École Normale Supérieure de Paris-Saclay

UNIVERSITY OF CAMBRIDGE

Data Augmentation in Graph Learning

Internship Report:

Mickael Assaraf

Under the supervision of:

Angelica Aviles-Rivero

Abstract

Graph neural networks, as powerful deep learning tools to model graph-structured data, have demonstrated remarkable performance on numerous graph learning tasks. To counter the data scarcity and the overfitting, increasing graph data augmentation research has been conducted lately. However, conventional data augmentation methods can hardly handle graph-structured data. During this internship, I studied existing data augmentation methods for graph learning, in particular on automatic methods to determine the "good" augmentations in the search space of the augmentation. I then reduced my study to the methods allowing to do contrastive graph learning; indeed contrastive learning is a type of self supervised neural network model requiring different contrastive views that can be searched in the space of the augmentations. I finally dedicated myself to an ablation study in the augmentation space in order to determine the most efficient augmentations for graph contrastive learning.

Les Graph neural networks, en tant que puissants outils d'apprentissage profond pour modéliser les données structurées en graphes, ont démontré des performances remarquables dans de nombreuses tâches d'apprentissage de graphes. Pour contrer la pénurie de données et l'excès d'ajustement, de plus en plus de recherches sur l'augmentation des données de graphes ont été menées récemment. Cependant, les méthodes conventionnelles d'augmentation des données peuvent difficilement traiter les données structurées en graphes. Au cours de ce stage, j'ai étudié les méthodes d'augmentation des données existantes pour l'apprentissage des graphes, en particulier les méthodes automatiques pour déterminer les "bonnes" augmentations dans l'espace de recherche de l'augmentation. J'ai ensuite réduit mon étude aux méthodes permettant de faire de l'apprentissage contrastif de graphes ; en effet l'apprentissage contrastif est un type de modèle de réseau de neurones auto-supervisé nécessitant différentes vues contrastives qui peuvent être recherchées dans l'espace des augmentations. Je me suis enfin consacré à une étude d'ablation dans l'espace des augmentations afin de déterminer les augmentations les plus efficaces pour l'apprentissage contrastif de graphes.

I would like to sincerely thank Angelica Aviles-Rivero for proposing me this subject, for having supervised this internship, for having always supported me, and for having helped me to make clearer all the notions that I could study. I would also like to thank all the Image analysis group of damtp for the great conferences I attended during my internship, especially the researchers or students under the supervision of Angelica with whom the weekly meetings were very enriching.

Contents

1 Definition of Graph

Before speaking of machine learning with graphs, it is necessary to give a description more formal of what is "graph data". Formally a graph $G = (V, E)$ is defined by a set of nodes V and a set of edges E between these nodes. We can also define the adjacency matrix $A \in [0, 1]^{|V| \times |V|}$ as:

$$A[i, j] = 1 \;\Leftrightarrow\; (i, j) \in E$$

We also denote $X \in \mathbb{R}^{|V| \times d}$ the node feature matrix where d is the feature dimension. Graphs defined in this way can represent many things such as

- molecule as a Graph. It is a very convenient and common abstraction to describe this 3D object as a graph, where the nodes are atoms and the edges are covalent bonds. The characteristics of the nodes can be the names of the atoms (and any other information needed to describe the molecule).

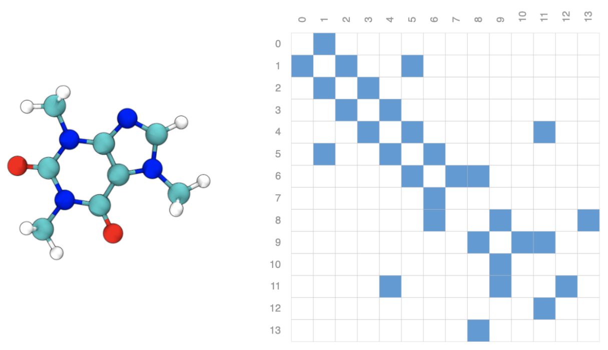


Figure 1: On the left the graphic representation of a molecule (the features are for example the nature of the atoms). On the right we can observe its adjacency matrix

- Social networks as graphs. Social networks are tools to study patterns in collective behaviour of people, institutions and organizations. We can build a graph representing groups of people by modelling individuals as nodes, and their relationships as edges.

- Citation networks as graphs. Scientists routinely cite other scientists' work when publishing papers. We can visualize these networks of citations as a graph, where each paper is a node, and each directed edge is a citation between one paper and another. Additionally, we can add information about each paper into each node, such as a word embedding of the abstract.

2 Background on Machine learning

Neural networks, also known as Artificial Neural Networks (NNs) are a subset of Machine Learning and are at the heart of Deep Learning algorithms. Their name and structure are inspired by the human.

Here is a liitle introduction of deep neural network. We can think of a deep neural network (NN) in the simplest case as a composition of parameterized (affine) linear functions $f^l(.)$, indexed by $l \in \{1, \dots, L\}$, each followed by an element-wise non-linear function $\sigma(.)$:

$$\mathbf{NN}_{\boldsymbol{\theta}} = f^L \circ \sigma \circ f^{L-1} \circ \dots \circ \sigma \circ f^2 \circ \sigma \circ f^1$$

To train NNs, we can use of backpropagation (Werbos, 1982) and variants of the stochastic gradient descent algorithm such as the Adam optimizer (Kingma and Ba, 2014) to minimize an objective function L. For exemple we can define for vector embeddings the multi-layer perceptron (MLP) as follows:

$$f^l(\mathbf{h}) = \mathbf{W}_l\mathbf{h} + \mathbf{b}_l$$

where $\mathbf{W}_l$ is the so-called weight matrix of the l-th layer and $\mathbf{b}_l$ is the bias vector. Both constitute the set of learnable parameters for a layer. We use h to denote feature vectors (or hidden representations or embeddings) in a NN. For $\sigma(h)$, we can typically use of the $ReLU(h) = max(0, h)$ activation function in MLPs.

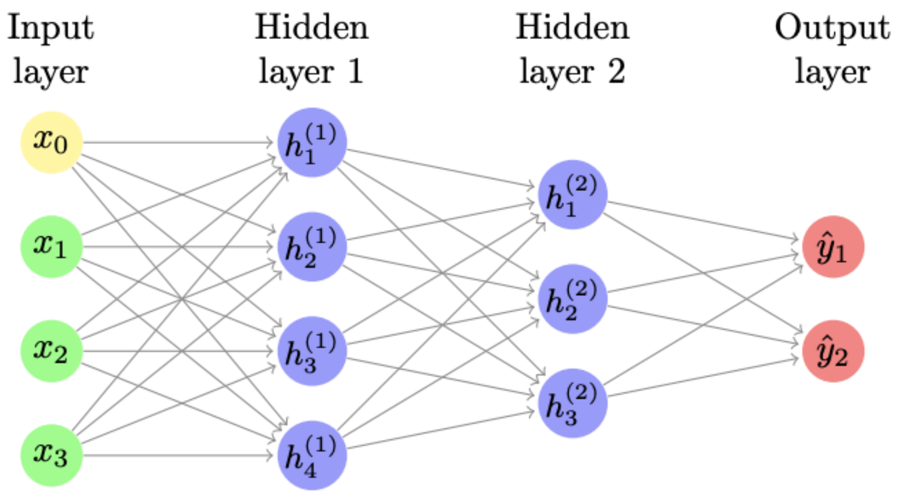


Figure 2: Shematic representation of a neural network with 2 layers (in particular here it is a MLP because each input is connected to each output of the layer)

The idea behind the structure of neural network with backpropagation was motivated by those following results:

Theorem 2.1 (Universal approximation theorem). *Let $C(X, Y)$ denote the set of continuous functions from X to Y. Let $\sigma \in C(\mathbb{R}, \mathbb{R})$. Note that $(\sigma \circ x)_i = \sigma(x_i)$, so $\sigma \circ x$ denotes σ applied to each component of x.*

Then σ is not polynomial if and only if for every $n \in \mathbb{N}, m \in \mathbb{N}$, compact $K \subseteq \mathbb{R}^n, f \in C(K, \mathbb{R}^m), \varepsilon > 0$ there exist $k \in \mathbb{N}, A \in \mathbb{R}^{k \times n}, b \in \mathbb{R}^k, C \in \mathbb{R}^{m \times k}$ such that

$$\sup_{x \in K} \|f(x) - g(x)\| < \varepsilon$$

where

$$g(x) = C \cdot (\sigma \circ (A \cdot x + b))$$

2.1 Overfitting in deep learning

In this section we will talk about a major problem when trying to do machine learning which is the overfitting phenomenon. This problem is well known when we do sampling and is even more important when we use deep neural networks. Indeed, the fact that deep neural networks have millions of parameters to optimize can result in creating models that are too complex. That is to say models that fit very well to a training data set but that are not generalizable to any data. Overfitting happens when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data.

