of augmentation combinations for PROTEINS with Augmentation 1 20% (left) - Augmentation 2 80% (bottom). The results are the change in percentage points compared to our baseline. The left plot shows the results when we use the mean-accuracy, the right plot for the max-accuracy. The x-axis and y-axis, define the augmentation combination. All augmentations use 80% augmentation ratio and are averaged over five runs.

Finally, we test the third option, in Figure 7.6, where one augmentation has 20% augmentation ratio and the other one 80% augmentation ratio. The best augmentation increases for the mean-accuracy from 3.68 to 3.96 percentage points. The max-accuracy increased from 2.23 to 2.57 percentage points. While the best combination only slightly increased, we notice that we have many more combinations with high accuracy than before. 20%-20% has 13 combinations with ≥3.00 percentage points increase, 80%-80% has 28 combinations, 20%-80% has 66 combinations. A surprising result is that the negative outlier from before, two times 0-masking (7), has vanished. While two times 20% resulted in -0.29, and two times 80% resulted in -2.49, combining 20% 0-masking with 80% 0-masking results in an increase of 2.01 percentage points.

We have seen that combining different augmentations, especially with different augmentation ratios, results in better pre-training. While many combinations result in good performances, we are interested in the highest accuracy. The difference between the best single augmentation and the best combination is low. For mean-accuracy, the best single augmentation is dropping high-degree nodes (2) with 80% augmentation ratio, which results in a 3.68 percentage points increase. The best combination is 0-masking (7) with 20% augmentation ratio and dropping high-degree nodes (2) with 80%, which increases the result by 3.96 percentage points. We found a better combination, but it took us 300 combinations for a small increase, even though we only tested half of our augmentations and two different augmentation ratios. Generally, a good approximation can be found quickly by testing the generally good single augmentations and combining them with low and high augmentation ratios. Since there are infinitely many possible combinations, an unknown, very good combination might exist.

7.4 Loss Functions

To evaluate the different loss functions, we use an augmentation that performs well for nearly all datasets combined with the original graph. Identical is the first augmentation, and the lowest degree subgraph (16) with 20% augmentation ratio is the second augmentation. We evaluate all loss functions with the basic settings described in Section 7.2 and run the experiments 20 times to calculate the average mean-accuracy and max-accuracy. We test seven different loss functions. Cosine (5.1), euclidian (5.3), TS-SS sum (5.20), TS-SS ratio (5.21), Triplet (5.12), Weighting-1 (5.8), and Weighting-2 (5.9). For the following evaluations, we use m=5 for the margin. We later see in our evaluations that m=5 is a reasonable parameter.

We show the results in Table 7.4. The top table shows the mean-accuracy results, and the lower table shows the max-accuracy results. Bold numbers are the reference values for the default loss of using one positive with 127 negatives and our NT-Xent loss variant, which we call cosine in the table. We highlight the best loss function in red. As augmentations, we use Identical and the lowest-degree subgraph (16) with an augmentation ratio of 20%. The accuracy values we have to compare our new loss functions to are a 2.46 percentage point increase for the mean-accuracy and a 1.09 percentage point increase for the max-accuracy with the PROTEINS dataset. The NCI1 dataset increases by 0.57 percentage points for the mean-accuracy and by 0.25 percentage points for the max-accuracy. These are the previous results for the selected augmentation and augmentation ratio when we use our default loss.

		positive samples	1	1	1	1	1	1	4
		negative samples	0	1	2	10	50	127	8
mean-acc	PROTEINS	cosine	1.03	2.34	2.46	2.52	1.33	**2.46**	2.66
		euclidian	0.95	2.75	2.80	2.27	1.71	1.46	2.94
		TS-SS sum	0.54	2.26	2.04	2.13	1.85	2.13	2.06
		TS-SS ratio	-	1.14	1.32	1.31	1.24	1.29	0.58
	NCI1	cosine	-0.92	0.12	0.14	0.64	-0.14	**0.57**	0.17
		euclidian	-0.95	-0.71	-0.38	-0.21	-0.05	-0.05	-0.22
		TS-SS sum	-0.10	-0.22	-0.40	-0.18	-0.22	-0.01	-0.40
		TS-SS ratio	-	-0.50	-0.48	-0.42	-0.55	-0.35	-0.14

		positive samples	1	1	1	1	1	1	4
		negative samples	0	1	2	10	50	127	8
max-acc	PROTEINS	cosine	-0.24	1.08	1.20	1.04	0.14	**1.09**	1.27
		euclidian	-0.23	1.29	1.53	0.84	0.44	0.21	1.30
		TS-SS sum	-0.67	1.11	1.02	1.09	0.84	0.97	0.79
		TS-SS ratio	-	-0.14	-0.27	0.03	-0.13	0.08	-0.52
	NCI1	cosine	-0.78	-0.11	-0.05	0.28	-0.30	**0.25**	-0.08
		euclidian	-0.82	-0.75	-0.58	-0.38	-0.25	-0.28	-0.38
		TS-SS sum	-0.08	-0.36	-0.44	-0.28	-0.45	-0.23	-0.45
		TS-SS ratio	-	-0.41	-0.46	-0.44	-0.42	-0.54	-0.09

Table 7.4: Comparison of four loss functions. The **bold** numbers are the reference value for the cosine loss with one positive and 127 negatives, the method we have used so far. The red numbers are the best loss functions. The table shows the percentage point increase in accuracy of the pre-training step compared to only fine-tuning.

We first look into the different loss functions with the default setting of 127 negatives. The cosine loss significantly outperforms the euclidian loss, while the TS-SS sum performance is between both. The TS-SS ratio loss is consistently worse than the TS-SS sum. Generally, cosine distance is regarded as a good distance metric in the high dimensional space, so we expect these results.

For all loss functions using no negatives results in worse performance. Since negative samples are a crucial part of contrastive learning, we expect these results. Interestingly, the mean-accuracy for PROTEINS is still positive, although worse than with negatives, compared to the baseline. Pre-training without negatives can still be helpful on some datasets.

In the next step, we lower the negative samples for the different loss functions. For cosine loss, we see a drop-off in performance when we go down to 50 negatives. For 2-10 negatives, we get the same performance or slightly better as for 127 negatives. This result contradicts the assumption from image representation learning, where previous research [7] suggests that a high number of negative samples benefits contrastive learning. The euclidian loss shows a different behavior, depending on the dataset. For PROTEINS, the performance increases the fewer negatives we use, down to two negatives being the optimal result. For NCI1, we get the optimal performance with 127 negatives and only get worse with fewer negatives. Interestingly, the best performance of the euclidian loss results in a 2.80 percentage point increase in the mean-accuracy. The best-performing cosine loss

		margin m	0.1	1.0	2.5	5.0	10.0	20.0
mean-acc	PROTEINS	euclidian	0.99	1.45	2.17	2.79	2.57	2.79
		Triplet	1.40	1.88	1.85	2.02	1.93	2.08
	NCI1	euclidian	-0.91	-0.64	-0.47	-0.36	-0.41	-0.33
		Triplet	-0.44	-0.59	-0.51	-0.55	-0.43	-0.62

		margin m	0.1	1.0	2.5	5.0	10.0	20.0
max-acc	PROTEINS	euclidian	-0.35	0.47	0.87	1.57	1.36	1.58
		Triplet	0.28	0.89	0.71	0.99	0.97	1.07
	NCI1	euclidian	-0.83	-0.63	-0.53	-0.45	-0.53	-0.49
		Triplet	-0.56	-0.71	-0.73	-0.64	-0.68	-0.70

Table 7.5: Evaluation of the margin parameter m. The red numbers are the best margin. The blue numbers are the second-best margin.

only increases the mean-accuracy by 2.52 percentage points. Similarly, for the max-accuracy, the best euclidian results outperform the best cosine result with a 1.53 over a 1.20 percentage point increase. Both the TS-SS sum and TS-SS ratio are less affected by the number of negatives. They show similar results throughout all settings. We additionally tested the Triplet loss, for which the number of negatives is always one, and the calculation is similar to the max-margin euclidian loss formula. Table 7.5 shows the results for different margin parameters m. The higher margins between 5-20 result in the best performance. TS-SS sum and Triplet loss have a similar performance and are generally better than the TS-SS ratio. Euclidian and cosine loss with few negatives remain the best two options. For PROTEINS, the euclidian loss with two negatives, and for NCI1, the cosine loss with ten negatives has the best performance.