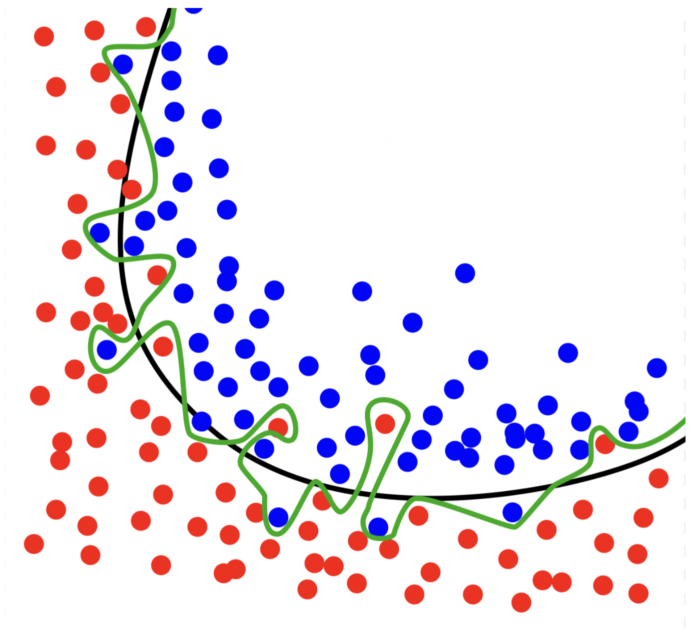


Figure 3: In this example the black curve classifies the data correctly while the green curve overfits the data.

What can we do to prevent overfitting? There are several methods to reduce the overfitting here are the main ones:

- Decrease the network complexity: By removing certain layers or decreasing the number of neurons (filters in CNN) the network becomes less prone to overfitting as the neurons contributing to overfitting are removed or deactivated. The network also has a reduced number of parameters because of which it cannot memorize all the data points & will be forced to generalize.

- Data Augmentation: One of the best strategies to avoid overfitting is to increase the size of the training dataset. As discussed, when the size of the training data is small the network tends to have greater control over the training data. But in real-world scenarios gathering of large amounts of data is a tedious & time-consuming task, hence the collection of new data is not a viable option.

- Dropouts: At each iteration different set of neurons are deactivated & this results in a different set of results. Many deep learning frameworks implement dropouts as a layer which receives inputs from the previous layer, the dropout layer randomly selects neurons which are not fired to the next layer. By deactivating certain neurons which might contribute to overfitting the performance of the network on test data improves.

2.2 Data Augmentation in Machine Learning

The performance of most ML models, and deep learning models in particular, depends on the quality, quantity and relevancy of training data. However, insufficient data is one of the most common challenges in implementing machine learning in the enterprise. This is because collecting such data can be costly and time-consuming in many cases.

For machine learning models, collecting and labeling of data can be exhausting and costly processes. Transformations in datasets by using data augmentation techniques allow to reduce these operational costs.

For example for images we can consider simple geometrical transformations like: Flipping, Rotation, Scaling Ratio, Noise injection, Translation, Cropping... However, simple transformations are not always efficient increases. Indeed, depending on what we are trying to teach the machine learning model, a class of augmentation can harm the performance. This is why current research focuses on finding automatic ways to find augmentations. For example, [Cubuk et al., 2018] describes a simple procedure called AutoAugment to automatically search for improved data augmentation policies. Specifically, AutoAugment consists of two parts: search algorithm and search space. The search algorithm is designed to find the best policy regarding highest validation accuracy. The search space contains many policies which details various augmentation operations and magnitudes with which the operations are applied.

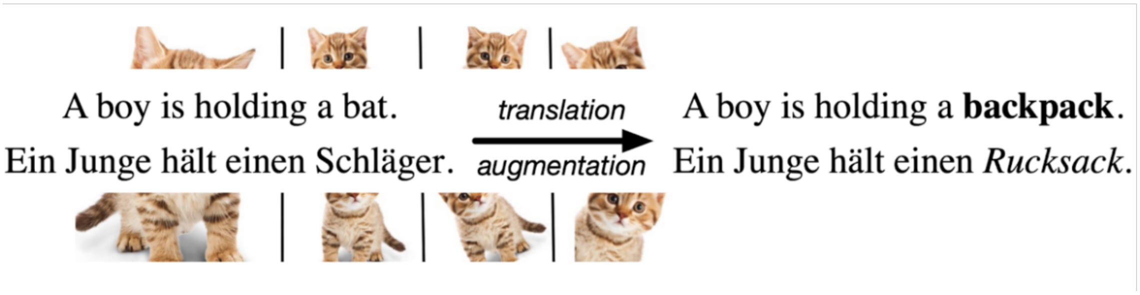

A boy is holding a bat.
Ein Junge hält einen Schläger.
translation
augmentation
A boy is holding a **backpack**.
Ein Junge hält einen *Rucksack*.

Figure 4: Some examples of data augmentation for machine learning on images and for natural language processing (NLP)

2.3 Self Supervised Contrastive learning (SSCL)

One area where data augmentation is very important is self supervised contrastive learning. In this section, we will describe what contrastive learning is.

Self supervised learning is a method to learn embeddings in an autonomous way, i.e. without the need of a label for the data. The goal is often to use these embeddings to either pre-train a neural network or to directly perform a downstream task (e.g. a classification from a linear classifier...).

The contrastive learning is based on the following principle:

Two data representing the same thing (for example having the same label) will have "close" emdedding. As well as two data representing different things (for example having the same label) will have emdedding "distant".

The basic contrastive learning framework consists of selecting a data sample, called "anchor," a data point belonging to the same distribution as the anchor, called the “positive” sample, and another data point belonging to a different distribution called the “negative” sample. The SSL model tries to minimize the distance between the anchor and positive samples, i.e., the samples belonging to the same distribution, in the latent space, and at the same time maximize the distance between the anchor and the negative samples.

So the major difference between the different methods of contrastive learning are:

- the choice of the contrastive loss: $\mathcal{L}_{contrastive}$
- the choice of positive samples: this is where the data augmentation comes into play, we can assume that the transformed data comes from the same distribution as the anchor
- the choice of negatives samples

One of the best known examples in the field for images is the SIMCLR method [Chen et al., 2020] which uses the NT-Xent loss (the normalized temperature-scaled cross entropy loss) also known as Info-NCE:

$$\mathcal{J}_{\text{InfoNCE}}\left(\boldsymbol{v}_i\right)=-\frac{1}{P}\sum_{\boldsymbol{p}_j\in\mathcal{P}(\boldsymbol{v}_i)}\log\frac{e^{\theta\left(\boldsymbol{v}_i,\boldsymbol{p}_j\right)/\tau}}{e^{\theta\left(\boldsymbol{v}_i,\boldsymbol{p}_j\right)/\tau}+\sum_{\boldsymbol{q}_j\in\mathcal{Q}(\boldsymbol{v}_i)}e^{\theta\left(\boldsymbol{v}_i,\boldsymbol{q}_j\right)/\tau}}$$

where τ is a hyperparameter to be fixed. This loss is known to be a lower bound on the mutual information up to a constant, so by optimizing this loss we maximize the mutual information between the different contrastive views.

3 Background on Graph neural network

In this section, I will introduce the graph neural network (GNN) formalism, which is a general framework for defining deep neural networks on graph data.

The key idea is that we want to generate representations of nodes that actually depend on the structure of the graph, as well as any feature information we might have.