At last, we now increase the number of positives. We compare ourselves to the results from one positive and two negatives. We now increase the number of positives to four. Accordingly, we also increase the number of negatives from two to eight as described in Section 5.2.3. For PROTEINS, we see an increase for both cosine and euclidian loss, TS-SS sum, and TS-SS ratio are unaffected or even decrease slightly. The cosine loss increases from 2.46 to 2.66 percentage points for the mean-accuracy and from 1.20 to 1.27 percentage points for the max-accuracy. The euclidian loss increases from 2.75 to 2.94 percentage points for the mean-accuracy and from 1.29 to 1.30 percentage points for the max-accuracy. For NCI1, we see no improvements or only small improvements for all loss functions, some of which are negative. We can improve our cosine and euclidian results with more positive samples. By increasing the number of positives by a factor of four, we also increase the training the model gets. Later evaluations of transfer learning, warm-starting, and increased pre-training episodes show better improvements with a smaller increase in training. Increasing the number of positive samples in the loss function does not seem to be a valuable trade-off between resources and results.

		No weight	Weighting-1	Weighting-2
mean-acc	PROTEINS	2.75	2.80	2.62
	NCI1	-0.71	-0.68	-0.61

		No weight	Weighting-1	Weighting-2
max-acc	PROTEINS	1.29	1.52	1.30
	NCI1	-0.75	-0.61	-0.77

Table 7.6: Evaluation of two weighted loss functions for the euclidian max-margin loss. The top table shows the mean-accuracy results, and the lower table shows the max-accuracy results. The red numbers are the best loss function.

7.4.1 Margin Parameter

The euclidian function and the Triplet loss introduce an additional parameter m, the max-margin, as described in Section 5.2.1. In Table 7.5, we test the influence of the margin parameter m in the euclidian loss and the triplet loss. The top table shows the mean-accuracy results, and the lower table shows the max-accuracy results. As augmentations, we use Identical and the lowest-degree subgraph (16) with an augmentation ratio of 20%. The table shows the percentage increase in accuracy of the pre-training step compared to only fine-tuning. We highlight the best margin in red and the second best in blue.

The euclidian loss performs best with a margin of 5-20 on both PROTEINS and NCI1. The results above five do stay the same and do not increase further. The Triplet loss also performs best for PROTEINS with a margin between 5-20. For NCI1, a low margin of 0.1 also shows a good performance. We conclude that m=5 is a reasonable margin parameter that performs well in all our tests.

7.4.2 Weighting Loss Functions

We also test an improvement of the euclidian loss. We weight the loss of graphs in the mini-batch compared to the rest. We describe the two formulas in Section 5.2.4. We want to assign more weight to the loss of a graph if the positive distance is high and the negative distance is low. Weighting-1 does not limit the negative distance, while Weighting-2 limits the size of the negative loss to the margin m=5. The reference result is the euclidian loss with one negative, called no weight. We show the results in Table 7.6 and mark the best methods with red. As augmentations, we use Identical and the lowest-degree subgraph (16) with an augmentation ratio of 20%. The table shows the increase in percentage points compared to the baseline.

We see that the weighting functions are better in both settings and datasets. The mean-accuracy increased from 2.75 to 2.80 percentage points for the PROTEINS dataset. The max-accuracy decreased from 1.29 to 1.52 percentage points. For the NCI1 dataset, the mean-accuracy increased from -0.71 to -0.61 percentage points. The max-accuracy increased from -0.75 to -0.61 percentage points. This simple example shows that a good weighting function improves the loss function further.

Pre-training	Episodes	Fine-tuning	Episodes	Shrink & Perturb	mean-acc	max-acc
PROTEINS	100	PROTEINS	100	No	2.32	1.00
PROTEINS	100	PROTEINS	100	Yes	-0.35	-1.29
PROTEINS	100	PROTEINS	100	Average weight	-2.70	-1.05
NCI1	100	NCI1	100	No	0.37	0.14
NCI1	100	NCI1	100	Yes	-1.22	-0.82
NCI1	100	NCI1	100	Average weight	-0.90	-0.59
FRANKEN.	100	FRANKEN.	100	No	0.30	0.19
FRANKEN.	100	FRANKEN.	100	Yes	0.45	0.58
FRANKEN.	100	FRANKEN.	100	Average weight	0.10	0.23

Table 7.7: Results from testing the shrink & perturb step in our default setup. We apply it between pre-training and fine-tuning.

7.5 Shrink & Perturb

Warm-starting describes that we start a neural network with trained weights instead of randomly initializing them. We do this in our fine-tuning step. Ash et al. [20] describe a general approach to improve warm-starting neural networks. We describe their approach in Section 5.3 and present the results in Table 7.7. Between pre-training and fine-tuning, we apply warm-starting, but until now, without a shrink & perturb step, which we use as a reference. The default shrink & perturb variant is by Ash et al. with the parameters they found work best (5.22). We call our adaptation the average weight method (5.23)

For PROTEINS and NCI1, both shrink & perturb methods do not improve the results. They decrease them very strongly. The FRANKENSTEIN dataset does profit from the default shrink & perturb. The mean-accuracy increases from 0.30 to 0.45 percentage points. For the max-accuracy, we increase from 0.19 to 0.58 percentage points. When we use max-accuracy, the average weight method also slightly increases the results from 0.19 to 0.23. For the mean-accuracy, the accuracy decreases more strongly from 0.30 to 0.10 percentage points.

We see in later evaluations that the FRANKENSTEIN dataset profits from fewer episodes, potentially due to fast overfitting. The shrink & perturb step could revert part of the overfitting. Generally, the shrink & perturb does not improve the results when we use it between pre-training and fine-tuning the same dataset. In most cases, it strongly decreases the accuracy. During the next section, we test the effect of transfer learning. During this evaluation, we test the shrink & perturb method in the setting where the dataset on pre-training and fine-tuning is different.

7.6 Transfer Learning

In this section, we test the performance of transfer learning. Since the pre-training does not rely on labels, the assumption is that we can pre-train with different other datasets as well. We described the approach in Section 5.4. We show the results in Table 7.8. We might profit from more training due to more episodes or larger datasets. If we switch out the pre-training dataset with a much larger dataset, the larger training size might have a more significant impact than the data itself. To make

Pre-training			Fine-tuning				
Dataset	Episodes	Total #Graphs	Dataset	Episodes	Shrink & Perturb	mean-acc	max-acc
PROTEINS	100	111.300	PROTEINS	100	No	**2.32**	**1.00**
PROTEINS	370	411.810	PROTEINS	100	No	2.51	1.20
NCI1	100	411.000	PROTEINS	100	No	1.00	1.33
NCI1	100	411.000	PROTEINS	100	Yes	-0.93	1.04
NCI1	100	411.000	PROTEINS	100	Average weight	-2.43	-1.87
PROTEINS	390	434.070	PROTEINS	100	No	2.32	0.96
FRANKEN.	100	433.700	PROTEINS	100	No	1.06	1.41
FRANKEN.	100	433.700	PROTEINS	100	Yes	-0.44	1.83
FRANKEN.	100	433.700	PROTEINS	100	Average weight	-1.22	0.28

Pre-training			Fine-tuning				
NCI1	100	411.000	NCI1	100	No	**0.37**	**0.14**
NCI1	27	110.970	NCI1	100	No	0.15	-0.09
PROTEINS	100	111.300	NCI1	100	No	-0.17	-0.28
PROTEINS	100	111.300	NCI1	100	Yes	-1.15	-0.76
PROTEINS	100	111.300	NCI1	100	Average weight	-0.71	-0.48
NCI1	100	411.000	NCI1	100	No	0.37	0.14
FRANKEN.	100	433.700	NCI1	100	No	-0.37	-0.37
FRANKEN.	100	433.700	NCI1	100	Yes	-1.15	-0.71
FRANKEN.	100	433.700	NCI1	100	Average weight	-0.72	-0.49

Pre-training			Fine-tuning				
FRANKEN.	100	433.700	FRANKEN.	100	No	**0.30**	**0.19**
FRANKEN.	26	112.762	FRANKEN.	100	No	0.90	0.83
PROTEINS	100	111.300	FRANKEN.	100	No	0.21	0.32
PROTEINS	100	111.300	FRANKEN.	100	Yes	0.50	0.59
PROTEINS	100	111.300	FRANKEN.	100	Average weight	0.10	0.37
FRANKEN.	100	433.700	FRANKEN.	100	No	0.30	0.19
NCI1	100	411.000	FRANKEN.	100	No	0.23	0.35
NCI1	100	411.000	FRANKEN.	100	Yes	0.52	0.58
NCI1	100	411.000	FRANKEN.	100	Average weight	0.20	0.33

Table 7.8: We test the performance of transfer learning. The **bold** numbers are the reference value for 100 episodes of pre-training with the same dataset. We compare them to transfer learning with a different dataset. Since other datasets have different sizes, we offer a version with increased/decreased training with the same dataset to compare the results. The red numbers mark the best of the four variants.

our results more comparable, we always give the base performance for 100 episodes of pre-training with the same dataset, for example, with the PROTEINS dataset. We then test the performance for 100 episodes of pre-training with different datasets like NCI1 and FRANKENSTEIN. Since NCI1 is 3.7 times as large as PROTEINS and FRANKENSTEIN is 3.9 times as large, we additionally state the performance for 370 and 390 episodes of pre-training with PROTEINS. This way, we ensure that the total number of graphs during pre-training is comparable. The amount of pre-training is still different since other parameters, like the size of the graphs, also influence the training. However, we get a better approximation to compare our results.

If we look at the first example, the table at the top in Table 7.8, we fine-tune on PROTEINS. The base results are 2.32 percentage points for mean-accuracy and 1.00 for max-accuracy. For mean-accuracy, transfer learning does not work with either NCI1 or FRANKENSTEIN. They perform significantly worse while increasing the episodes of PROTEINS to 370 slightly increases the results. For max-accuracy, pre-training with NCI1 without shrink & perturb increases the results from 1.00 to 1.33 percentage points, while training with 370 episodes only results in a 1.20 percentage point increase. For FRANKENSTEIN, the max-accuracy results are even better. The results increase both with and without shrink & perturb. We significantly increased the results from 1.00 to 1.83 percentage points with shrink % perturb.

The middle table shows the results of fine-tuning on NCI1. For NCI1, none of the different transfer learning setups can improve the results. They are worse than the reference value and even negative compared to the baseline.

In the lower table, we see the results for fine-tuning on FRANKENSTEIN. Interestingly we see that lowering the episodes for FRANKENSTEIN increases the results, which suggests that FRANKENSTEIN overfits very fast during pre-training. Pre-training with PROTEINS does increase the reference results. Nevertheless, as we explained, less training with the same dataset is still better than transfer learning with PROTEINS. Pre-training with NCI1 increases mean-accuracy results only when we apply the shrink & perturb step. Max-accuracy is increased by all three shrink & perturb methods. We get the best performance with the default shrink & perturb, the same as with mean-accuracy. We increase the mean-accuracy from 0.30 to 0.52 percentage points and the max-accuracy from 0.19 to 0.58 percentage points.

We see that transfer learning is, like other methods, dataset-dependent. NCI1 is negatively affected by transfer learning. We perform worse than the baseline without pre-training. We show that other datasets like PROTEINS and FRANKENSTEIN benefit from pre-training with different datasets. In most cases, we perform slightly worse with a different pre-training dataset than using the same dataset. However, this shows that pre-training with completely different graphs can improve fine-tuning. In some cases, we showed better results with transfer learning than with the same dataset. A shrink & perturb step further improved the results in most cases between different datasets. Previous tests of the shrink & perturb step showed that it often does not work when we use the same dataset for pre-training and fine-tuning.

Pre-Pre-training		Pre-training			Fine-tuning				
Dataset	Episodes	Dataset	Episodes	Total #Graphs	Dataset	Episodes	Shrink & Perturb	mean-acc	max-acc
-	-	PROTEINS	100	111.300	PROTEINS	100	No	**2.32**	**1.00**
-	-	PROTEINS	470	523.110	PROTEINS	100	No	2.31	0.98
NCI1	100	PROTEINS	100	522.300	PROTEINS	100	No	3.08	1.70
NCI1	100	PROTEINS	100	522.300	PROTEINS	100	Yes	3.63	1.91
NCI1	100	PROTEINS	100	522.300	PROTEINS	100	Average weight	2.46	1.03
-	-	PROTEINS	490	545.370	PROTEINS	100	No	2.90	1.09
FRANKEN.	100	PROTEINS	100	545.000	PROTEINS	100	No	3.23	1.70
FRANKEN.	100	PROTEINS	100	545.000	PROTEINS	100	Yes	3.65	1.89
FRANKEN.	100	PROTEINS	100	545.000	PROTEINS	100	Average weight	2.52	1.04

Pre-Pre-training		Pre-training			Fine-tuning				
Dataset	Episodes	Dataset	Episodes	Total #Graphs	Dataset	Episodes	Shrink & Perturb	mean-acc	max-acc
-	-	NCI1	100	411.000	NCI1	100	No	**0.37**	**0.14**
-	-	NCI1	127	521.970	NCI1	100	No	0.72	0.40
PROTEINS	100	NCI1	100	522.300	NCI1	100	No	0.29	0.06
PROTEINS	100	NCI1	100	522.300	NCI1	100	Yes	-0.88	-0.49
PROTEINS	100	NCI1	100	522.300	NCI1	100	Average weight	-0.33	-0.08
-	-	NCI1	200	822.000	NCI1	100	No	0.65	0.30
FRANKEN.	100	NCI1	100	844.700	NCI1	100	No	0.34	0.13
FRANKEN.	100	NCI1	100	844.700	NCI1	100	Yes	-0.66	-0.18
FRANKEN.	100	NCI1	100	844.700	NCI1	100	Average weight	-0.40	-0.27

Pre-Pre-training		Pre-training			Fine-tuning				
Dataset	Episodes	Dataset	Episodes	Total #Graphs	Dataset	Episodes	Shrink & Perturb	mean-acc	max-acc
-	-	FRANKEN.	100	433.700	FRANKEN.	100	No	**0.30**	**0.19**
-	-	FRANKEN.	126	546.462	FRANKEN.	100	No	0.01	-0.25
PROTEINS	100	FRANKEN.	100	545.000	FRANKEN.	100	No	0.57	0.45
PROTEINS	100	FRANKEN.	100	545.000	FRANKEN.	100	Yes	0.33	0.20
PROTEINS	100	FRANKEN.	100	545.000	FRANKEN.	100	Average weight	0.18	0.22
-	-	FRANKEN.	200	867.400	FRANKEN.	100	No	-0.36	-0.73
NCI1	100	FRANKEN.	100	844.700	FRANKEN.	100	No	0.62	0.41
NCI1	100	FRANKEN.	100	844.700	FRANKEN.	100	Yes	0.61	0.58
NCI1	100	FRANKEN.	100	844.700	FRANKEN.	100	Average weight	0.42	0.36

Table 7.9: We test the performance of warm-starting. The **bold** numbers are the reference value for 100 episodes of pre-training with the same dataset. We compare them to warm-starting with a different dataset in a pre-pre-training step before our normal pre-training. Since the pre-pre-training introduces additional training, we offer a version with increased pre-training with the same dataset to compare the results. The red numbers mark the best of the four variants.