The primary challenge in developing complex encoders for graph-structured data is that the usual deep learning toolbox does not apply. For example, convolutional neural networks (CNNs) are well-defined only over grid-structured inputs (e.g., images), while recurrent neural networks (RNNs) are well-defined only over sequences (e.g., text).
To define a deep neural network over general graphs, it is important to define a new kind of deep learning architecture. So the goal here is how we can take an input graph $G=(V,E)$, along with a set of node features $X\in\mathbb{R}^{d\times|V|}$, and use this information to generate node embeddings z_u, $\forall u\in V$. Naturally, we want the information to be transmitted from one person to another through the graph.

So, During each message-passing iteration in a GNN, a hidden embedding $\mathbf{h}_u^{(k)}$ corresponding to each node $u\in\mathcal{V}$ is updated according to information aggregated from u 's graph neighborhood $\mathcal{N}(u)$. This message-passing update can be expressed as follows:

$$\mathbf{h}_u^{(k+1)}=\text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)},\text{AGGREGATE}^{(k)}\left(\left\{\mathbf{h}_v^{(k)},\forall v\in\mathcal{N}(u)\right\}\right)\right)$$

where UPDATE and AGGREGATE are abitrary differentiable function (e.g predefined GNN). At each iteration k of the GNN, the AGGREGATE function takes as input the set of embeddings of the nodes in u 's graph neighborhood $\mathcal{N}(u)$ and generates a message based on this aggregated neighborhood information. The update function UPDATE then combines the message with the previous embedding $\mathbf{h}_u^{(k-1)}$ of node u to generate the updated embedding $\mathbf{h}_u^{(k)}$. The initial embeddings at $k=0$ are set to the input features for all the nodes, i.e., $\mathbf{h}_u^{(0)} = \mathbf{x}_u, \forall u \in \mathcal{V}$. After running K iterations of the GNN message passing, we can use the output of the final layer to define the embeddings for each node, i.e.,

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}, \forall u \in \mathcal{V}$$

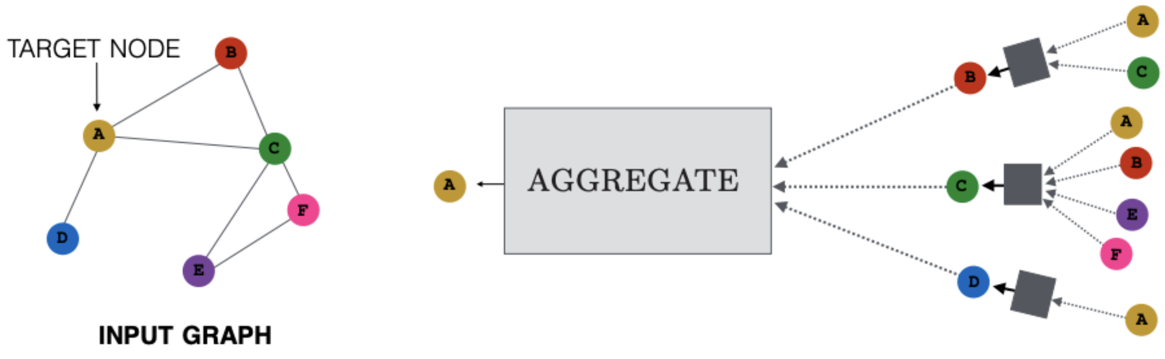


Figure 5: The AGGREGATE function is used to transmit information from neighboring nodes to the target node

3.1 The Graph Convolutional Network

For images a major idea to build neural networks is to perform convolution on the image where the kernels are parameterized by weights to be learned empirically it works very well. So work has been done to generalize convolution on graphs. Here is an introduction on convolution on graphs.

Definition 1 (Laplacian matrix)**.** *Given a simple graph G with n nodes $v_1, \ldots, v_n$, its Laplacian matrix $\mathbf{L}$ is defined element-wise as*

$$\mathbf{L}_{i,j} := \begin{cases} \deg(v_i) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases}$$

or equivalently by the matrix

$$L = D - A,$$

where D is the degree matrix and A is the adjacency matrix of the graph. Since G is a simple graph, A only contains 1 s or 0 s and its diagonal elements are all 0.

What is the link between the definition of the graph Laplacian and the Laplacian of a discrete function? For example, for a function in dimension 2, we can define the Laplacian for a sampling step h:

$$\begin{aligned}\Delta_{\text{discrete}} f(x) &= \frac{f(x+h,y)+f(x-h,y)-2f(x,y)}{h^2} + \frac{f(x,y+h)+f(x,y-h)-2f(x,y)}{h^2} \\ &= \sum_{y \in \text{neighbourhood(x)}} \frac{f(y)-f(u)}{h^2}\end{aligned}$$

Let f be a signal on a graph i.e. a function of $\mathbb{R}^{|V|}$ (for example if each node has N features we can see it as signals $f_1, \ldots, f_N$ on the graph. We then notice the analogy with the graph Laplacian because:

$$\mathbf{L} f_i(u) = - \sum_{v \in \text{neighbourhood(u)}} f_i(v) - f_i(u)$$

Theorem 3.1. **L** *is symmetric and positive semi-definite*

Proof. $\forall \mathbf{x} \in \mathbb{R}^{|\mathcal{V}|}$

$$\begin{aligned} \mathbf{x}^\top \mathbf{L} \mathbf{x} &= \frac{1}{2} \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{V}} \mathbf{A}[u, v] (\mathbf{x}[u] - \mathbf{x}[v])^2 \\ &= \sum_{(u,v) \in \mathcal{E}} (\mathbf{x}[u] - \mathbf{x}[v])^2 \geq 0 \end{aligned}$$

□

Thus we can diagonalize **L** in an orthonormal basis by ordering the eigenvalues in ascending order:

$$\mathbf{L} = \mathbf{U} \mathbf{\Lambda} \mathbf{U}^T$$

Now the eigenvectors of the Laplacian are the complex exponentials and the eigenvalues indicate the frequency. So we interpret the eigenvalues as the different frequencies of the signal.

$$-\Delta \left(e^{2\pi i s t} \right) = - \frac{\partial^2 \left(e^{2\pi i s t} \right)}{\partial t^2} = (2\pi s)^2 e^{2\pi i s t}$$

Thus, the Fourier transform of signal (or function) $\mathbf{f} \in \mathbb{R}^{|\mathcal{V}|}$ on a graph can be computed as

$$\mathbf{s} = \mathbf{U}^\top \mathbf{f}$$

and its inverse Fourier transform computed as

$$\mathbf{f} = \mathbf{U} \mathbf{s}$$

So now we can discuss about graph convolution. According to the Fourier theory we have the following property:

$$\mathcal{F}[f * g] = \mathcal{F}[f] \mathcal{F}[g]$$

The convolution on the graphs of a signal f with a filter h and defined by the term to term multiplication:

$$\mathbf{f} \star_{\mathcal{G}} \mathbf{h} = \mathbf{U} \left(\mathbf{U}^\top \mathbf{f} \odot \mathbf{U}^\top \mathbf{h} \right)$$

where **U** is the matrix of eigenvectors of the Laplacian **L** and where we have used $\star_{\mathcal{G}}$ to denote that this convolution is specific to a graph $\mathcal{G}$. However, in order to avoid diagonalizing the Laplacian, it is interesting to define vector polynomial filters of $\mathbf{\Lambda}$ in the fourier domain, $\mathcal{P}(\mathbf{\Lambda}) = (\mathcal{P}(\lambda_1), \ldots, \mathcal{P}(\lambda_n))^\top$ because we have equality:

$$\mathbf{f} \star_{\mathcal{G}} \mathbf{h} = \mathbf{U} \left(\mathcal{P}(\mathbf{\Lambda}) \odot \mathbf{U}^\top \mathbf{f} \right) = \left(\mathbf{U} \mathcal{P}(\Lambda) \mathbf{U}^\top \right) \mathbf{f} = \mathcal{P}(\mathbf{L}) \mathbf{f}$$

Moreover, this definition ensures a notion of locality. If we use a degree k polynomial, then we ensure that the filtered signal at each node depends on information in its k-hop neighborhood.