7.7 Warm-Starting

We call this section warm-starting. We defined warm-starting in Section 5.5. It is a further improvement of transfer learning. Transfer learning has the advantage that we can use additional data for training. One downside is that the data in the pre-training is usually quite different, and we can only use the weights from some layers. We described this problem in Section 5.4. We have seen in the previous section that most transfer learning settings could not improve the results. For NCI1, all the tests have been even worse than the baseline.

Instead of pre-training only on a different dataset, we now keep our original setup of pre-training and fine-tuning on the same dataset. Additionally, we introduce a pre-pre-training step in which we use a different dataset. We want to benefit from the different datasets in the pre-pre-training step, while the pre-training step with the same dataset as the fine-tuning ensures that our model is well suited for the data in the fine-tuning.

Like before, in transfer learning, we offer an alternative of pre-training with more episodes to better compare our different approaches and make sure the impact of the size of the datasets is limited.

We show the results in Table 7.8. In the top table, we see the results from fine-tuning PROTEINS. Increasing the pre-training episodes up to 490 increases the results slightly. We increase the mean-accuracy from 2.32 to 2.90 and the max-accuracy from 1.00 to 1.09. Adding a pre-pre-training step with either NCI1 or FRANKENSTEIN performs for both datasets much better. Both perform nearly identically. The mean accuracy increases from 2.32 to 3.65, while the max-accuracy increases from 1.00 to 1.89. Using the default shrink & perturb step significantly improved the results.

The middle table shows the results for NCI1. We see a similar result as in transfer learning. Increasing the episodes rather than pre-pre-train with other datasets is still the best option. However, we see that only some combinations are worse than the baseline. Especially the case of pre-pre-training with FRANKENSTEIN and no shrink & perturb step is nearly identical to the evaluation without the pre-pre-training. 0.37 to 0.34 percentage points for mean-accuracy and 0.14 to 0.13 percentage points for max-accuracy. In contrast to transfer learning, we do not lose anything by using the FRANKENSTEIN dataset. Nevertheless, increasing the episodes for the NCI1 pre-training is still better.

The lower table shows the results for FRANKENSTEIN as a fine-tuning dataset. We can see again that more pre-training episodes on the FRANKENSTEIN dataset significantly decrease the performance. Similar to transfer learning, we see again that applying the same amount of extra training with another dataset will instead increase the Frankenstein performance. In this setting, the shrink & perturb step performs slightly worse than without it. We manage to increase the mean-accuracy from 0.30 to 0.57 percentage points and the max-accuracy from 0.19 to 0.45 percentage points with PROTEINS as pre-pre-training. Even better is the performance with the bigger pre-pre-training dataset NCI1. Now we increase the mean-accuracy from 0.30 to 0.62 percentage points and the max-accuracy from 0.19 to 0.58 percentage points.

We show that warm-starting with an additional pre-pre-training step increases the results compared to transfer learning. Especially for the PROTEIN dataset, we significantly improved our results with an additional dataset. At the same time, we showed that increasing the training on the same dataset does not improve the results by the same magnitude. We can attribute the improvement to the additional dataset.

7.8 Eval Mode

We make an interesting observation during our evaluations. Instead of using the training mode during fine-tuning, using the eval mode increases the performance for some datasets. We show the results in Table 7.10. 10% train mode is the previous baseline without a pre-training step from

	Dataset	PROTEINS	DD	NCI1	MUTAG	FRANKEN.	COIL-DEL
mean-acc	10% train mode	69.82±5.13	74.30±3.89	73.85±2.89	78.83±8.41	60.85±2.91	21.69±2.09
	10% eval mode	71.95±3.47	74.53±3.88	71.38±2.73	83.58±9.47	61.57±2.27	26.06±2.64
	Full data	75.99±2.91	79.09±3.14	83.03±1.64	88.29±7.92	66.86±2.09	74.66±2.13
max-acc	10% train mode	73.42±3.93	76.75±3.20	75.70±2.71	85.86±6.53	63.84±1.90	22.71±2.06
	10% eval mode	75.11±2.91	77.12±3.12	72.96±2.59	85.65±9.24	64.31±1.56	27.65±2.31
	Full data	78.64±3.93	81.85±3.20	84.77±2.71	94.13±6.53	69.19±1.90	76.84±2.06

	Dataset	IMDB-B	COLLAB	GITHUB	RDT-B	RDT-5K	COLORS-3
mean-acc	10% train mode	66.90±4.80	73.45±1.63	61.01±1.78	86.72±2.11	51.38±2.06	51.78±2.10
	10% eval mode	69.76±5.47	72.69±1.78	66.72±1.70	78.51±3.14	52.92±2.22	95.47±2.26
	Full data	73.68±4.95	82.39±1.55	69.09±1.37	92.28±1.60	56.76±1.55	99.75±0.24
max-acc	10% train mode	69.30±4.56	75.14±1.25	62.57±1.39	89.06±1.54	53.08±1.60	53.34±1.89
	10% eval mode	72.43±5.15	74.39±1.40	67.94±1.54	80.90±2.30	55.26±1.77	97.00±1.27
	Full data	78.36±3.93	84.10±3.20	70.40±2.71	94.00±1.90	58.45±6.53	99.97±2.06

Table 7.10: We compare the baseline performances of the datasets with their variance to an approach where we use the evaluation mode instead of the training mode. 10% train mode and Full data are our previous baseline results for fine-tuning only.

Table 7.2, where we use 10% labeled data. The same applies to the full data, where we instead use 100% labeled data. The 10% eval mode evaluation is identical to the 10% train mode, with the single difference of setting the mode to eval instead of train.

Only three of the 12 datasets decrease with the eval mode. The accuracy of NCI1 and COLLAB decreases slightly, while RDT-B decreases by about eight percentage points. On the other hand, the rest of the nine datasets increases. PROTEINS, DD, FRANKENSTEIN, and RDT-5K only slightly increase. MUTAG, COIL-DEL, IMDB-B, and GITHUB increase by around 3-6 percentage points, while COLORS-3 increases by 44 percentage points, close to the performance of the full data.

For some datasets like PROTEINS, IMDB-B, GITHUB, and RDT-5K, the improvements by the eval mode are equal to or even more significant than the improvements by the current best pre-training approaches from Yin et al. [13], and Suresh et al. [12]. We significantly increase the accuracy, even though we do not use pre-training or other improvements.

There are two differences between the train and eval modes. Firstly, dropout is deactivated during the eval mode. However, in our model, we do not use dropout.

Secondly, batch normalization layers calculate a running estimate of its computed mean and variance during training. In the eval mode, we use both mean and variance from the training for normalization. We use multiple batch normalization layers in our network. Since we only set the eval mode, we never use the train mode. We do not calculate any values for the running estimate of mean and variance. We have to use the default values in both training and evaluation. The values are mean=0 and variance=1 and are never changed.