One of the most popular baseline graph neural network models—the graph convolutional network (GCN). In their seminal work, [Kipf and Welling, 2016] built off the notion of graph convolutions to define one of the most popular GNN architectures, commonly known as the graph convolutional network (GCN). The key insight of the GCN approach is that we can build powerful models by stacking very simple graph convolutional layers. A basic GCN layer is defined in [Kipf and Welling, 2016] as

$$\mathbf{H}^{(k)} = \sigma\left(\tilde{\mathbf{A}}\mathbf{H}^{(k-1)}\mathbf{W}^{(k)}\right)$$

where $\tilde{\mathbf{A}} = (\mathbf{D}+\mathbf{I})^{-\frac{1}{2}}(\mathbf{I}+\mathbf{A})(\mathbf{D}+\mathbf{I})^{-\frac{1}{2}}$ is a normalized variant of the adjacency matrix (with self-loops) and $\mathbf{W}^{(k)}$ is a learnable parameter matrix. This model was initially motivated as a combination of a simple graph convolution (based on the polynomial $\mathbf{I}+\mathbf{A}$), with a learnable weight matrix, and a non-linearity.

3.2 The Graph Isomorphism Network

In the previous section we studied the fact that convolution on graphs are very interesting filters to build neural networks. However, we will see in this section that they can be totally inefficient to distinguish very simple graphs. This has motivated the introduction of another class of neural networks (GIN) theoretically able to be better in the case of distinguishing isomorphic graphs.

Here is a theoretical introduction to this.

In an intuitive sense, two graphs being isomorphic means that they are essentially identical. Isomorphic graphs represent the exact same graph structure, but they might differ only in the ordering of the nodes in their corresponding adjacency matrices. Formally, if we have two graphs with adjacency matrices $\mathbf{A}_1$ and $\mathbf{A}_2$, as well as node features $\mathbf{X}_1$ and $\mathbf{X}_2$, we say that two graphs are isomorphic if and only if there exists a permutation matrix $\mathbf{P}$ such that

$$\mathbf{P}\mathbf{A}_1\mathbf{P}^\top = \mathbf{A}_2 \text{ and } \mathbf{P}\mathbf{X}_1 = \mathbf{X}_2.$$

Suppose we want to detect if 2 graphs are isomorphic then the problem is a simple optimization, i.e:

$$\min_{\mathbf{P}\in\mathcal{P}} \left\|\mathbf{P}\mathbf{A}_1\mathbf{P}^\top - \mathbf{A}_2\right\| + \|\mathbf{P}\mathbf{X}_1 - \mathbf{X}_2| \overset{?}{=} 0$$

with $\mathcal{P}$ the set of permutations. But we must then in a set $\mathcal{P}$ of cardinal $|V|!$ which is immense for usual graphs (more than 100 nodes)! In fact there are no algorithms that can detect the isomorphisms of graphs in polynomial time. However there is the Weisfieler-Lehman algorithm which is very efficient despite the fact that there are graphs which it is unable to find if they are isomorphic or not.

Here is the Weisfieler-Lehman algorithm:

- Let us consider two graphs G_1 and G_2 which we want to know if they are isomorphic or not. We "initialize" the algorithm by assigning labels to each node $l_{\mathcal{G}_i}^{(0)}(v)$. These labels can be for example the degree of the node or the features of the node.

- Then we assign new labels to each node by concatenating its label with those of its neighbors and applying a function HASH which assigns this family to a new label (i.e. injectively):

$$l_{\mathcal{G}_i}^{(i)}(v) = \text{HASH}\left(l_{\mathcal{G}_i}^{(i-1)}(v), \left\{\left\{l_{\mathcal{G}_i}^{(i-1)}(u)\forall u\in\mathcal{N}(v)\right\}\right\}\right)$$

- We repeat this step until the label of each node converges, that is until we reach an iteration K where $l_{\mathcal{G}_j}^{(K)}(v) = l_{\mathcal{G}_i}^{(K-1)}(v), \forall v \in V_j, j = 1, 2$
- We declare that two graphs are isomorphic if the family of iterated labels of each node is identical, that is if:
$$L_{\mathcal{G}_j} = \left\{\left\{ l_{\mathcal{G}_j}^{(i)}(v), \forall v \in \mathcal{V}_j, i = 0, \ldots, K-1 \right\}\right\}$$
is equal for both graphs $L_{\mathcal{G}_1} = L_{\mathcal{G}_2}$.

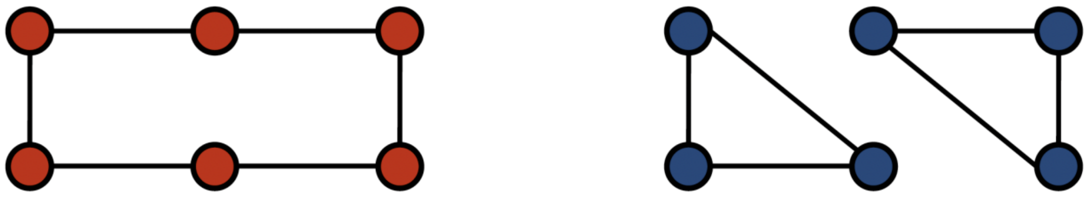


Figure 6: Example of graphs that the Weisfieler-Lehman algorithm cannot distinguish

It is very interesting to understand that there is a very deep link between the classical architecture of neural networks that we defined in section 3. Indeed we can show that no GNN with this architecture can do better than the Weisfieler-Lehman algorithm to distinguish graphs in fact we can easily understand that a neural network algorithm is efficient to distinguish graphs if and only if we have the following property:

Let z_{G_1} and z_{G_2} be the embeddings of two graphs after the neural network f then f is optimal for distinguishing graphs if:

$$G_1 \text{ is not isomorphic with } G_2 \text{ if and only if } z_{G_1} \neq z_{G_2}$$

Theorem 3.2. *Let be a GNN that consists of K message-passing layers of the following form:*

$$\mathbf{h}_u^{(k+1)} = \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, AGGREGATE^{(k)}\left(\left\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\right\}\right)\right),$$

where AGGREGATE is a differentiable permutation invariant function and UPDATE is a differentiable function. Further, suppose that we have only discrete feature inputs at the initial layer, i.e., $\mathbf{h}_u^{(0)} = \mathbf{x}_u \in \mathbb{Z}^d, \forall u \in \mathcal{V}$. *Then we have that* $\mathbf{h}_u^{(K)} \neq \mathbf{h}_v^{(K)}$ *only if the nodes u and v have different labels after K iterations of the WL algorithm.*

We can see that the major difference of these networks compared to the WL-algorithm lies in the non-injectivity of the UPDATE and AGGREGATE functions. It is then that [Xu et al., 2018] introduced the Graph Isomorphism Network (GIN) model which gives a model with few GNN parameters but as powerful as the WL algorithm.
They are defined as follows:

$$\mathbf{h}_u^{(k)} = \text{MLP}^{(k)}\left(\left(1 + \epsilon^{(k)}\right)\mathbf{h}_u^{(k-1)} + \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k-1)}\right)$$

where $\epsilon^{(k)}$ is a trainable parameter.

4 Definition of the problem

4.1 Data Augmentation for Graph deep Learning

Graph structured data is known to be much more complicated than data in images or text. Moreover, the data space is not Euclidean which makes graph augmentation methods more complicated. However, data augmentation techniques on graphs can greatly affect the performance of the models. For example, the semantics of the data when performing transformations on graphs is very dependent on the type of graph itself. Let's first describe the possible transformations that we consider on graphs in order to do data augmentation:

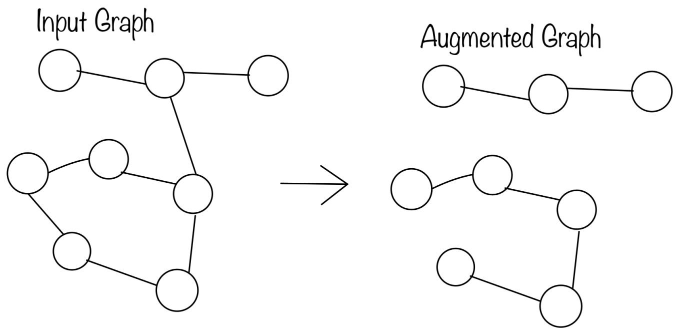
4.1.1 Structure-wise Augmentation

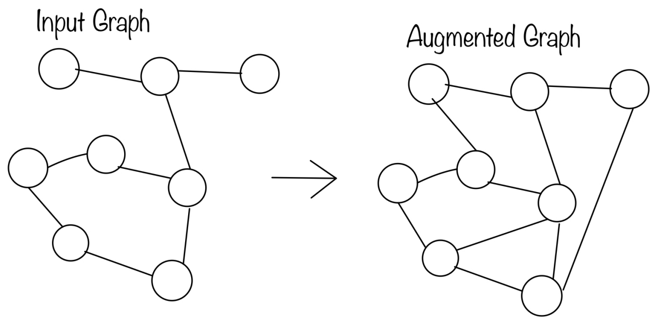
Graphs are inherently relational where the connections (i.e., edges) between data instances (i.e., nodes) are critical in understanding and analyzing graph data.

To transform a graph, we can try to change the geometry of the graph. We categorize them into: (1) edge addition/dropping, (2) node addition/dropping, (3) graph diffusion, and (4) graph sampling.

1. Edge Addition/Dropping:

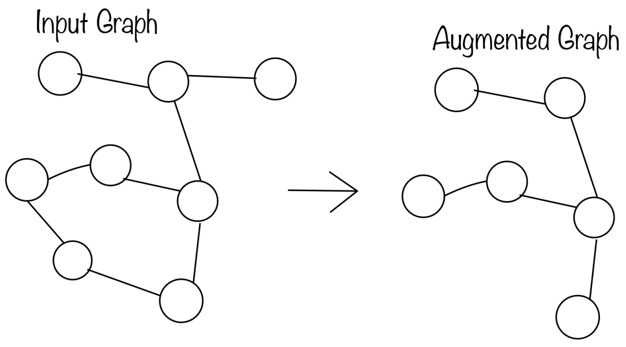
   Edge perturbation randomly adds and/or removes a portion of edges in the original graph. Formally, we sample a random masking matrix $\widetilde{\boldsymbol{R}} \in \{0,1\}^{N \times N}$, where each entry is drawn from a Bernoulli distribution $\widetilde{\boldsymbol{R}}_{ij} \sim \operatorname{Bern}(p_r)$. Here p_r is the probability for adding/removing an edge. The resulting adjacency matrix can be computed as $\widetilde{A} = \boldsymbol{A} \odot \widetilde{\boldsymbol{R}}$, where $\odot$ is a bit-wise operator.





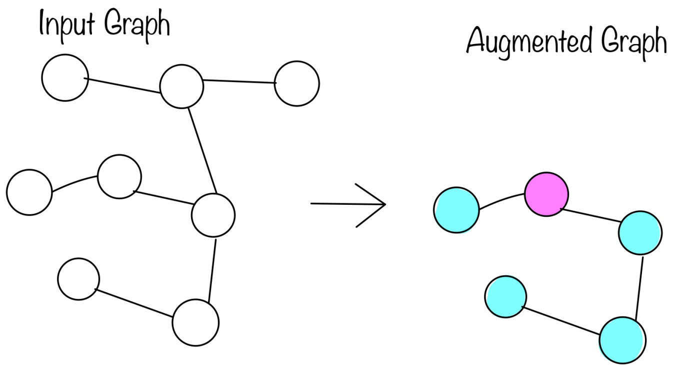
2. Node dropping

   Node dropping considers topology transformation in a node-wise manner. Similarly to Edge removing, we assign each node with a probability of p_r being dropped. Node Dropping is equivalent to masking all adjacent edges to the dropped node to zeros in the adjacency matrix.



3. Subgraph

   Subgraph sampling modifies the graph structure at the subgraph level. In this work, we are primarily concerned with Subgraphs induced by Random Walks (RWS). Starting from a node, we sample a random walk that has a probability p_{ij} to travel from node v_i to v_j and a probability p_e to return to the start node. Then, nodes appearing in this walk sequence are selected to construct a subgraph.



4. Diffusion

   As a structural augmentation strategy, graph diffusion can generate an augmented graph by providing the global views of the underlying structure. Graph diffusion injects the global topological information to the given graph adjacency by connecting nodes with their indirectly connected neighbors with calculated weights. The generalized graph diffusion can be formulated as:

$$\mathbf{S} = \sum_{k=0}^{\infty} \gamma_k \mathbf{T}^k$$

   where $\mathbf{T} \in \mathbb{R}^{N \times N}$ is the generalized transition matrix derived from the adjacency matrix $\mathbf{A}$ and θ is the weighting coefficient which determines the ratio of global-local information. Imposing $\sum_{k=0}^{\infty} \gamma_k = 1, \gamma_k \in [0, 1]$ and $\lambda_i \in [0, 1]$ where λ_i are eigenvalues of $\mathbf{T}$, guarantees convergence. Two popular examples of graph diffusion are personalized PageRank (PPR) [Page et al., 1999] (i.e., $\gamma_k = \alpha(1-\alpha)^k$) and the heat kernel [Kondor and Lafferty, 2002] (i.e., $\gamma_k = e^{-t}\frac{t^k}{k!}$). where α denotes teleport probability in a random walk and t is diffusion time.

$$\mathbf{S}^{\text{heat}} = e^{-(\mathbf{I}-\mathbf{T})t}$$
$$\mathbf{S}^{\text{PPR}} = \alpha(\mathbf{I}-(1-\alpha)\mathbf{T})^{-1}$$

4.1.2 Feature-wise Augmentation

Feature augmentation modifies to the attribute matrix $\boldsymbol{X}$. In this work, we concern the following two types of feature transformation functions.

1. Feature Masking. Feature Masking randomly masks a fraction of dimensions with zeros in node features: $\widetilde{\boldsymbol{X}}_v = [\boldsymbol{x}_1 \circ \widetilde{\boldsymbol{m}}; \boldsymbol{x}_2 \circ \widetilde{\boldsymbol{m}}; \cdots ; \boldsymbol{x}_N \circ \widetilde{\boldsymbol{m}}]^\top$, where $\circ$ is Hadamard product and $\boldsymbol{m} \in \{0,1\}^F$ is a random vector with each entry drawn from a Bernoulli distribution with a probability $(1-p_m)$

2. Feature dropout. Instead masking node entries in a column-wise manner, we compute a random matrix $\boldsymbol{M} \in \{0,1\}^{F\times|V|}$, and compute the new matrix feature $\widetilde{\boldsymbol{X}} = \boldsymbol{X} \circ \boldsymbol{M}$

3. Edge Feature Msaking. For the graph with edge features, we do the same than Feature masking but this time with the edge features

4.2 Graph Contrastive learning

In our case, we want to study the contrastive learning for graphs. Indeed, in the general case we have seen that should have a method of generating positive peers which is in the case self supervise created from a transformation of data. Here we are going to use the methods of data augmentation seen in the previous section. Now, we have seen previously that graphs can have two types of embedding, the nodes embedding for each nodes in the graph or the graph embedding which is the representation of the whole graph after a READOUT function. Thus, we can consider several modes for contrastive learning:

- Node-Node Contrastive learning: we do contrastive learning by contrasting the nodes embeddings. The anchor is an embedding of a node and the positive pairs are the embeddings of the node after transformation. The negative peers are all the other remaining nodes of the graph

- Graph-Graph Contrative learning: we do contrastive learning by contrasting the graph embeddings. We consider for the positive samples the transformed graphs and for the negative samples all the other graphs in the batch

- Graph-Node Contrastive learning: From an embedding of a graph which is the anchor. We take for positive sample all the embedding of the nodes of the augmented graph (so we need a function that computes the similarity between the representation of the graph and a node. The negative samples are the nodes of the other graphs of the batch.