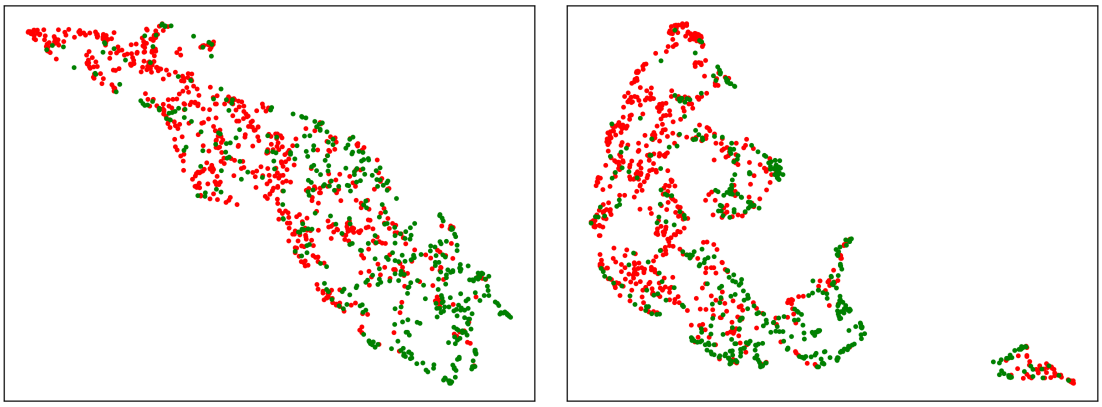

Figure 7.7: UMAP of the PROTEINS embedding space with two classes. Left: 1. Episode, Right: 100. Episode

7.9 Select the Best Augmentation for Each Graph

This section tests methods individually to select augmentations for each graph. We tested multiple approaches. A good performance usually depends on many parameters. Most importantly, we need a metric to decide the best augmentation for a graph. Finding such a metric comes down to the general problem that we need to know what a good augmentation is, which is also dataset dependent. We propose different approaches to select an augmentation from the embedding space. We show an example of the embedding space for PROTEINS visualized by UMAP in Figure 7.7. The left image shows the embedding space in the first episode, and the right image after 100 episodes of contrastive learning. The embedding is initially decent. We can see some separation between the two classes. We increased it through the contrastive pre-training slightly. The two classes are more grouped. We get fewer outliers directly next to other classes. Since we do not have the label information, in which class the graph is, we still end up in a pretty noisy embedding and can not separate the classes perfectly. Our idea is to use this embedding space and the position of different augmentations to approximate good augmentations.

To do that, we define a set of augmentations and augmentation ratios. We calculate all the combinations and their embedding in our model. We show two graphs with their augmentations in the embedding space in Figure 7.8. The augmentations all have 20% augmentation ratio. As expected, similar augmentations end up close together. We also see that methods like edge perturbation (9) tend to be very close to the original. Since we only change edges and none of the nodes or node attributes, this makes sense.

We have different options for how we pick augmentation from the embedding space. We could pick the augmentation that is furthest away from the original graph. The furthest augmentation has proven to be suboptimal because high augmentation ratios can lead to augmentations being very far away because they have little in common with the original graph. We should not pick these augmentations as positive samples since there is a high chance that these graphs are so far apart that they would not share the same label. With a similar idea, we could pick the augmentation closest to the original graph. This approach has the problem that results with a low augmentation ratio are selected, and we get a result close to using two times the original, identical graph. We have

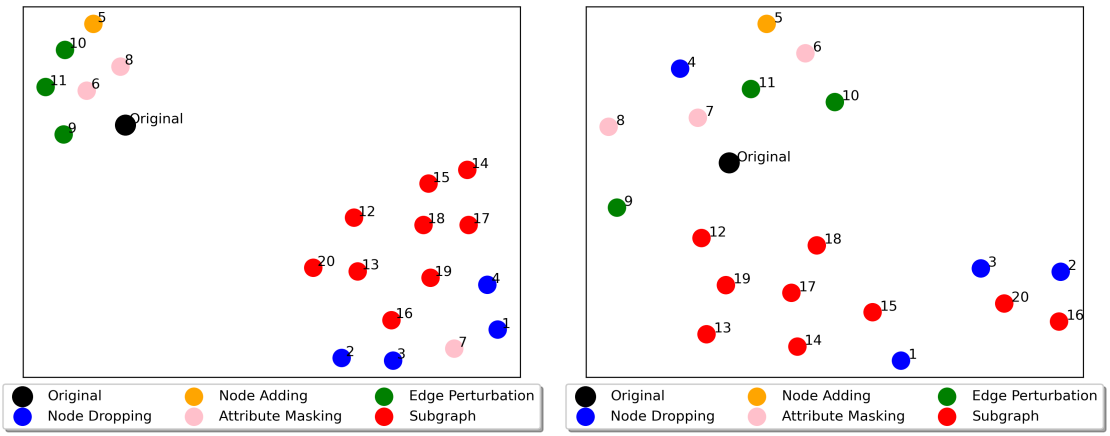

Figure 7.8: UMAP of the embedding space for one graph of PROTEINS with all augmentations. Similar augmentations have the same color.

seen this in the example in Figure 7.8 where augmentations like edge perturbation are very close to the original. Our previous results showed that these augmentations have mostly bad performances, which makes sense since they are similar to the original graph. Other ideas are picking the two augmentations that are furthest apart or closest together. They do suffer from similar problems.

For our evaluation, we instead use the variant where we build a mean graph of all the currently selected augmentations. We select the one augmentation closest to the mean graph and combine it with the original graph (Identical). We do this because contrastive learning does not only depend on a single graph. Because all other graphs in the batch are negatives, they influence each other. By picking a different augmentation for one graph in the batch, we change the loss of all the other graphs. The positive and all negative samples end up being very similar by picking augmentations close to the mean graph. While contrastive learning pushes negative samples apart, we pick augmentations that are close together and work against the model. This approach increases the difficulty of contrastive learning, which is generally beneficial.

Since the augmentation ratio is an important factor, datasets like PROTEINS tend to pick different augmentations, all with a very high augmentation ratio. The results are better than our benchmark from Figure 7.1, where we compared the different augmentations with 20% augmentation ratio. We can, however, not confirm if the improvements are due to the individual augmentation selection or because we use a higher augmentation ratio. To have values to compare to, especially for the other datasets, we decide to limit the augmentation ratio in our experiments to only 20%. In this case, we can directly compare our results to the earlier test of the single augmentation performance. We use augmentations 1-20. We have seen in our first tests that selecting a new augmentation for every graph in every episode does not work. This does not work because we pick the augmentation depending on the embedding of the model. As previously mentioned, the augmentation selection impacts the other graphs as well. By picking different augmentations, we influence how the embedding changes in the next episode. Doing this with all graphs causes heavy shifts of the embedding, which stops the model from learning and improving. Instead, we start by using no augmentations, only the original graphs. For every X episodes, we select the Y% of the graphs furthest away from the mean graph. For these Y% we calculate all of our 20 augmentations and pick the one that is closest to the mean graph.

With this evaluation, We want to show if we benefit from individually selecting augmentations for each graph with this strategy. A previous test in which we randomly assigned different augmentations for each graph did not improve the results. We got an average result between the performance from good and bad augmentations. As a reference, we look at our previous results where we always picked the same augmentation in Figure 7.1. We show our results for different X and Y in Table 7.11. We test different combinations of how many percent of the total graphs the model selects during each step and after how many epochs we repeat the augmentation selection. The reference values for picking the best augmentation for all graphs are bold. The best performance by picking individual augmentations is marked red.

FRANKENSTEIN performs in all tests significantly worse than the reference. All parameters showed even worse results than the baseline, even though the single augmentations mostly performed better than the baseline. COIL-DEL shows good results. We are better than most augmentations. Only the best two outperform our test. We can confirm our hypothesis for PROTEINS, NCI1, MUTAG, and IMDB-B. Our approach improved the result on all these datasets compared to the fixed augmentation. PROTEINS, NCI1, and IMDB-B show a substantial increase. MUTAG shows the best results. The mean-accuracy increases from 1.97 to 3.30 percentage points The max-accuracy increases from -0.37 to 0.11 percentage points.