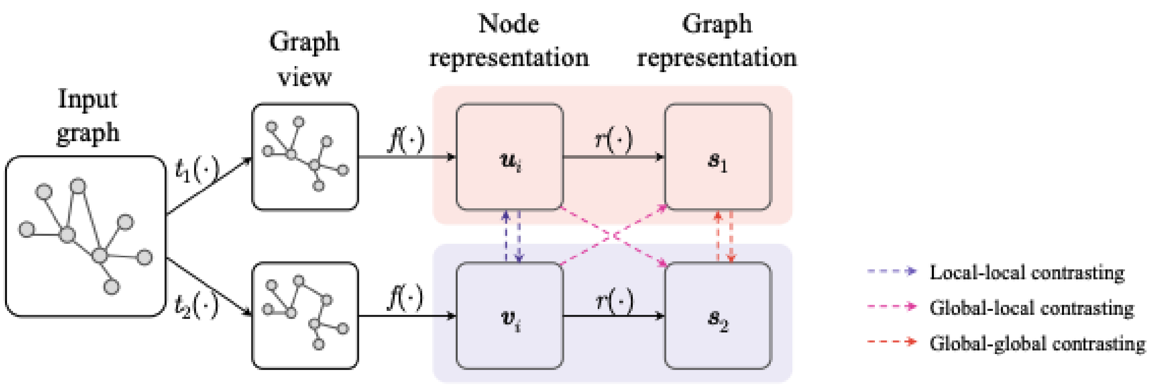


Figure 7: Schema showing the different modes of the contrastive learning graph

4.3 Question

However, there are several questions about data augmentation. Indeed, most of the current methods using graph contrastive learning use different methods to create sample positives.

For example, DGI [Veličković et al., 2018] adopts feature corruption where it conducts a row-wise shuffling on the raw node feature matrix X, ainsi cet methode utilise une view corrompu afin d'obtenir les negatif sample. The feature corruption by DGI can also be viewed as randomly swapping the nodes in the graph.

GraphCL [You et al., 2020] and InfoGCL [Xu et al., 2021] adopt four data augmentation operations: node dropping that randomly removes nodes along with its edges, edge perturbation, attribute masking that randomly masks off certain node attributes, and subgraph. Similar to the subgraph sampling operation used in GraphCL, SUBGCON [Jiao et al., 2020] utilizes a subgraph sampler to sample the augmented subgraph. GRACE [Zhu et al., 2020a] uses only the basic random edge dropping and attribute masking for creating different views of the graph.

Some methods use Graph diffusion. As an efficient GDA operation that can naturally creates a "future view" of the given graph. MVGRL [Hassani and Ahmadi, 2020] adopts the diffusion graph proposed by GDC [Klicpera et al., 2019] as the second view.

Automated GDA As aforementioned, most contrastive learning methods adopt a combination of several simple augmentation operations. The selection among the operations and their magnitudes significantly increases the number of hyperparameters. Therefore, automated solutions that can learn the augmentation strategies are developed.

JOAO [You et al., 2021] models the selection of GDA for GraphCL [You et al., 2020] as a bilevel optimization problem, where the outer level learns the augmentation strategy and the inner level learns graph representations with the given augmentations. LG2AR [Hassani and Khasahmadi, 2022] learns a probabilistic policy that contains a set of distributions over different augmentation operations, and samples augmentation strategy from the policy in each training epoch. GCA [Zhu et al., 2020b] designs adaptive augmentations based on the node centrality measures. Unlike the above-mentioned that finds the best augmentation strategy for the dataset, the adaptive augmentation of GCA gives different augmentation to nodes according to their importance.

Automatic data augmentation for graphs is the subject on which I started my internship, but due to the complexity of some methods, and the time I didn't manage to make a personal contribution to the current methods. I then focused on trying to understand each of the augmentations and their contribution in the case of fixing the augmentations beforehand. Here are the questions I tried to answer:

- What types of augmentations are most effective?
- Does adding several augmentations in a row affect performance? What is the optimal number of augmentations?

5 Experiment

To answer these questions we do an ablation study that is to say that we will fix all the hyperparameters of our network, the contrastive mode, the type of neural network, the contrastive loss and modified only what we are interested in that is to say the creation of contrastive views.

The settings we used is detailed here:

Neural network	contrastive mode	contrastive loss
GCN with 2 layers and 32 hidden dimensions	local-local contrastive learning	Loss info NCE

downstream task	classifier	evaluation metric
unsupervised node classification	ℓ_2-regularized logistic regression classifier	f1 score

optimizer	number of epoch	dataset
Adam	500	Cora

To understand how the augmentations improve the performances we first perform tests with only one augmentation in each branch, but we have 7 (8 counting the identity) possible augmentations which makes 42 different tests, we distinguish several families in these augmentations:

- in a branch an augmentation and in the other the identity
- the same augmentations in the two branches
- different augmentations in the two branches

Indeed I grouped these types of augmentations because I expect to have different results for these three families, is it better to make different augmentations which could improve the generalization but perhaps deteriorate the results or to make similar augmentations?

In a second step, I tried to multiply the number of augmentations in a row to see how the number of augmentations impact the performances. We expect the performance to augmentation with the number of augmentations and then decrease.

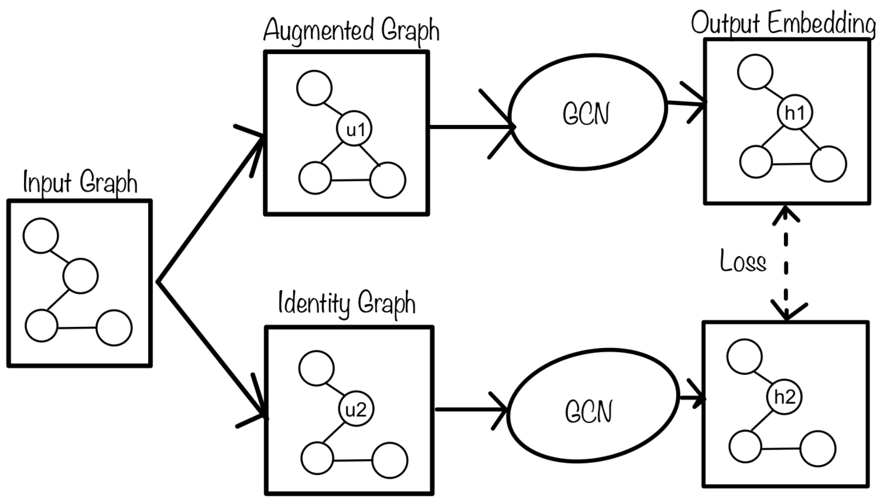


Figure 8: Schema of our experience in the case of a single augmentation on one branch and keeping the identity on the other branch

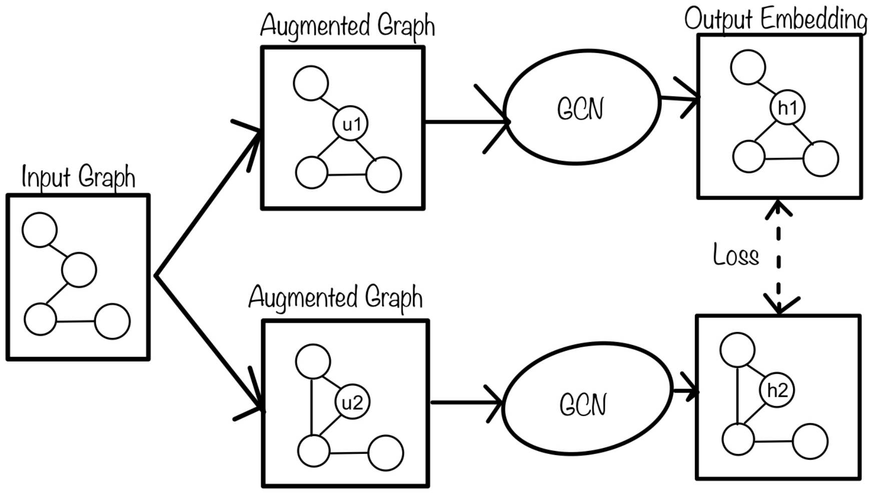


Figure 9: Schema of our experiment in the case of a single augmentation on both branches (here EdgeRemove on both branches)

5.1 Pytorch with PyGCL

In this section I will make a brief summary of the library used to code the experiments.