We showed that the general idea of picking an augmentation individually for each graph gives good results for most datasets and often outperforms the best single augmentation result. With this approach, we improve the results and save computational resources. When we run the augmentations individually, we need to calculate an augmentation for every graph in every epoch and repeat that 20 times for each augmentation. For 100 epochs, we calculate |dataset|*100*20 augmentations. Our approach with 10% of the graphs, repeated every ten epochs, introduces an overhead to calculate all 20 augmentations of |dataset|*0.1*10*20. We need this to decide which augmentation we pick. We also need to calculate the actual augmentations for the contrastive learning in every episode of |dataset|*100*1. The total computation adds up to |dataset|*0.1*10*20 + |dataset|*100*1 = |dataset|*120. We managed to reduce our augmentation calculation by a factor of 16.

We do, however, still have multiple problems. Our approach still depends on calculating all augmentations and augmentation ratio combinations. Especially if we pick many different augmentation ratios, we still end up with many combinations that we need to test, which is computationally expensive. We have also seen in our results that there exist new parameters. We have to decide how often we calculate the best augmentation for each graph and with how many percent of the graphs we do it. Different datasets show the best results with different parameters. There is no easy way to pick good parameters other than testing.

7.10 Generators

We further research the idea of selecting individual augmentations for each graph. Our previous test relied on the fact that we use predefined augmentations and augmentation ratios. We also need to test these computationally expensive combinations. Ideally, we find a generator that does not rely on predefined augmentations and their combination. The generator takes a graph as input and outputs an augmented graph that should still contain the key features. Generators can be, for example, VGAEs or adversarial networks.

		Graphs	Repeat	PROTEINS	NCI1	MUTAG	FRANKEN.	COIL-DEL	IMDB-B
mean-acc	best aug			**2.48**	**0.37**	**1.97**	**2.57**	**23.85**	**1.62**
	individual	20%	1	2.06	0.37	3.18	-0.23	21.69	1.82
	individual	20%	5	2.35	0.15	3.30	-0.36	22.39	1.72
	individual	20%	10	2.73	0.38	1.88	-0.38	22.22	1.42
	individual	20%	20	2.65	0.59	1.37	-0.13	21.74	2.02
	individual	10%	1	2.51	0.53	1.92	-0.65	22.67	2.02
	individual	10%	5	2.62	0.27	1.02	-0.41	22.21	1.70
	individual	10%	10	2.56	0.14	2.08	-0.47	22.60	1.08
	individual	10%	20	2.94	0.30	2.13	-0.28	21.02	2.20
	individual	5%	1	2.37	0.23	-0.41	-0.09	22.60	1.76
	individual	5%	5	2.60	0.33	2.13	-0.63	22.79	1.40
	individual	5%	10	3.45	0.30	0.12	-0.58	23.16	1.24
	individual	5%	20	2.55	0.53	1.33	-0.58	22.72	1.40

		Graphs	Repeat	PROTEINS	NCI1	MUTAG	FRANKEN.	COIL-DEL	IMDB-B
max-acc	best aug			**1.19**	**0.15**	**-0.37**	**2.06**	**24.16**	**1.83**
	individual	20%	1	0.62	-0.23	0.00	-0.46	21.88	1.48
	individual	20%	5	0.89	-0.20	0.11	-0.69	22.68	1.64
	individual	20%	10	1.26	0.05	-0.09	-0.72	22.58	1.28
	individual	20%	20	1.30	0.14	-0.32	-0.38	21.94	1.98
	individual	10%	1	1.12	0.18	-0.64	-0.91	22.95	1.94
	individual	10%	5	1.35	-0.07	-1.08	-0.67	22.61	1.80
	individual	10%	10	1.66	-0.32	-0.42	-0.78	22.87	0.76
	individual	10%	20	1.17	-0.17	0.11	-0.51	21.08	1.88
	individual	5%	1	1.23	-0.21	-1.61	-0.37	22.84	1.96
	individual	5%	5	1.54	-0.16	-1.05	-0.68	23.12	1.30
	individual	5%	10	1.53	-0.09	-1.06	-0.69	23.39	1.06
	individual	5%	20	1.55	0.15	-0.53	-0.71	23.04	1.48

Table 7.11: Performance for selecting the augmentation that is closest to the mean graph. The **bold** numbers are the reference value for the single augmentation with 20% augmentation (We pick the same augmentation for all graphs).
The red numbers are the best performance when we pick the augmentation which is closest to the mean graph.

Yin et al. [13] implemented such an approach. We tested some modifications to improve their setup further. We notice that the generator takes a batch as input and decides for all nodes in the batch if they should be dropped, kept, or masked. While this is computationally efficient, we suspect that this has disadvantages. We see that for multiple graphs in each batch, all nodes get dropped, which contradicts the idea that we want to keep the key features of the graph.

We implemented their generator approach into our model to use the batch information of the generator and reconstruct single graphs. We are then able two test two things. Firstly, we use the information if nodes are dropped and implement a node-dropping method that drops exactly these nodes. Secondly, we use the chance of dropping the node that the generator calculated to implement weighted node dropping with these probabilities.

Our second approach of using weighted node dropping showed worse performances than the original generator. The probabilities of the generator do not seem to translate well into drop percentages. As expected, the hard decision of which nodes get dropped by the generator is a good approximation for node dropping. We generally do not improve the results much when we translate the generator

Dataset	PROTEINS	DD	NCI1	COLLAB	GITHUB	IMDB-B	RDT-B	RDT-5K
Our best 20% result	74.70±3.03	78.13±3.50	75.85±2.51	77.45±1.28	67.12±1.20	71.13±5.29	90.73±1.29	54.20±2.12
Generator [13]	75.65±2.40	77.50±4.41	73.75±2.25	77.16±1.48	62.46±1.51	71.90±4.79	79.80±3.47	49.91±2.70

Table 7.12: Our hand-picked best augmentation in Figure 7.1 is compared to the generator results from Yin et al. [13]. Red numbers indicate the best performance.

node-dropping to single graphs. For some of the larger social networks, we see a slight improvement. GITHUB increased from 62.46% in the paper to 62.85%. RDT-5K increased from 49.91% to 50.88%. For other datasets, we do see, on the other hand, a slight decrease.

While their general approach is promising and computationally efficient, it still has disadvantages. We compare the results from our best single performance augmentation in Table 7.2 to the results of the generator in Table 7.12. We outperform the generator in most datasets. For the PROTEINS dataset, we increase even further from 74.70% to 75.99 through the combination of augmentations and augmentations ratios as described in Section 7.3.2. Our approach comes at the cost of intensive evaluations. It is also to note that Yin et al. [13] only used 30 episodes for training. It is unclear if and how much their approach would benefit from more episodes.

8 Conclusion and Future Work

This thesis presented the problem of finding good augmentations and improving contrastive learning for semi-supervised learning. We created 15 new augmentations and evaluated them together with eight existing augmentations on 12 different datasets. Through intensive testing, we managed to show that not only the augmentation itself but the metrics that select the nodes significantly impact the performance. Generally, the best strategies are dropping high-degree nodes or building subgraphs with low-degree nodes. Additionally, we have seen that the augmentation ratio is a very important, so far neglected, parameter. While an augmentation ratio of 20% is generally considered a reasonable value, we showed that many datasets perform best with a 60-80% augmentation ratio. Especially the combination of different augmentations with high and low augmentation ratios has proven to work best in our evaluations.

We implemented additional improvements to the contrastive learning pipeline. In contrast to research in image representation learning, we showed that we do not need many negative samples in the contrastive loss function. Reducing the number of negative samples from 127 down to 2-10 did not harm the results and, in fact, slightly improved them. Additionally, we showed that the NT-Xent loss is not the only loss function we can use. A euclidian max-margin loss function showed similar performance and even outperformed the NT-Xent loss on some datasets.

Transfer learning is a common approach in the semi-supervised setting where we intend to use unlabeled data from different datasets. Usually, we do transfer learning with specific, similar datasets from the same domain. We tested the approach with multiple models that showed different behavior in earlier evaluations under the same augmentations, which suggested that transfer learning would likely not work. Transfer learning alone did indeed not work in most scenarios. We introduced an improved form, which we called warm-starting. Instead of replacing the pre-training, we added transfer learning as a pre-pre-training step before our normal pre-training. This approach has proven to improve the results on multiple datasets significantly. We showed that in this scenario, a shrink & perturb step, introduced by Ash et al. [20], further improves the results in the graph pre-training setup.

Finally, we introduced the idea of selecting individual augmentations for each graph. We presented multiple ideas and showed the first results that indicate the potential of this approach. Strategies like selecting augmentations such that all augmented graphs are similar have shown to be a good approximation to increase the difficulty of the contrastive learning problem and benefit it. While this approach is still computationally intensive, it also relies on predefined augmentations and augmentations ratios.

The next step is to use generators that get a graph as input and output the new, augmented graph. We briefly introduced existing generators, which show decent results for some datasets. We can still match the performance on most datasets and even outperform them on multiple datasets with our predefined augmentations. Generally, a generator does not yet exist that manages to solve the task entirely. So far, all generators still have individual downsides. While some work through

augmenting nodes, others focus on edges or node attributes. Future work could focus on combining the approaches of different generators to work on nodes and edges. VGAEs and adversarial networks are also areas of interest in the search for better generators.

Bibliography

[1] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009. DOI: 10.1109/TNN.2008.2005605 (cit. on p. 15).

[2] T. N. Kipf, M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016. DOI: https://doi.org/10.48550/arXiv.1609.02907. arXiv: 1609.02907. [Online]. Available: http://arxiv.org/abs/1609.02907 (cit. on pp. 15, 18, 19).

[3] K. Xu, W. Hu, J. Leskovec, S. Jegelka, "How powerful are graph neural networks?" *CoRR*, vol. abs/1810.00826, 2018. DOI: https://doi.org/10.48550/arXiv.1810.00826. arXiv: 1810.00826. [Online]. Available: http://arxiv.org/abs/1810.00826 (cit. on pp. 15, 18, 19, 23, 27).

[4] K. Jha, S. Saha, H. Singh, "Prediction of protein–protein interaction using graph neural networks," *Scientific Reports*, vol. 12, p. 8360, May 2022. DOI: 10.1038/s41598-022-12201-9 (cit. on p. 15).

[5] J. Dejun, Z. Wu, K. Hsieh, C. Guangyong, B. Liao, Z. Wang, C. Shen, D.-S. Cao, J. Wu, T. Hou, *Could Graph Neural Networks Learn Better Molecular Representation for Drug Discovery? A Comparison Study of Descriptor-based and Graph-based Models*. Sep. 2020. DOI: 10.21203/rs.3.rs-79416/v1 (cit. on p. 15).

[6] W. Jiang, J. Luo, "Graph neural network for traffic forecasting: A survey," *CoRR*, vol. abs/2101.11174, 2021. arXiv: 2101.11174. [Online]. Available: https://arxiv.org/abs/2101.11174 (cit. on p. 15).

[7] T. Chen, S. Kornblith, M. Norouzi, G. E. Hinton, "A simple framework for contrastive learning of visual representations," *CoRR*, vol. abs/2002.05709, 2020. DOI: https://doi.org/10.48550/arXiv.2002.05709. arXiv: 2002.05709. [Online]. Available: https://arxiv.org/abs/2002.05709 (cit. on pp. 15, 20, 21, 23–25, 29, 62).

[8] K. He, H. Fan, Y. Wu, S. Xie, R. B. Girshick, "Momentum contrast for unsupervised visual representation learning," *CoRR*, vol. abs/1911.05722, 2019. DOI: https://doi.org/10.48550/arXiv.1911.05722. arXiv: 1911.05722. [Online]. Available: http://arxiv.org/abs/1911.05722 (cit. on pp. 15, 23, 25).

[9] Y. You, T. Chen, Y. Sui, T. Chen, Z. Wang, Y. Shen, "Graph contrastive learning with augmentations," *CoRR*, vol. abs/2010.13902, 2020. DOI: https://doi.org/10.48550/arXiv.2010.13902. arXiv: 2010.13902. [Online]. Available: https://arxiv.org/abs/2010.13902 (cit. on pp. 15, 17, 20, 21, 25, 27, 37, 39, 40, 52, 56).

[10] Y. You, T. Chen, Y. Shen, Z. Wang, "Graph contrastive learning automated," *CoRR*, vol. abs/2106.07594, 2021. DOI: 10.48550/arXiv.2106.07594. arXiv: 2106.07594. [Online]. Available: https://arxiv.org/abs/2106.07594 (cit. on pp. 16, 25).

[11] Y. You, T. Chen, Z. Wang, Y. Shen, "Bringing your own view: Graph contrastive learning without prefabricated data augmentations," *CoRR*, vol. abs/2201.01702, 2022. DOI: 10.48550/ARXIV.2201.01702. arXiv: 2201.01702. [Online]. Available: https://arxiv.org/abs/2201.01702 (cit. on pp. 16, 25).

[12] S. Suresh, P. Li, C. Hao, J. Neville, "Adversarial graph augmentation to improve graph contrastive learning," *CoRR*, vol. abs/2106.05819, 2021. DOI: https://doi.org/10.48550/arXiv.2106.05819. arXiv: 2106.05819 (cit. on pp. 16, 26, 70).

[13] Y. Yin, Q. Wang, S. Huang, H. Xiong, X. Zhang, "Autogcl: Automated graph contrastive learning via learnable view generators," *CoRR*, vol. abs/2109.10259, 2021. DOI: https://doi.org/10.48550/arXiv.2109.10259. arXiv: 2109.10259. [Online]. Available: https://arxiv.org/abs/2109.10259 (cit. on pp. 16, 26, 27, 70, 74, 75).

[14] N. Sharma, V. Jain, A. Mishra, "An analysis of convolutional neural networks for image classification," *Procedia Computer Science*, vol. 132, pp. 377–384, 2018. DOI: 10.1016/j.procs.2018.05.198 (cit. on p. 17).

[15] W. L. Hamilton, R. Ying, J. Leskovec, "Inductive representation learning on large graphs," Jun. 7, 2017. arXiv: 1706.02216v4 [cs.SI] (cit. on p. 19).

[16] Y. Ouali, C. Hudelot, M. Tami, "An overview of deep semi-supervised learning," *CoRR*, vol. abs/2006.05278, 2020. arXiv: 2006.05278. [Online]. Available: https://arxiv.org/abs/2006.05278 (cit. on p. 19).

[17] J.-B. Grill, F. Strub, F. Altché, C. Tallec, P. H. Richemond, E. Buchatskaya, C. Doersch, B. A. Pires, Z. D. Guo, M. G. Azar, B. Piot, K. Kavukcuoglu, R. Munos, M. Valko, "Bootstrap your own latent: A new approach to self-supervised Learning," arXiv, Tech. Rep., Sep. 2020. DOI: 10.48550/arXiv.2006.07733. [Online]. Available: http://arxiv.org/abs/2006.07733 (visited on 12/05/2022) (cit. on p. 19).

[18] R. Hadsell, S. Chopra, Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, vol. 2, 2006, pp. 1735–1742. DOI: 10.1109/CVPR.2006.100 (cit. on pp. 20, 29).

[19] K. Sohn, "Improved deep metric learning with multi-class n-pair loss objective," in *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, R. Garnett, Eds., vol. 29, Curran Associates, Inc., 2016. [Online]. Available: https://proceedings.neurips.cc/paper/2016/file/6b180037abbebea991d8b1232f8a8ca9-Paper.pdf (cit. on p. 20).