PyTorch is an open source Python machine learning library based on Torch developed by Meta. PyTorch allows to perform tensor calculations necessary for deep learning. In addition to being able to perform calculations on large databases, PyTorch has its own implementation of the automatic differentiation algorithm, gradient descent, which makes it a very efficient tool for our problem.

Moreover, in order to do graph contrastive learning I used PyGCL. PyGCL is an open-source Graph Contrastive Learning (GCL) library for PyTorch, which features modularized GCL components from published papers, standardized evaluation, and

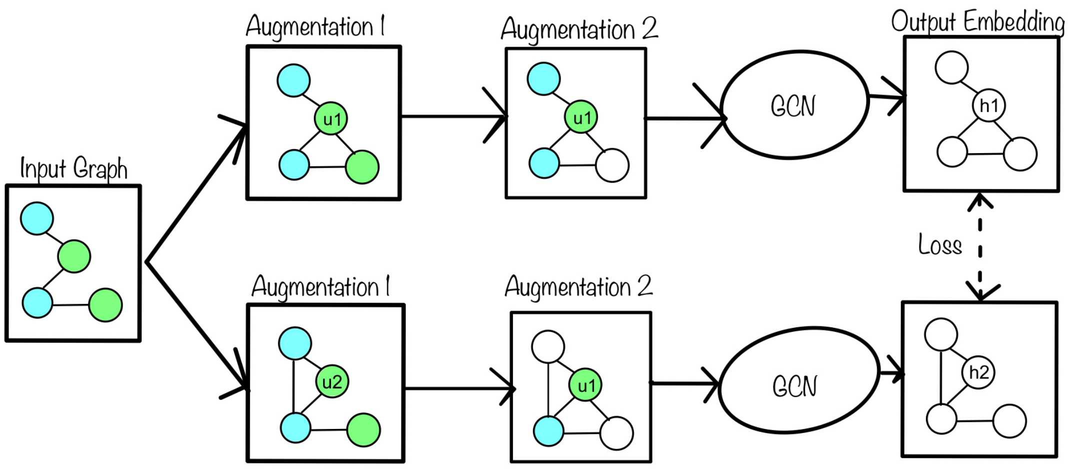


Figure 10: Schema of our experience in the case of two augmentations on the two branches (here EdgeRemove then FeatureDropout on the two branches)

experiment management. It implements the main components of graph contrastive learning algorithms: Graph augmentation (transforms input graphs into congruent graph views), Contrasting architectures and modes (generate positive and negative pairs according to node and graph embeddings), Contrastive objectives (computes the likelihood score for positive and negative pairs), and negative mining strategies (not used in our study).

5.2 Dataset

In this section I will present the dataset used. One of the most commonly used datasets is the Cora dataset which consists of 2708 scientific publications classified in one of seven classes. The citation network is composed of 5429 links. Each publication in the dataset is described by a word vector of value 0/1 indicating the absence/presence of the corresponding word in the dictionary. The dictionary is composed of 1433 unique words. It is therefore a small dataset that contains only 1 graph. The nodes are classified into one of seven topics, so the goal in doing node classification is to predict the topic of each publication.

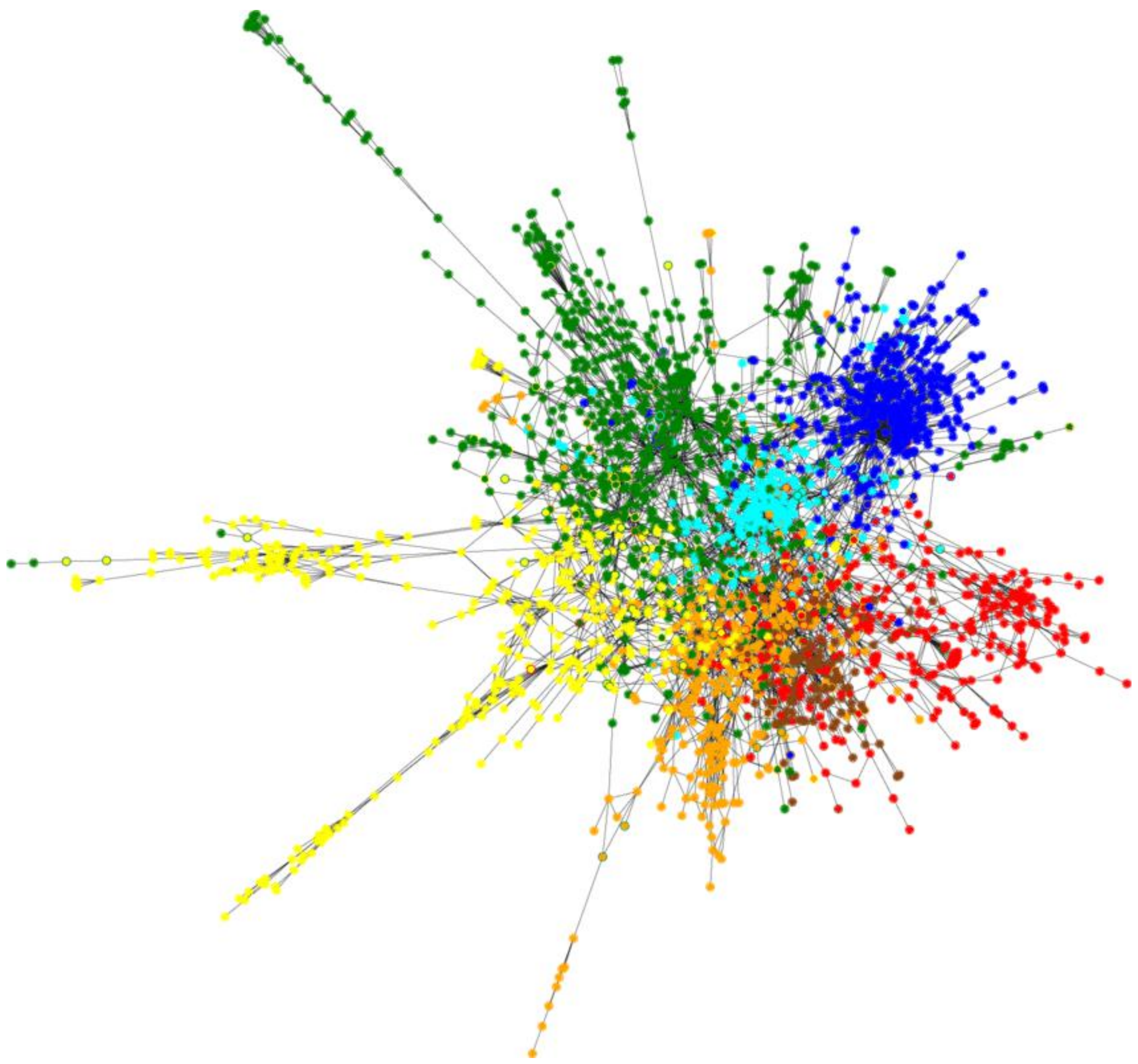


Figure 11: Graphical scheme of the Cora dataset, the colors represent the labels (subjects)

5.3 Results

In this section I will present the results obtained. First, we will run our program for a single augmentation in each branch.

Observation 1. For this dataset, for all parameters Feature Dropout is the best augmentation

Table 1 shows that for one augmentation Feature Dropout outperforms every other augmentation. Table 2 shows that for two augmentations in a row we have better performance with Feature Dropout.

Observation 2. In the case of a single augmentation, except for FeatureDropout, it is better to perform augmentations of the same nature on both branches

Table 1 shows that except for Edge FeatureMasking, it is better to make augmentations of the same type than only one augmentation on one branch and the identity on the other. Similarly, Table 1 shows that except for Feature Dropout which outperforms all others, it is better to have augmentations of the same type on both branches (for example EdgeRemove with NodeDropping or EdgeAttrMasking with EdgeFeatureMasking).

Thus, in view of observation 2, for the other experiments we will perform the same augmentations on the 2 branches. In a second step, we can analyze the performances for 2 augmentations in a chain (same augmentation on the 2 branches).

Observation 3: For 2 and 3 augmentations in a row, the performance seems to increase more and more. From 4 augmentations the performance seems to decrease

Table 2 gives us the results for 2 augmentations in a row (here we only have values greater than 0.7), but the average over the whole table is 0.71, similarly for 3 augmentations the average over the whole table is 0.74, while for 1 and 4 augmentations we have an average f1 score of 0.69. Thus increasing the number of augmentations is effective in strengthening generalization. However, this has its limits when we multiply too many increases in a row.

Observation 4: It seems interesting to compose augmentation at both structure and attribute level
Table 3 and Table 4 show us that when we compose NodeDropping augmentations with FeatureDropout, we get the maximum performance. This shows that both topology and structures are important for learning graph representations.

Augmentation 1	**Augmentation 2**	**f1 score**
EdgeRemove	Identity	0,62
EdgeAttrMasking	Identity	0,59
NodeDropping	Identity	0,68
RawSampling	Identity	0,68
PPR	Identity	0,64
EdgeFeatureMasking	Identity	0,70
FeatureDropout	Identity	0,81
EdgeRemove	EdgeRemove	0,69
EdgeAttrMasking	EdgeAttrMasking	0,64
NodeDropping	NodeDropping	0,70
RawSampling	RawSampling	0,68
PPR	PPR	0,64
EdgeFeatureMasking	EdgeFeatureMasking	0,63
FeatureDropout	FeatureDropout	0,84
EdgeRemove	NodeDropping	0,76
EdgeAttrMasking	PPR	0,70
EdgeAttrMasking	EdgeFeatureMasking	0,71
RawSampling	PPR	0,74
FeatureDropout	EdgeRemove	0,81
FeatureDropout	EdgeAttrMasking	0,80
FeatureDropout	NodeDropping	0,81
FeatureDropout	RawSampling	0,79
FeatureDropout	PPR	0,82
FeatureDropout	EdgeFeatureMasking	0,83

Table 1: Performance tables for a single augmentation per branch. For the last part of the table I only put the augmentations where the f1 score was greater than 0.7. In green is indicated the maximums by categories.

Augmentation 1	**Augmentation 2**	**f1 score**
EdgeRemove	RawSampling	0,75
EdgeRemove	EdgeFeatureMasking	0,72
EdgeAttrMasking	EdgeRemove	0,70
NodeDropping	EdgeRemove	0,74
NodeDropping	EdgeAttrMasking	0,72
NodeDropping	EdgeFeatureMasking	0,70
FeatureDropout	EdgeRemove	0,78
FeatureDropout	EdgeAttrMasking	0,76
FeatureDropout	NodeDropping	0,83
FeatureDropout	RawSampling	0,82
FeatureDropout	PPR	0.79
FeatureDropout	EdgeFeatureMasking	0,80

Table 2: Performance table for two augmentations in a row on both branches (the same types for each branch). I only put the augmentations where the f1 score was greater than 0.7. In green is indicated the maximum.

Augmentation 1	**Augmentation 2**	**Augmentation 3**	**f1 score**
EdgeAttrMasking	NodeDropping	EdgeFeatureMasking	0,70
EdgeAttrMasking	NodeDropping	FeatureDropout	0,81
EdgeAttrMasking	RawSampling	FeatureDropout	0,81
EdgeAttrMasking	RawSampling	EdgeRemove	0,71
EdgeAttrMasking	PPR	FeatureDropout	0,80
EdgeAttrMasking	EdgeFeatureMasking	FeatureDropout	0,81
NodeDropping	RawSampling	EdgeFeatureMasking	0,71
NodeDropping	RawSampling	FeatureDropout	0,79
NodeDropping	RawSampling	EdgeRemove	0,71
NodeDropping	PPR	FeatureDropout	0,78
NodeDropping	EdgeFeatureMasking	FeatureDropout	0,81
NodeDropping	FeatureDropout	EdgeRemove	0,84
RawSampling	PPR	FeatureDropout	0,75
RawSampling	PPR	EdgeRemove	0,75
RawSampling	EdgeFeatureMasking	FeatureDropout	0,80
RawSampling	FeatureDropout	EdgeRemove	0,76
PPR	FeatureDropout	EdgeRemove	0,82
EdgeFeatureMasking	FeatureDropout	EdgeRemove	0,79

Table 3: Performance table for three augmentations in a row on both branches (the same types for each branch). I only put the augmentations where the f1 score was greater than 0.7. In green is indicated the maximum.

6 Conclusion

In this internship report, I first presented the basics of graph deep learning, where I studied two very popular neural networks architecture. I then explained the interest of data augmentation for graphs which is nowadays a very popular research topic. Then I presented a taxonomy for contrastive graph learning. Finally, I analyzed the design choice for the data augmentation part of Graph contrastive learning. While Graph contrastive learning has already demonstrated strong empirical performance across a variety of downstream tasks, it is still in its infancy with many challenges left widely open. However some limitations to my work need to be acknowledged.

- Lack of theoretical justification. Our work only presents empirical studies which has thrown up many questions in need of further theoretical justification for better understanding the underlying mechanisms of GCL

- Limited downstream task. My empirical study only includes experiments on node classification. A broader range of downstream tasks of different granularities, e.g., graph classification and link prediction by graph regression and community detection, might be useful for drawing more convincing conclusions.

- Limited number of datasets. Due to computer problems and lack of time I have only studied the results on one dataset which is obviously not enough to draw convincing conclusions

References

[Chen et al., 2020] Chen, T., Kornblith, S., Norouzi, M., and Hinton, G. E. (2020). A simple framework for contrastive learning of visual representations. *CoRR*, abs/2002.05709.

[Cubuk et al., 2018] Cubuk, E. D., Zoph, B., Mané, D., Vasudevan, V., and Le, Q. V. (2018). Autoaugment: Learning augmentation policies from data. *CoRR*, abs/1805.09501.

[Hassani and Ahmadi, 2020] Hassani, K. and Ahmadi, A. H. K. (2020). Contrastive multi-view representation learning on graphs. *CoRR*, abs/2006.05582.

[Hassani and Khasahmadi, 2022] Hassani, K. and Khasahmadi, A. H. (2022). Learning graph augmentations to learn graph representations.

[Jiao et al., 2020] Jiao, Y., Xiong, Y., Zhang, J., Zhang, Y., Zhang, T., and Zhu, Y. (2020). Sub-graph contrast for scalable self-supervised graph representation learning. In *2020 IEEE International Conference on Data Mining (ICDM)*, pages 222–231.

[Kipf and Welling, 2016] Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907.

[Klicpera et al., 2019] Klicpera, J., Weißenberger, S., and Günnemann, S. (2019). Diffusion improves graph learning. *CoRR*, abs/1911.05485.

[Kondor and Lafferty, 2002] Kondor, R. and Lafferty, J. (2002). Diffusion kernels on graphs and other discrete input spaces. *ICML*, Vol. 2.

[Page et al., 1999] Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab. Previous number = SIDL-WP-1999-0120.

[Veličković et al., 2018] Veličković, P., Fedus, W., Hamilton, W. L., Liò, P., Bengio, Y., and Hjelm, R. D. (2018). Deep graph infomax.

[Xu et al., 2021] Xu, D., Cheng, W., Luo, D., Chen, H., and Zhang, X. (2021). Infogcl: Information-aware graph contrastive learning. *CoRR*, abs/2110.15438.

[Xu et al., 2018] Xu, K., Hu, W., Leskovec, J., and Jegelka, S. (2018). How powerful are graph neural networks? *CoRR*, abs/1810.00826.

[You et al., 2021] You, Y., Chen, T., Shen, Y., and Wang, Z. (2021). Graph contrastive learning automated. *CoRR*, abs/2106.07594.

[You et al., 2020] You, Y., Chen, T., Sui, Y., Chen, T., Wang, Z., and Shen, Y. (2020). Graph contrastive learning with augmentations.

[Zhu et al., 2020a] Zhu, Y., Xu, Y., Yu, F., Liu, Q., Wu, S., and Wang, L. (2020a). Deep graph contrastive representation learning. *CoRR*, abs/2006.04131.

[Zhu et al., 2020b] Zhu, Y., Xu, Y., Yu, F., Liu, Q., Wu, S., and Wang, L. (2020b). Graph contrastive learning with adaptive augmentation. *CoRR*, abs/2010.14945.