[20] J. T. Ash, R. P. Adams, "On the difficulty of warm-starting neural network training," *CoRR*, vol. abs/1910.08475, 2019. arXiv: 1910.08475. [Online]. Available: http://arxiv.org/abs/1910.08475 (cit. on pp. 22, 33, 49, 65, 77).

[21] C. Sharma, "Transfer learning and its application in computer vision: A review," Mar. 2022 (cit. on p. 22).

[22] J. Devlin, M. Chang, K. Lee, K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. arXiv: 1810.04805. [Online]. Available: http://arxiv.org/abs/1810.04805 (cit. on p. 22).

[23] I. Jolliffe, J. Cadima, "Principal component analysis: A review and recent developments," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, p. 20 150 202, Apr. 2016. DOI: 10.1098/rsta.2015.0202 (cit. on p. 22).

[24] L. van der Maaten, G. Hinton, "Viualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, Nov. 2008 (cit. on p. 22).

[25] L. McInnes, J. Healy, J. Melville, *Umap: Uniform manifold approximation and projection for dimension reduction*, 2018. DOI: 10.48550/ARXIV.1802.03426. [Online]. Available: https://arxiv.org/abs/1802.03426 (cit. on p. 22).

[26] M. Zhang, Z. Cui, M. Neumann, Y. Chen, "An end-to-end deep learning architecture for graph classification," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, Apr. 2018. DOI: 10.1609/aaai.v32i1.11782. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/11782 (cit. on p. 23).

[27] D. Erhan, P.-A. Manzagol, Y. Bengio, S. Bengio, P. Vincent, "The difficulty of training deep architectures and the effect of unsupervised pre-training," in *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, D. van Dyk, M. Welling, Eds., ser. Proceedings of Machine Learning Research, vol. 5, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA: PMLR, Apr. 2009, pp. 153–160. [Online]. Available: https://proceedings.mlr.press/v5/erhan09a.html (cit. on p. 23).

[28] A. Krizhevsky, "Learning multiple layers of features from tiny images," *University of Toronto*, May 2012 (cit. on p. 23).

[29] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848 (cit. on p. 23).

[30] T. Chen, S. Kornblith, K. Swersky, M. Norouzi, G. E. Hinton, "Big self-supervised models are strong semi-supervised learners," *CoRR*, vol. abs/2006.10029, 2020. arXiv: 2006.10029. [Online]. Available: https://arxiv.org/abs/2006.10029 (cit. on p. 25).

[31] X. Chen, H. Fan, R. B. Girshick, K. He, "Improved baselines with momentum contrastive learning," *CoRR*, vol. abs/2003.04297, 2020. arXiv: 2003.04297. [Online]. Available: https://arxiv.org/abs/2003.04297 (cit. on p. 25).

[32] T. Chen, S. Bian, Y. Sun, "Are powerful graph neural nets necessary? A dissection on graph classification," *CoRR*, vol. abs/1905.04579, 2019. DOI: https://doi.org/10.48550/arXiv.1905.04579. arXiv: 1905.04579. [Online]. Available: http://arxiv.org/abs/1905.04579 (cit. on p. 27).

[33] D. P. Kingma, J. Ba, *Adam: A method for stochastic optimization*, 2014. DOI: 10.48550/ARXIV.1412.6980. [Online]. Available: https://arxiv.org/abs/1412.6980 (cit. on p. 27).

[34] S. Ioffe, C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015. arXiv: 1502.03167. [Online]. Available: http://arxiv.org/abs/1502.03167 (cit. on p. 27).

[35] A. Heidarian, M. J. Dinneen, "A hybrid geometric approach for measuring similarity level among documents and document clustering," in *2016 IEEE Second International Conference on Big Data Computing Service and Applications (BigDataService)*, 2016, pp. 142–151. DOI: 10.1109/BigDataService.2016.14 (cit. on pp. 28, 32).

[36] J. Kim. "TS-SS Python implementation." (2022), [Online]. Available: https://github.com/taki0112/Vector_Similarity (cit. on p. 32).

[37] L. Page, S. Brin, R. Motwani, T. Winograd, "The pagerank citation ranking: Bringing order to the web.," Stanford InfoLab, Technical Report 1999-66, Nov. 1999. [Online]. Available: http://ilpubs.stanford.edu:8090/422/ (cit. on pp. 37, 43).

[38] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *J. ACM*, vol. 46, no. 5, pp. 604–632, Sep. 1999, ISSN: 0004-5411. DOI: 10.1145/324133.324140. [Online]. Available: https://doi.org/10.1145/324133.324140 (cit. on pp. 37, 44).

[39] M. R. Garey, R. L. Graham, D. S. Johnson, "The Complexity of Computing Steiner Minimal Trees," *SIAM Journal on Applied Mathematics*, vol. 32, no. 4, pp. 835–859, Jun. 1977, ISSN: 0036-1399. DOI: 10.1137/0132072. [Online]. Available: https://epubs.siam.org/doi/10.1137/0132072 (visited on 11/06/2022) (cit. on p. 44).

[40] C. Morris, N. M. Kriege, F. Bause, K. Kersting, P. Mutzel, M. Neumann, "Tudataset: A collection of benchmark datasets for learning with graphs," in *ICML 2020 Workshop on Graph Representation Learning and Beyond (GRL+ 2020)*, 2020. arXiv: 2007.08663. [Online]. Available: www.graphlearning.io (cit. on pp. 49, 50).

[41] C. Morris. "TUDataset collection." (2022), [Online]. Available: https://chrsmrrs.github.io/datasets/docs/datasets/ (cit. on pp. 49, 50).

[42] K. Borgwardt, underlineCS, S. Schönauer, S. Vishwanathan, A. Smola, H. Kriegel, "Protein function prediction via graph kernels," *Bioinformatics*, vol. 21 Suppl 1, pp. i47–56, Jan. 2005 (cit. on p. 50).

[43] P. Dobson, A. Doig, "Distinguishing enzyme structures from non-enzymes without alignments," *Journal of molecular biology*, vol. 330, pp. 771–83, Aug. 2003. DOI: 10.1016/S0022-2836(03)00628-4 (cit. on p. 50).

[44] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, K. M. Borgwardt, "Weisfeiler-lehman graph kernels," *J. Mach. Learn. Res.*, vol. 12, no. null, pp. 2539–2561, Nov. 2011, ISSN: 1532-4435 (cit. on p. 50).

[45] N. Wale, G. Karypis, "Comparison of descriptor spaces for chemical compound retrieval and classification," English (US), in *Proceedings - Sixth International Conference on Data Mining, ICDM 2006*, ser. Proceedings - IEEE International Conference on Data Mining, ICDM, 2006, pp. 678–689, ISBN: 0769527019. DOI: 10.1109/ICDM.2006.39 (cit. on p. 50).

[46] A. K. Debnath, R. L. L. de Compadre, G. Debnath, A. J. Shusterman, C. Hansch, "Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity," *Journal of Medicinal Chemistry*, vol. 34, no. 2, pp. 786–797, Feb. 1991. DOI: 10.1021/jm00106a046 (cit. on p. 50).

[47] F. Orsini, P. Frasconi, L. D. Raedt, "Graph invariant kernels," in *International Joint Conference on Artificial Intelligence*, 2015 (cit. on p. 50).

[48] K. Riesen, H. Bunke, "IAM graph database repository for graph based pattern recognition and machine learning," in *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2008, pp. 287–297. DOI: 10.1007/978-3-540-89689-0_33 (cit. on p. 50).

[49] S. J. Nayar, "Columbia object image library (coil100)," 1996 (cit. on p. 50).

[50] P. Yanardag, S. V. N. Vishwanathan, "Deep graph kernels," *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015 (cit. on p. 50).

[51] B. Knyazev, G. W. Taylor, M. R. Amer, "Understanding attention and generalization in graph neural networks," in *Neural Information Processing Systems*, 2019 (cit. on p. 50).

All links were last followed on December 14, 2022.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature