Maastricht University

Faculteit Wetenschappen

School voor Informatietechnologie

master in de informatica

Masterthesis

Deep Learning on Compressed Graphs

Jeroen Bollen

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Stijn VANSUMMEREN

Prof. dr. Jan VAN DEN BUSSCHE

2021
2022

UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie
master in de informatica

Masterthesis

Deep Learning on Compressed Graphs

Jeroen Bollen
Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :
Prof. dr. Stijn VANSUMMEREN
Prof. dr. Jan VAN DEN BUSSCHE

Contents

Chapter 1

Introduction

Deep learning and artificial intelligence are hot topics in this day and age, and many different neural networks exist that can tackle the task of making predictions based on various kinds of data. Convolutional neural networks have proven useful when dealing with data in the form of images, and recurrent neural networks or even more recently transformer neural networks have proven their worth when it comes to dealing with text input.

1.1 Neural networks and deep learning

Deep learning is a branch of machine learning that uses mutli-layer neural networks to learn representations of data suited for a particular task [LBH15]. The model can be represented as a function that takes as input a data point, and outputs a prediction. To train the model, we provide an example function, which has a limited but representative domain, which the model will learn to mimic.

The nature of the output of the model depends on the problem the model is trained to solve. Classification tasks will train a model to generate, for every data point, a probability distribution over a limited set of classes. For example, an image classification task can be used to make a model that distinguishes between photos of cats and photos of dogs.

Classification tasks are what we will focus on in this thesis, but many other types of tasks exists. Think about text translation or face recognition, which are objectives that cannot be thought of as classification tasks.

Various types of neural networks can be used within the deep learning framework. Convolutional neural networks have proven useful when dealing with data in the form of images, and recurrent neural networks or even more recently transformer neural networks have proven their worth when it comes to dealing with text input. In this thesis we primarily focus on graph neural networks.

1.2 Graph neural networks

The category of data we are most concerned with in this thesis is graph data. Many real-world objects have natural representations as graphs. Molecules or even proteins are not simply atoms, but also the bonds between those atoms. A user on a social medium is not merely their avatar and their biography, but can also be represented by their friendships and connections. This very thesis can be represented not just by its words and text, but also by the literature it cites. The structure within these graphs contains critical information about the objects they represent.

If we task a neural network with predicting the category an academic paper belongs to based on its title and abstract, we can expect the accuracy to be reasonably good. In fact, Hu et al. have trained a traditional neural network to do exactly this, and managed to get an accuracy of 61% over 40 categories [Hu+20]. It is not unreasonable to expect however that if the neural network were somehow able to not just take into account the text-content of the academic papers, but also the citation graph of how different papers relate to each other, its accuracy could improve. After all, one might expect that a group of papers that cites each other probably discusses a similar topic, and thus might very well belong in the same category.

This is where graph neural networks come in. These are neural networks that are able to take into account the structural information of a graph, and use it while making predictions. In fact, Hu et al. have used graph neural networks to tackle the same classification task on academic papers, and were able to improve the accuracy of their predictions to 73%. Others have improved on this further, with Chien et al. increasing accuracy to 76% [Chi+21].

1.3 Compressing graphs

Graph neural networks have one major downside however: they tend to use a lot more resources when it comes to computational power and memory. This is a problem especially during training, which typically iteratively evaluates the model millions of times. While inconvenient, this is not unexpected. Models operating on graph data are able to use a lot more information than other models, and thus require more resources to be able to process all this extra data.

Graph neural networks are limited in the extent to which they can use the structural data of a graph. When making a prediction for a node, they limit themselves to the structural information in that node's neighbourhood. On top of that, they are not able to distinguish every kind of neighbourhood.

In this thesis we investigate whether we can abuse this principle to 'compress' a graph into a smaller graph, which when used during training would have a minimal impact or even no impact at all on the performance of the model, while by reducing the size of the graph, greatly speeding up the training process.

1.4 Contributions and outline

In Chapter 2 we will provide some definitions and notations that will be used in the rest of the thesis, especially those for graphs and classifiers. In Chapter 3 we will provide some background into deep learning on graphs, and graph neural networks. Especially in Chapter 3.4 we will provide an overview of how expressive graph neural networks are, and what similarities they have to the Weisfeiler-Lehman isomorphism test.

In Chapter 4 we will present a graph compression algorithm that tries to exploit the expressiveness of graph neural networks, to create a graph that is smaller than the original graph, while still containing to the eye of a graph neural network, the same information. This algorithm is the main contribution of this thesis. In Chapter 5 we present the methodology for our experiments, to test our graph compression algorithm by training graph neural networks on these compressed graphs, and in Chapter 6 we present the results of these experiments.

Finally, in Chapter 7 we will conclude the thesis, draw our final conclusions, and discuss possible future directions to navigate in.

Chapter 2

Definitions and notations

This chapter briefly defines fundamental concepts used in this thesis.

2.1 Graphs

We define a graph G as a tuple $G = (V, E, L)$ where:

- V is a set of nodes,
- $E \subseteq V^2$ is a multiset of directional edges,
- Σ is a set of labels, and
- $L : V \to \Sigma$ is a function mapping nodes to labels.

We also define the following notations to aid in describing the nodes and edges in a graph:

- $v \in G$ denotes that a node v exists in the graph G,
- $(v_1 \to v_2) \in G$ denotes that an edge from node v_1 to node v_2 exists in the graph G,
- $\mathrm{inv}(v) = \{\!\{v_1 | (v_1 \to v) \in E\}\!\}$ is the multiset of incoming vertices for v, and
- $\mathrm{indeg}(v) = |\,\mathrm{inv}(v)|$ is the number of incoming edges to v.

As follows from this definition, all graphs in this thesis will be directed graphs. An undirected graph can be represented simply as a graph where for every edge, there is an edge going in the opposite direction, that is, the following property holds.

$$\forall v_1, v_2 \in G : (v_1 \to v_2) \in G \implies (v_2 \to v_1) \in G$$

It is also allowed for a node to have one or multiple edges to itself.

Furthermore, since the collection of edges is a multiset, it is possible for the same edge to occur many times within the same graph. To construct a multiset, we use a similar notation as typical for constructing a set, but use double brackets instead of single brackets. For instance, the multiset containing all possible unique edges in a graph twice can be denoted as $\{\!\{e | e \in V^2\}\!\} \cup \{\!\{e | e \in V^2\}\!\}$.

The multiplicity of a multiset is the maximum number of times an element appears in it. We use the notation $E|_{\leq c}$ to restrict the multiplicity of a multiset E to c. For instance, the multiset $\{\!\{1, 2, 2, 3, 3, 3\}\!\}|_{\leq 2}$ is equal to the multiset $\{\!\{1, 2, 2, 3, 3\}\!\}$.

2.2 Classifiers

As we are within the domain of deep learning, it is typical for our graph to be split up between a train set, a validation set, and a test set. Since we are dealing with classification tasks, we will define a classifier as a function that takes a node, and returns a probability distribution over classes. Given a graph $G = (V, E, L)$, we define a classifier D as a function $D : \mathrm{dom}(V) \rightarrow \mathbb{Q}^d$ where $\mathrm{dom}(D) \subset V$ is the set of nodes on which our classifier is defined, d is the number of classes, and $\mathbb{Q}^d$ is a probability distribution over classes. The classes of the nodes of the datasets we work with in this thesis are known, so the probability distribution is simply a vector with 1 for the class of the node, and 0 for the other classes.

Typically, a graph will be accompanied by three classifiers, D_{train}, D_{val}, and D_{test}, to be used for training, validation and testing respectively.

Chapter 3

Graph learning

In this chapter we focus on existing literature on graph learning. We discuss techniques without graph neural networks as well as graph neural networks themselves. Much of the information found in this chapter can be found in “Graph representation learning” by Hamilton [Ham20].

3.1 Graph statistics

We can measure many statistics within a graph, some of which relate to nodes, some of which relate to edges, and some of which relate to an entire graph. These statistics can be used as features for machine learning tasks.

3.1.1 Node-level statistics

Node level statistics are statistics that try to measure or describe certain properties of nodes within a graph. These are scalar values that can be calculated for, and assigned to, every node within a graph.

The **node centrality** is a category of node statistics, which tries to describe for every node broadly how central it is within a graph. Below four examples are given.

- The **degree centrality** of a node is equal to its degree, that is the amount of edges connected to it. A node is more central if it has a lot of neighbours.
- The **closeness centrality** of a node is the reciprocal of the sum of the distance to every other node. This is usually normalised by dividing it by the amount of nodes in the graph. A node is more central if a lot of other nodes can be reached quickly through it.
- The **betweenness centrality** of a node is the likelihood that when taking the shortest path between any two nodes, the node to be measured is in that path. A node is more central if it is an important connection hub in the graph.
- The **eigenvector centrality** of a node is the likelihood that a random walk through the graph ends up at the node to be measured. A node is more central if it is more likely to randomly end up there.

Simply because a node has a high centrality according to one measure does not mean this will be the case for all measures. A node can have a higher degree-centrality while still having a low closeness centrality.

Furthermore, centrality isn’t the only kind of node-level statistic we can measure. For example, the **clustering coefficient** of a node is the proportion of edges between its neighbours over the possible amount of edges between its neighbours.

3.1.2 Edge-level statistics

Just like we can measure statistics for individual nodes, we can measure statistics for edges. An edge does not have to exist for an edge-level statistic to be calculated. They can typically be evaluated for every two nodes in a graph. This could be useful for tasks like edge-prediction, where we try to complete an incomplete graph by adding edges.

Neighbourhood overlap measures how many neighbours two nodes have in common. This metric will generally be higher for edges between nodes with a high degree. There are three main techniques to normalise the overlap measure.

- The **Sorensen-Dice coefficient** divides the overlap by the sum of the degrees of the two nodes.
- The **cosine similarity** tries to achieve the same by dividing by the square root of the product of the degrees of the two nodes.
- Finally, the **Jaccard similarity** divides the overlap by the amount of neighbours the two nodes have in common.

Even when normalised, nodes with a high degree will still be more similar to other nodes than nodes with a lower degree. There are also two main ways to address this bias.

- The **resource allocation index** measures, for every shared node, the two nodes being measured have the reciprocal of the degree of that shared node. This gives a bias towards shared neighbours that have a low degree, and are thus less likely to be shared.
- The **Academic/Adar index** works the same as the resource allocation index, but takes the logarithm of the degree of the shared node.

Instead of merely taking into account the direct neighbourhood of two nodes to measure their similarity, it is also possible to take into account the entire graph.

The **Katz index** counts the amount of walks that exist between a pair of nodes. We can bias the Katz index in favour of shorter paths by adding a constant $0 < \beta \leq 1$, and calculating the similarity as $kats(x, y) = \sum_{i=0}^{+\infty} \beta^i A^i[x, y]$. The weight β^i will be smaller for longer paths.

Similar to this is the Personalised Pagerank algorithm, which for every two nodes calculates the chance that starting at one, after an infinite-length random walk we end up at the other.

3.1.3 Graph-level statistics

Finally, statistics can also be measured not just for components of the graph, but for the entire graph itself. This allows to compare multiple graphs with each other.

The foremost way to calculate graph-level statistics is using graph kernels. These are functions that, when given two objects, can quantify their similarity. The most well known of this is the Weisfeiler-Lehman graph kernel, based on the graph isomorphism test that goes by the same name [She+11]. We will describe both the isomorphism test and the kernel in Chapter 3.4.1.

3.2 Shallow embeddings

In machine learning, every data point is typically represented by an embedding, which is a vector of rational numbers. As we are dealing with graphs, and within graphs these embeddings relate to nodes, we will call them node embeddings.

As described in Chapter 2.2, the codomain of a classifier is also a vector of rational numbers, and thus can also be seen as a node embedding.

We have two options to find node embeddings: we can either learn the embeddings directly, or we can train a model to generate the embeddings for us. When embeddings are learned directly, we call them shallow embeddings. This name arises from the fact that no deep learning model was used.

Let us say we want to predict the Katz index of any two nodes v_1 and v_2. Note how even two nodes that do not have an edge between them will have a Katz index statistic. We can construct a model

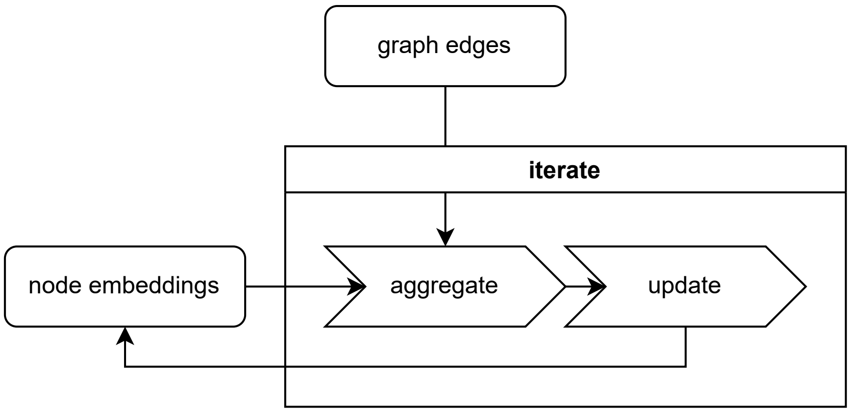


Figure 3.1: A visual representation of the message-passing algorithm graph neural networks use.

$DEC(ENC[v_1], ENC[v_2])$ where DEC is a decoder and ENC is a lookup table for shallow node embeddings. We can learn these shallow node embeddings, as well as the decoder if required, by optimising this model for a function, for instance the Katz index.

The goal is that these shallow embeddings will generalise. That is, while during training we should visit every node, we might not necessarily visit every pair of nodes. The decoder should still be able to make an accurate prediction on pairs of nodes it hasn't seen during training.

Do note how absolutely no node features are used in this framework. The shallow node embeddings rely only on the structure of the graph. Instead of training a neural network that can generate these node embeddings, we learn the node embeddings themselves.

3.3 Graph neural networks

Other than learning shallow node embeddings directly, we can also train a model to generate embeddings for us. Graph neural networks are particularly suited for this.

To achieve this, graph neural networks use a message-passing algorithm. At the start, every node's embedding is set to equal its features. Then, in every iteration, every node will aggregate all the embeddings of its neighbours, and then update its own embedding using those aggregated embeddings as well as its own current embedding. Figure 3.1 shows a visual representation of the message-passing algorithm. What exactly happens in the aggregation and update steps depends on the graph neural network used.

We will denote a node embedding for a node v after n generations of message-passing as h_v^n.

$$\begin{aligned} h_v^0 &= \text{an initial node embedding based on the node label} \\ h_v^n &= \text{upd}_n(h_v^{n-1}, \text{aggr}_n(v)) \end{aligned}$$

In Sections 3.3.1 and 3.3.2, we will discuss some possible aggregation and update methods respectively.

3.3.1 Aggregation methods

The most basic form of aggregation is simply summing the node embeddings of all neighbours, as shown in Equation 3.1.

$$h_v^n = \text{upd}_n(h_v^{n-1}, \sum_{(u \to v) \in E} h_u^{n-1}) \tag{3.1}$$

This basic form of aggregation will produce higher values for nodes with a high indegree. It can be normalised by either dividing it by the degree of v, or by dividing it by both the degree of v and the degree of u. The latter is called **symmetric normalisation**. These can be seen in Equation 3.2 and 3.3 respectively.

$$h_v^n = \text{upd}_n\left(h_v^{n-1}, \sum_{(u\to v)\in E} \frac{h_u^{n-1}}{\text{indeg}(v)}\right) \tag{3.2}$$

$$h_v^n = \text{upd}_n\left(h_v^{n-1}, \sum_{(u\to v)\in E} \frac{h_u^{n-1}}{\sqrt{\text{indeg}(v)\cdot\text{indeg}(u)}}\right) \tag{3.3}$$

Normalising the aggregation like this is not always beneficial. It obscures information about the degree of the nodes, while structural information is exactly what we are trying to use.

A second class of aggregations are the **set pooling aggregations**. They use a traditional multilayer perceptron neural network, or MLP, to perform the aggregation. The most basic version of this, defined in Equation 3.4, simply adapts Equation 3.1 by applying an MLP to every node embedding. Variations of this use the element-wise maximum as in Equation 3.5. These can also be normalised in the same ways as seen before.

$$h_v^n = \text{upd}_n\left(h_v^{n-1}, \sum_{(u\to v)\in E} \text{MLP}(h_u^{n-1})\right) \tag{3.4}$$

$$h_v^n = \text{upd}_n\left(h_v^{n-1}, \text{emax}(\{\!\{h_u^{n-1} | (u \to v) \in E\}\!\})\right) \tag{3.5}$$

Both basic aggregation and set pooling aggregations are **permutation-invariant**, i.e. they will give the same result regardless of in which order we perform the aggregation. Janossy pooling as defined in Equation 3.6 is a aggregation method that will yield different results depending on in which order the neighbours of a node are collected.

$$h_v^n = \text{upd}_n\left(h_v^{n-1}, \text{MLP}(\rho(\text{permute}(\{\!\{h_u^{n-1} | (u \to v) \in E\}\!\})))\right) \tag{3.6}$$

Here ρ is a permutation-sensitive function that might have its own internal parameters that require learning. Because this yields different results depending on the aggregation order, it might make sense to try and mitigate this. The most obvious one is to choose permute so it always puts the contents of the same multiset into the same order. This can be achieved by sorting the elements of the multiset from small to large. A second way to adopt Janossy pooling to produce more stable results is by averaging the result over many or all permutations. The latter is shown in Equation 3.7.

$$h_v^n = \text{upd}_n\left(h_v^{n-1}, \text{MLP}\left(\sum_{\pi\in\Pi} \frac{\rho\left(\text{permute}_\pi(\{\!\{h_u^{n-1} | (u \to v) \in E\}\!\})\right)}{|\Pi|}\right)\right) \tag{3.7}$$

In this equation, Π is the set of all possible permutations.

The final aggregation method we will discuss is neighbourhood attention aggregation. This is again similar to Equation 3.1. It uses an attention head att to prioritise some neighbours. Neighbourhood attention aggregation is defined in Equation 3.8. The attention head itself might also contain parameters that have to be learned, or even an entire MLP.

$$h_v^n = \text{upd}_n\left(h_v^{n-1}, \sum_{(u\to v)\in E} \text{att}(v,u)\cdot h_u^{n-1}\right) \tag{3.8}$$

3.3.2 Update methods

The most basic update function simply sums the current node embedding, the embedding of its environment, adds a bias b, and gives it all to an activation function σ. This is shown in Equation 3.9. W_{self}^n and W_{neigh}^n are a learnable matrices.

$$h_v^n = \sigma\left(W_{self}^{n-1} \times h_v^{n-1} + W_{neigh}^{n-1} \times \text{aggr}_n(v) + b_n\right) \tag{3.9}$$

Often, to simplify the update function, no distinction is made between the current node and its neighbourhood. Every node is simply considered to have an edge to itself. This results in the simpler Equation 3.10.

$$h_v^n = \sigma\left(W_{neigh}^{n-1} \times \text{aggr}_n(v) + b_{n-1}\right) \tag{3.10}$$

When we combine this idea with the symmetric normalisation as seen in Equation 3.3, we get a **graph convolutional network**. This is defined in Equation 3.11. Graph convolutional networks are known to only work well on undirected graphs.

$$h_v^n = \sigma\left(W_{neigh}^{n-1} \times \sum_{(u \to v) \in E} \frac{h_u^{n-1}}{\sqrt{\text{indeg}(v) \cdot \text{indeg}(u)}} + b_{n-1}\right) \tag{3.11}$$

This basic update step does have some limitations when it comes to deeper networks. As more message-passing iterations are added to the network, the impact of earlier iterations, which represents the immediate neighbourhood of every node, will be drowned out. Gradually, all generated embeddings for all nodes will become similar.

A way we can counteract this effect is by storing the intermediate embeddings generated at each step of the message passing algorithm, and creating a new final embedding by concatenating all of these.

3.4 Expressiveness of graph neural networks

Knowing the framework graph neural networks conform to, we can investigate how powerful or expressive they are, i.e. what sort of structural information can and can they not use to make predictions. To do so, we must first understand the Weisfeiler-Lehman isomorphism test, and a more generalised adaptation of it.

3.4.1 Weisfeiler-Lehman isomorphism

The Weisfeiler-Lehman isomorphism test is a test that in nearly all cases can determine whether two graphs are isomorphic.

We borrow the definition of graph isomorphism from Hsieh, Hsu, and Hsu, adopted to our own definition of a graph [HHH06]. Specifically, given two graphs $G_1 = (V_1, E_1, L_1)$ and $G_2 = (V_2, E_2, L_2)$, we say that G_1 and G_2 are isomorphic if and only if there exists a bijective function $f : V_1 \to V_2$ for which the following holds.

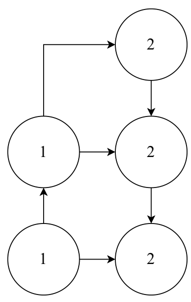
1. **Label-equivalence**: $\forall v_1 \in V_1 : L(v_1) = L(f(v_1))$.
2. **Edge-equivalence**: $\forall (v_1, u_1) \in E_1 : (f(v_1), f(u_1)) \in E_2$.

Applying the Weisfeiler-Lehman algorithms for the purpose of graph isomorphism testing was already described by Shervashidze et al., who built on earlier work on graph reductions done by Weisfeiler and Leman [She+11; WL68].

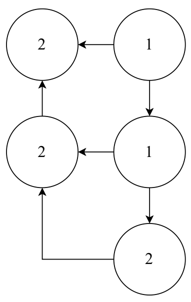
This test takes two graphs, and either passes or fails. If the test passes, it means the two graphs are possibly isomorphic, however, if the test fails, the two graphs will never be isomorphic.

The test works by finding a **canonical labelling** $K_\infty : V \to \mathrm{T}_\infty$ for a graph. Two graphs that produce the same canonical labelling pass the Weisfeiler-Lehman isomorphism test. We can find a canonical labelling K_∞^G for a graph G by computing a sequence of functions $K_0^G, K_1^G, K_2^G, ..., K_\infty^G$ which are inductively defined in Equation 3.12. Each K_i^G maps the nodes in G to "canonical labels" drawn from an alphabet T_∞. In practical algorithms, T_∞ is just the set of natural numbers. In the following definitions, it is important that hash is an injective function with codomain T_∞.

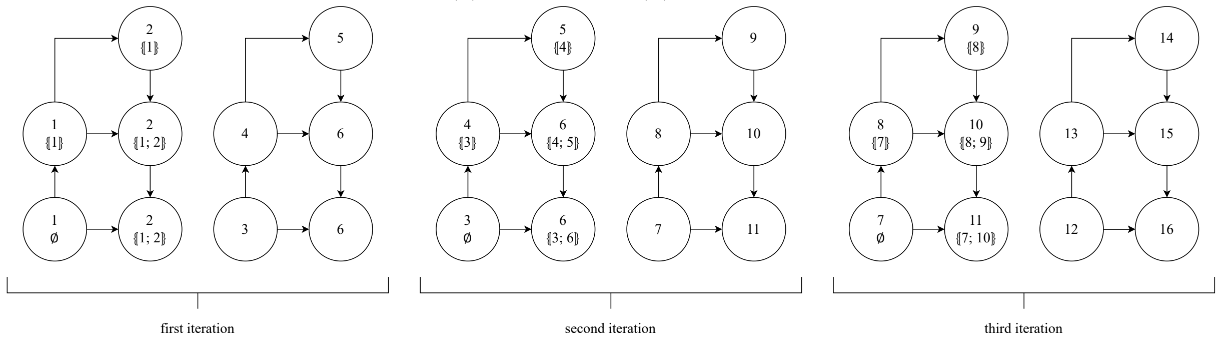
$$\begin{aligned} K_0^G(v) &= \text{hash}(L(v)) \\ K_i^G(v) &= \text{hash}\left(K_{i-1}^G(v), \{\!\{K_{i-1}^G(u) | u \in inv(v)\}\!\}\right) \end{aligned} \tag{3.12}$$



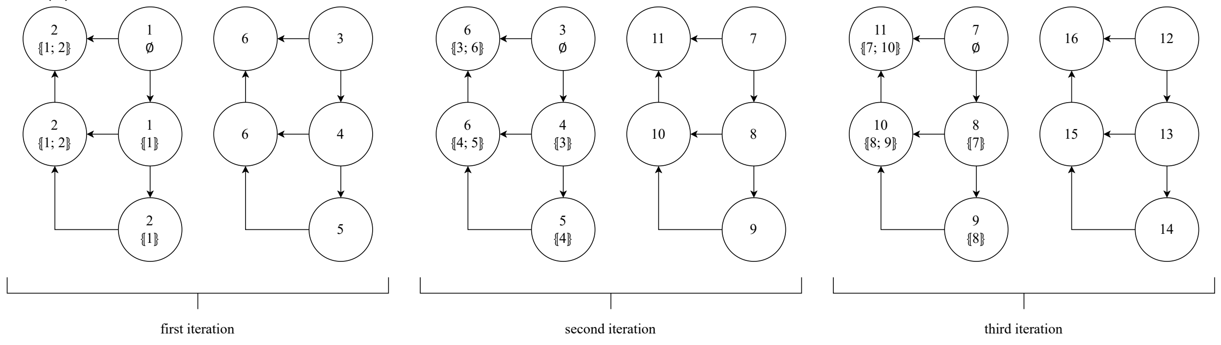
(a) Graph A.



(b) Graph B.



(c) The Weisfeiler-Lehman isomorphism test applied to graph A. We reach a stable labelling after three iterations.



(d) The Weisfeiler-Lehman isomorphism test applied to graph B. Like graph A, we reach a stable labelling after three iterations.

Figure 3.2: The Weisfeiler-Lehman isomorphism test applied to two isomorphic graphs. Both graph A and graph B have the same canonical labelling for their nodes, and thus as a pair they pass the Weisfeiler-Lehman isomorphism test.

For two graphs $G = (V^G, E^G, L^G)$ and $H = (V^H, E^H, L^H)$, we say two labellings $K_i^G : V^G \to \mathrm{T}_i^G$ and $K_i^H : V^H \to \mathrm{T}_i^H$ are equivalent if and only if there exists a bijection $f : V^G \to V^H$ such that the following holds.

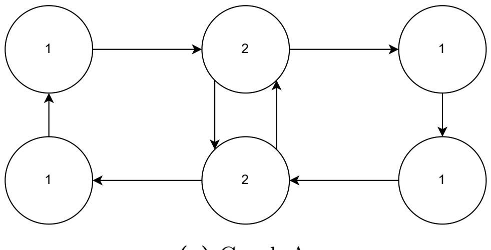
$$\forall v \in V^G : G(v) = L^H(f(v)) \tag{3.13}$$

$$\forall v_1, v_2 \in V^G : K_i^G(v_1) = K_i^G(v_2) \iff K_i^H(f(v_1)) = K_i^H(f(v_2)) \tag{3.14}$$

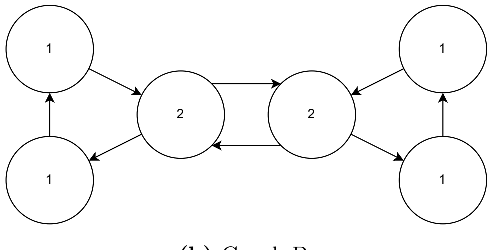
When a labelling K_l^G is equivalent to K_{l+1}^G we say the labelling K_l^G is stable, and we can decide that all labellings K_i^G where $i \geq l$ are equivalent to K_l^G. We define $K_\infty^G = K_l^G$ with K_l^G stable. Such a K_l^G always exists.

When two graphs have a labelling K_∞^G and K_∞^H equivalent to each other, the graphs pass the Weisfeiler-Lehman isomorphism test. We say the graphs are indistinguishable by Weisfeiler-Lehman. An example of this is given in Figure 3.2.

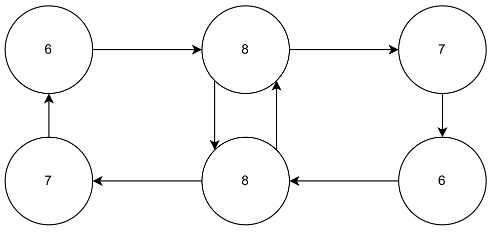
As previously mentioned, just because two graphs pass the Weisfeiler-Lehman isomorphic test, and thus are indistinguishable by Weisfeiler-Lehman, this does not guarantee the two graphs are isomorphic. In



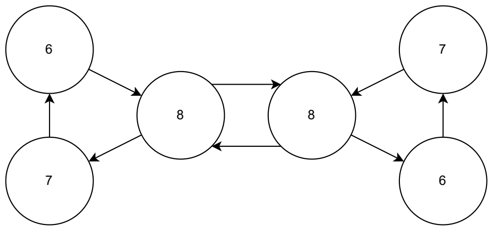
(a) Graph A.



(b) Graph B.



(c) A canonical labelling for graph A, retrieved by applying Weisfeiler-Lehman.



(d) A canonical labelling for graph B, retrieved by applying Weisfeiler-Lehman.

Figure 3.3: Even though both graphs are not isomorphic, they will pass the Weisfeiler-Lehman isomorphism test. This should make clear that simply passing the test does not guarantee isomorphism, but merely doesn't rule it out.

fact, no algorithm that can efficiently do this in polynomial time is known. Figure 3.3, which was inspired by an example provided by Sato, demonstrates two graphs that are indistinguishable by Weisfeiler-Lehman, yet not isomorphic [Sat20].

3.4.2 Bisimulation

We can also create a more generalised version of the Weisfeiler-Lehman isomorphism test, that can test whether two graphs are bisimilar too.

Given two graphs $G = (V_G, E_G, L_G)$ and $H = (V_H, E_H, L_H)$ where L_G and L_H share the same codomain Σ, we say two rooted graphs (G, v) and (H, u) are ***c*-graded, *l*-rooted bisimilar**, denoted as $(G, v) \sim_{\#}^{c,l} (H, u)$, when the following condition must be satisfied.

- **label-equivalence**: $L_G(v) = L_H(u)$

Additionally, when $l > 0$, the following conditions must be satisfied.

- ***c*-graded forth**: $\forall k \in \mathbb{N}, k \leq \min\left(c, \text{indeg}_G(v)\right)$ and all pairwise distinct $v_1 ... v_k \in \text{inv}_g(v)$ there are pairwise distinct $u_1 ... u_k \in inv_H(u)$ such that $\forall i \in \mathbb{N}, i \leq k : (G, v_i) \sim_{\#}^{c,l-1} (H, u_i)$.

- ***c*-graded back**: $\forall k \in \mathbb{N}, k \leq \min\left(c, \text{indeg}_H(u)\right)$ and all pairwise distinct $u_1 ... u_k \in inv_H(v)$ there are pairwise distinct $v_1 ... v_k \in \text{inv}_g(v)$ such that $\forall i \in \mathbb{N}, i \leq k : (H, u_i) \sim_{\#}^{c,l-1} (G, v_i)$.

When a bijective function $f : V_G \to H_G$ exists so that $\forall v_G \in V_G : (G, v_G) \sim_{\#}^{c,l} (H, f(v_G))$, and vice-versa $\forall v_H \in V_H : (G, f(v_H)) \sim_{\#}^{c,l} (H, v_U)$, we say the two graphs are c-graded l-round bisimilar. When $l = 1$ and $c = 1$, we say the two graphs are **bisimilar**.

The definition for c-graded l-round bisimilarity was largely adopted from Otto [Ott19].

Two graphs G and H are ∞-graded ∞-round bisimilar if and only if they are indistinguishable to Weisfeiler-Lehman. Two graphs are ∞-graded l-round bisimlilar if and only if their labellings K_l as defined in Equation 3.12 are equivalent. We can further adopt the definition of the labellings given in Equation 3.12 to also test for c-graded bisimlilarity.

For future convenience, we will declare the procedure $\text{LAB}(G, l, c)$ which produces the l-round c-graded labellings.

$$\begin{aligned}\text{LAB}(G,0,c)(v) &= \text{hash}(L(v))\\ \text{LAB}(G,i,c)(v) &= \text{hash}\left(\text{LAB}(G,i-1,c)(v), \{\!\{\text{LAB}(G,i-1,c)(u) | u \in inv(v)\}\!\}|_{\leq c}\right)\end{aligned} \tag{3.15}$$

3.4.3 Parallels between Weisfeiler-Lehman and message-passing

To understand the expressiveness of graph neural networks, it is important to see that the message-passing algorithm is largely analogous to the Weisfeiler-Lehman isomorphism test.

In fact, we can implement the labelling seen in Equation 3.15 within the message-passing framework, by during the update step using the hash function used in the Weisfeiler-Lehman test. In fact, the message-passing algorithm is only equal in power to the Weisfeiler-Lehman algorithm when the upd function, like the hash function, is injective [Sat20].

This implies that if two graphs are indistinguishable to Weisfeiler-Lehman, then they are indistinguishable to message-passing. Furthermore, if for two graphs G and H, $K_l^G(v) = K_l^H(v')$, than a graph neural network with l message passing steps will assign the same embedding to $v \in G$ and $v' \in H$ [Sat20].

Chapter 4

Compression

In the previous chapter we learned that graph neural networks are only as powerful as the Weisfeiler-Lehman isomorphism test. The goal of this thesis is to abuse this fact, by for a graph G, creating a smaller, compressed graph G', which is indistinguishable to Weisfeiler-Lehman from G. This graph, being smaller, could be easier to use during training as it would be less computationally expensive.

4.1 Graph compression

The idea behind the algorithm introduced in this chapter, is that we can "merge" two nodes that have the same labelling into a single node. After all, these two nodes cannot be distinguished from each other by the message-passing algorithm.

To aid with this, we define the concept of equivalence classes. For a graph G, an l-round, c-graded equivalence class is a set of nodes which all have the same labelling in $\mathrm{LAB}(G, l, c)$. We also define the function $\mathrm{eqclass}_{K^G}(v)$ for some labelling K^G. It computes the largest subset S of nodes of the graph G, $S \subseteq V_G$, such that $\forall v \in S, \forall u \in V_G : K^G(v) = K^G(u) \implies u \in S$.

We define a procedure $\mathrm{COMP}(G, l, c)$ to compress a graph $G = (V, E, L)$ into a smaller graph $G' = (V', E', L')$. Graphs G and G' are l-round, c-graded bisimilar. The definition can be found in Algorithm 1.

The algorithm works by first choosing a representative node $u' \in V$ for every equivalence class in the labelling. Then, for every representative node, we iterate over all nodes $v \in V$ that have an outgoing edge to u', and find the representative node v' that represents the equivalence class v belongs to. We add the edge $v' \to u'$ to the compressed graph G'. Finally, any edge which has more than c duplicates has these excess duplicates over c removed. This can be done without breaking the l-round c-graded bisimilarity of the compressed graph with the uncompressed graph.

It should be noted the compression algorithm is not unambiguous. Depending on which node is chosen to represent an equivalence class, the compressed graph can end up being larger or smaller. The node chosen does not impact the information present in the compressed graph, after all, the uncompressed and compressed graphs are still l-round, c-graded bisimilar, but it does impact the size of the compressed graph as well as the general weight of the information. A representative with a higher indegree will not have any incoming nodes with different labels than a representative with a smaller indegree, but it will have some duplicates.

We always choose a node with the smallest indegree to represent an equivalence class. This heuristic tries to minimize the size of the compressed graph. It does not guarantee is however, since in a later step any duplicate edges over c will be removed, which could turn a node with a high indegree into one with a much lower indegree.

Every node in the compressed graph G' will represent an equivalence class of nodes in the uncompressed graph G for some labelling.

Algorithm 1 Graph compression

```
input: A graph $G = (V, E, L)$, $l$, $c$
output: A compressed graph $G' = (V', E', L')$

function COMP(G, l, c)
    $V' \leftarrow \{\}$
    $E' \leftarrow \{\!\{\}\!\}$
    $K \leftarrow LAB(G, l, c)$                    ▷ A canonical labelling for the graph.
    $S \leftarrow \{\}$                            ▷ A set of seen equivalence classes.

    for $v \in V$ ordered by indeg($v$) from small to large do
        if $K(v) \notin S$ then                    ▷ Choose one representative per equivalence class.
            $V' \leftarrow V' \cup \{v\}$
            $S \leftarrow K(v)$
        end if
    end for

    $L' \leftarrow L$, limiting its domain to $V'$

    for $(u', (v \to u')) \in V' \times E$ do
        Find $v' \in V'$ such that $K(v) = K(v')$.   ▷ $v$ and $v'$ are in the same equivalence class.
        $E' \leftarrow E' \cup \{\!\{(v', u')\}\!\}$
    end for

    $E' \leftarrow E'|_{\leq c}$
    $G' \leftarrow (V', E', L')$
    return $G'$
end function
```

4.2 Compression of train, validation, and test sets

To be able to train a model over a compressed graph, we also have to compress the train, validation and test classifiers. After all, the original classifiers mapped each node to a ground truth. In the compressed graph, every node can represent multiple nodes in the original graph, and might thus correlate to multiple ground truths in the original graph.

Given a classifier D, and a compressed graph G', we can define a compressed classifier $D' : V \to \mathbb{Q}^d$. For every node $v \in G'$, we define $D'(v)$ to be the probability distribution over the multiset $\{\!\{D(v) | v \in \text{eqclass}_{K^G}(v')\}\!\}$. For example, given classifier that maps every node to either red or green, and an equivalence class containing two red nodes and 8 green nodes, we would end up with a probability distribution of $(0.2, 0.8)$.

Chapter 5

Methodology

In the thesis we attempt to exploit the expressiveness of graph neural networks to compress large graphs into smaller graphs that should contain the same amount of information as the original graph, at least from the perspective of a graph neural network.

First we will present the datasets used to perform any experiments in this thesis. We will then present a framework all regular and graph neural networks used fit into. We then describe a variety of loss functions employed, the exact parameters used for training and compressing, and finally we present a large number of models used in the experiments.

The experiments themselves are outlined in Chapter 6.

5.1 Datasets

Before we can test out the compression algorithm, we need to decide on which datasets to use. We chose to work with the ogbn-arxiv[1] and deezer-europe datasets.

An summary of both datasets is provided in Table 5.1.

5.1.1 Dataset ogbn-arxiv

The ogbn-arxiv dataset is a dataset provided by from Hu et al. [Hu+20]. In their Open Graph Benchmark[2] collection, they provide various datasets, as well as node-level, edge-level and graph-level tasks to perform on those datasets. These datasets are wildly used to test graph neural network performance, which makes them particularly useful. From the many datasets within the collection, only the ogbn-arxiv dataset was chosen to work with, which is the smallest one amongst the node-classification datasets. The dataset ogbn-products was also considered, and briefly worked with, but due to the bigger size it became too difficult to reliably train classifiers on this graph without significantly deviating from the methodology. This is because the datasets were sufficiently large they occasionally caused PyTorch, the deep learning framework used, to run out of working memory on the GPU. A solution for this could be to use batches of data, but sampling batches of a graph has challenges on its own, so it was decided to use the other datasets.

[1]Arxiv is pronounced like archive.

[2]`https://ogb.stanford.edu/`

Table 5.1: An overview of the sizes of both datasets used. The table shows for each dataset, the amount of edges, the amount of nodes, the amount of nodes in the train set, the amount of nodes in the validation set, the amount of nodes in the test set, the size of the node features, and the amount of classes the classification task must be trained for.

Dataset	Edges	Total nodes	Train count	Val. count	Test count	Features	Classes
ogbn-arxiv	1166243	169343	90941	29799	48603	128	40
deezer-europe	185504	28281	18854	4713	4714	128	2

Table 5.2: A sample of seven nodes in the ogbn-arxiv dataset. Shown are the title of the paper the node represents, the index of its category, as well as the name and a description of that category.

	Category		
Title	Index	Name	Description
Geometry and the complexity of matrix multiplication	9	cs.CC	Computational Complexity
On the Design and Analysis of Multiple View Descriptors	16	cs.sc	Symbolic Computation
Tensor Graph Convolutional Networks for Prediction on Dynamic Graphs	24	cs.ML	Machine Learning
Lattice polytopes in coding theory	28	cs.IT	Information Theory
Universal Hashing for Information Theoretic Security	28	cs.IT	Information Theory
Fine-Grained Analysis of Propaganda in News Articles	30	cs.CL	Computation and Language
A CF-Based Randomness Measure for Sequences	39	cs.DM	Discrete Mathematics

The dataset represents a collection of papers retrieved from arXiv, a free academic paper distribution system. The collection is limited to only computer science papers. Specifically every paper is represented by one node in the graph. The label of the node is equal to an embedding of the paper's title and abstract. Papers have outgoing edges to the other papers they cite. The classification task with this dataset is to categorise all papers into one of forty categories. A sample of this dataset can be found in Table 5.2.

The dataset was already split into a train, validation and test dataset by Hu et al., and these subsets were kept. The splitting was done based on the year the paper was published in, with papers published until 2017 added into the train set, papers published in 2018 added into the validation set, and papers published in 2019 making up the test set.

5.1.2 Dataset deezer-europe

The deezer-europe dataset was retrieved from Rozemberczki and Sarkar [RS20]. Much like the ogbn-arxiv dataset, it was chosen because of its relatively approachable size. The dataset was collected from Deezer, a music streaming service. The collection specifically limits itself to European users of Deezer. The graph is a social-network style graph, where every node represents a user, and is labelled with an embedding of the artists the users liked on the platform. Users have outgoing edges to any other users they follow on the platform. The classification task with this dataset is to categorise every user as either male or female.

The dataset was randomly split into a train, validation and test set, with the train set taking up $\frac{4}{6}$ of the entire set, and the validation and test set both taking up $\frac{1}{6}$ each.

5.1.3 Dataset compression

The two datasets chosen have continuous node labels, which means it is extremely rare for two nodes to have the same label. At least in the ogbn-arxiv dataset, no two nodes share the same label. This is an issue for the compression algorithm as defined in Section 4.1, as two nodes with a different label can never end up within the same equivalence class, and as a result no compression occurs.

Since we were unable to find a node-classification dataset with discrete labels, we chose to instead pre-process the datasets before compressing them. We create a pre-compression neural network on each dataset, and to train these models we only use the node features of every graph, so they are not graph neural networks already. What these pre-compression models look like exactly will be explained in Section 5.5.3. We use the predictions of the pre-compression models to construct a new, discrete graph, where every node is a one-hot encoded vector of the node's most likely class. We name these new datasets discrete-ogbn-arxiv and discrete-deezer-europe, respectively.

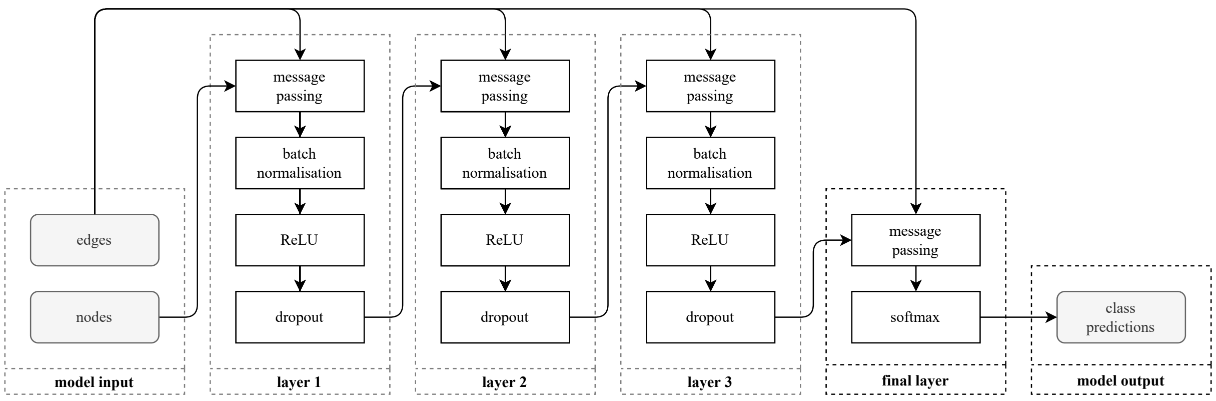


Figure 5.1: The generic model all models in this thesis will be based on. The dropout steps are only present during training. For models that are not graph neural networks, the edges and all its outgoing flows will not be present, and the message passing step will instead simply be a linear matrix multiplication of the node features with a learnable matrix.

It is important to note that these discrete datasets will have both correctly labelled nodes, i.e. nodes where the label matches the class defined by the to be learned classifiers, as well as incorrect labelled nodes, i.e. nodes where the label does not match the class defined by the classifier. This is because the labels of the discrete datasets are generated by the pre-compression models, and this model is very unlikely to have 100% accuracy. After all, if it was, there would be little point to compressing the graph and training a new model.

For each of these two discrete graphs, we create another six compressed graphs, by running the compression algorithm for values of $l \in \{1, 2, 3\}$ and for values of $c \in \{1, \infty\}$. We refer to them simply as $\text{COMP}(dataset, l, c)$ for the dataset, the value of l and the value of c, named after Algorithm 1.

5.2 Model structure

While many models were used in the production of this thesis, both traditional multilayer perceptrons, and graph neural networks, they can all be generalised within the same framework. The models are strongly inspired by the models presented by Hu et al. [Hu+20].

Every model will contain a series of message-passing layers, followed by a final layer that transforms the output to something suitable for the classification task.

Specifically, every message-passing layer will contain a message-passing aggregation and update step, followed by a batch normalisation step which helps by speeding up training by normalising the standard deviations of every node embedding, and a ReLU activation step. The final layer will only contain a message-passing step followed by a softmax step. During training, a dropout step is added between every layer to prevent overfitting. An illustration representing this generic model is provided in Figure 5.1. The minimum amount of layers in a model is one. Such a model would have just the final layer.

While the size of the model's input and output depend on the dataset it is meant to operate on, the size of the features passed between layers is configurable. It is, however, always the same between all layers within a model. For brevity we refer to the size of the input feature vector as the input size, the size of the output feature vector as the output size, and the size of the features passed between layers as the hidden size.

We will use three message-passing algorithms: None, GCN and GraphSAGE.

- The None algorithm is a simple linear matrix multiplication with a learnable matrix, and an added bias. It will not perform any actual message-passing. Models using this algorithm are not graph neural networks.
- The GCN algorithm is a message-passing step as described in Equation 3.11. Do note that this will naturally not be a good match for our datasets, as they are directed graphs.

- The GraphSAGE message-passing algorithm was first presented by Hamilton, Ying, and Leskovec [HYL17]. It is heavily based on the Weisfeiler-Lehman isomorphism test, replacing the hash function in the algorithm with a trainable MLP, and only aggregating a fixed-size subset of neighbours. This makes it an ideal candidate for the experiments.

5.3 Loss functions

All networks will generate a probability distribution as output, which will be used to generate a classification. To minimise the difference between the predicted probability distribution, and the ground-truth distribution, we need to define a loss function.

Given the prediction distribution p and the ground-truth distribution q, and n the number of classes in the probability distributions, we will present four loss functions in the remainder of this section.

All loss functions are defined to calculate the loss of a single node. To calculate the loss over the entire graph, it is sufficient to calculate the average loss over the nodes. When the graph is a compressed graph, this average is weighted by the size of the equivalence class that each node in the compressed graph represents.

5.3.1 Cross-entropy loss

The cross-entropy loss function is a typical loss function for classification tasks. The cross-entropy loss function is defined in Equation 5.1.

$$\mathcal{L}_{ce}(p,q) = -\sum_{i=1}^{n} \log(p_i) \cdot q_i \tag{5.1}$$

Typically the cross-entropy loss function works with a one-hot encoded vector q, where the value of the ith element is 1 if the ith class is the correct one, and 0 otherwise. This is because usually it is meant to be used for classification tasks, not for comparing probabilities. We will create an adapted version of the cross-entropy loss function that converts the ground-truth probabilities q into a one-hot encoded vector where the most likely class is set to 1, and all others are 0. We will call this the cross-entropy classification loss function, or $\mathcal{L}_{cec}$. For uncompressed graphs every node already has exactly one matching class, but this function is especially important for the compressed graphs where q is an actual probability distribution with possible probabilities for many classes.

5.3.2 Kullback-Leibler divergence

The Kullback-Leibler divergence loss function is similar to the cross-entropy loss. It is based on the Kullback-Leibler divergence, which is defined in Equation 5.2. This divergence is undefined when either p or q are zero for any of their elements, but as p_i gets closer to 0, the result of the single iteration of the summation will converge to 0 as well. As such, we will consider any part of the summation where p_i is 0 to be 0 as well.

While it is unlikely any model will predict a probability of 0 for any class, this will be common in our ground-truth distribution. That's why the actual Kullback-Leibler divergence loss function we use, as defined in Equation 5.3, will swap around p and q.

$$kl(p,q) = \sum_{i=1}^{n} p_i \cdot \log\left(\frac{p_i}{q_i}\right) \tag{5.2}$$

$$\mathcal{L}_{kl}(p,q) = kl(q,p) \tag{5.3}$$

5.3.3 Jensen-Shannon divergence

Finally, we also consider the **Jensen-Shannon divergence** loss function, which is defined as in Equation 5.4. This loss function is similar to the Kullback-Leibler divergence loss function, but is symmetric i.e. $\mathcal{L}_{js}(p,q) = \mathcal{L}_{js}(q,p)$. It achieves this by, instead of measuring the divergence between p and q directly, it first calculates the average distribution m, and measures the distances between p and m, and q and m.

$$\begin{aligned} m &= \frac{(p+q)}{2} \\ \mathcal{L}_{js}(p,q) &= \frac{kl(p,m)+kl(q,m)}{2} \end{aligned} \tag{5.4}$$

We do not need to swap around our parameters as we had to with the Kullback-Leibler divergence loss function, as m will never contain any 0 values, as p will never have any either.

5.4 Training

All models in this thesis were trained on the full graph, with all nodes features and all edges. The trained models were optimised using an Adam optimiser, with a learning rate of 0.01. The dropout rate was set to 0.5. These values were adopted from Hu et al. [Hu+20], whose implementation the models in this thesis were based on.

The Adam optimiser uses a gradient-descent based method to find the model parameters. As is custom, the gradient was only calculated on the training data.

All models were implemented in, and all training was done with PyTorch [Pas+19]. PyTorch is a well-known general-purpose deep learning framework for the Python programming language.

PyTorch in itself does not provide any functionality for constructing or training graph neural networks. For this we used the PyG library, short for 'PyTorch Geometric' [FL19]. It provides off-the-shelf message-passing algorithms for both GCN and GraphSAGE.

The training was done on the infrastructure of the Flemish Supercomputer Center (VSC)[3].

The compression algorithm was implemented in the Rust programming language[4]. This choice was made since it allows to quickly and relatively easily create a fast-running implementation, which is important as the compression algorithm cannot easily make use of GPU acceleration. For the datasets used in this thesis, the compression algorithm will be able to find a stable labelling within a few seconds. The ndarray[5] and ndarray-npy[6] libraries were used to read and create datasets that are also easily used by Python code, and the dashmap[7] and rayon[8] libraries were used to parallelise the compression algorithm over multiple CPU cores. Rust's built-in testing features were used to ensure the compression algorithm was working correctly, comparing it to a manually compressed graph.

5.5 Model variants

Many variants of the basic model structure described in Chapter 5.2 were trained. A summary of all model types is presented in 5.3.

5.5.1 Reference models

In total 18 reference models were trained. These reference models do not operate on compressed graphs at all, and are used as a baseline, as they illustrate the accuracy of graph neural networks models trained on uncompressed and original graphs, with continuous node features. All these models have a hidden size of 128 and use the cross-entropy loss function. They were all trained for 100 epochs on the ogbn-arxiv or deezer-europe datasets directly.

Every reference model was trained with either 1, 2 or 3 layers, and using the MLP, GCN or GraphSAGE for their message-passing steps.

[3]https://www.vscentrum.be/
[4]https://www.rust-lang.org/
[5]https://github.com/rust-ndarray/ndarray
[6]https://github.com/jturner314/ndarray-npy
[7]https://github.com/xacrimon/dashmap
[8]https://github.com/rayon-rs/rayon

The naming convention of these models is the prefix ”M” followed by the abbreviated name of the original dataset it was trained on, i.e. ‘arxiv’ for the ogbn-arxiv dataset, and ”deezer” for the deezer-europe dataset. This is followed by either ‘none’, ‘gcn’ or ‘sage’ depending the message-passing algorithm used, finally followed by the number of layers.

For example, the M-arxiv-none-1 model is the reference model trained on the ogbn-arxiv dataset doing no message-passing, and is one layer deep.

5.5.2 Discretised reference models

In addition to the already mentioned reference models, we also trained the following reference models on the discretised datasets. They serve as a baseline of how well graph neural networks perform on uncompressed but discrete-label graphs. All these models count 3 layers, a hidden size of 256, and were trained for 500 epochs. This was only done for the ogbn-arxiv dataset, as a sanity check that training on discretised data would be possible at all.

In total three of these models were trained on the , one for each message passing algorithm. They were all trained on the discrete-ogbn-arxiv dataset.

- The **M-pre-ref-arxiv-none** model uses no message passing.
- The **M-pre-ref-arxiv-gcn** model uses GCN message passing.
- The **M-pre-ref-arxiv-sage** model uses GraphSAGE message passing.

5.5.3 Pre-compression models

The pre-compression models are classic neural networks which do not do any message passing. This means they do not take into account any edge information, and thus are not graph neural networks.

They are used to create the discrete graphs described in Section 5.1.3. Only two of these models were created.

- **M-pre-arxiv** is a 4-layer model with 256 hidden features. It was trained for 500 epochs on the ogbn-arxiv dataset. This model was always used to create the discrete version of the ogbn-arxiv dataset used to create the compressed variant of this dataset.
- **M-pre-deezer** is a 4 layer model with 128 hidden features. It was trained for 100 epochs on the deezer-europe dataset. Just like with the M-pre-arxiv model, this model was used to create the discrete version of the deezer-europe dataset, which all compressed versions of the deezer-europe dataset were based on.

5.5.4 Post-compression models

The post-compression models are models trained on the compressed graphs. Many variants of these were created, but all variants had a hidden size of 128, and were trained for 100 epochs on their respective datasets.

On every compressed dataset, twelve post-compression models were trained, one for each (loss function, message-passing algorithm) pair. Every post-compression model had as many layers as the amount of compression-rounds used to create the dataset it was trained on.

The naming convention for the post-compression models is the prefix ”M-post” followed by the abbreviated name of the original dataset it was trained on, i.e. ‘arxiv’ for a compressed dataset based on ogbn-arxiv, and ‘deezer’ for a compressed dataset based on deezer-europe. This is followed by the abbreviated name of the message-passing algorithm used, one of ‘none’, ‘gcn’ or ‘sage’. This in its turn is followed by the abbreviated name of the loss function, which is either ‘ced’ for the cross-entropy loss, ‘cec’ for the cross-entropy classification loss, ‘kld’ for the Kullback-Leibler divergence, or ”jsd” for the Jensen-Shannon divergence. This is finally followed by a description of the compression parameters already present in the name of the compressed dataset worked with.

For example, the ‘M-post-arxiv-sage-kld-bi-3’ model was trained on the ogbn-arixiv-bi-3 dataset, using GraphSAGE as the message-passing algorithm, and the Kullback-Leibler divergence loss function.

Table 5.3: A summary of the different models trained.

	Input graph		
Model kind	Compressed	Label kind	Purpose
Reference	No	Continuous	Classification
Discrete Reference	No	Discrete	Classification
Pre-compression	No	Continuous	Discretising labels
Post-compression	Yes	Continuous	Classification

Chapter 6

Results

In this chapter we will present the results of the experiments we ran.

6.1 Accuracy of reference models

It is worth to investigate the reference models on their own to better understand the performance we are trying to achieve in the experiments, as well as to see how the exploitation of structural data within a graph can help in improving the accuracy of models in each dataset.

6.1.1 Comparison with other papers

To ensure our models are constructed and their training is performed in a correct way, it is important to compare them with results from other papers. In their paper, Hu et al. measured an accuracy of around 58% for a classical neural network, while their GraphSAGE model achieved an accuracy of around 71% [Hu+20]. Both their models managed to achieve their peak accuracy after only three layers.

This is similar to the performance of the M-arxiv-none-3 and M-arxiv-sage-3 models, which see a performance of 55% and 71% respectively.

The deezer-europe dataset was retrieved from Rozemberczki and Sarkar who also trained a variety of graph neural networks on it for the same task [RS20]. Their peak accuracy was 68% using the MixHop message-passing algorithm, which was first introduced by Abu-El-Haija et al. [Abu+19]. They achieved this in two message-passing steps. This is comparable to our M-post-deezer-sage-2 model, which achieved an accuracy of 65%. While the paper cited does not train any conventional neural networks on the dataset, our M-post-deezer-none-1 model already achieves an accuracy of 65%. As such, it is not easy for a graph neural network to use the structural information in this dataset.

6.1.2 Impact of structural data on accuracy

How accuracy increases as we add more layers is displayed in Figure 6.1a. Accuracy does increase using a graph neural network, but does not increase beyond 2 layers. This indicates that while the structure of the graph can be used to improve accuracy, looking beyond the immediate neighbourhood of a node yields little improvements.

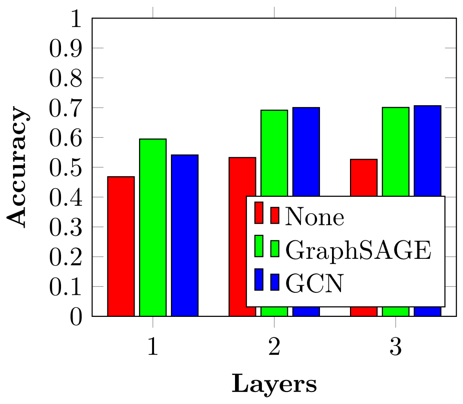
The same chart is provided for the deezer-europe dataset in Figure 6.1b. The reference models do not improve in accuracy when a graph neural network is used over a classic neural network, regardless of how many layers are added.

6.2 Feasibility of training on discretised datasets

To know whether it makes sense to train a graph neural network model on a compressed dataset, we must first verify that it is feasible to train on a discretised dataset. After all, all compressed datasets are based on discretised datasets.

Figure 6.1: Two figures showing the accuracy of the reference models by the amount of layers.

(a) The accuracy of the reference models trained on the ogbn-arxiv dataset. For all variants, peak accuracy is reached after two layers already. Using a graph neural network increases accuracy over an MLP.



(b) The accuracy of the reference models trained on the deezer-europe dataset. Using a graph neural network yields no improvement over a regular MLP. As such, the amount of layers is not of particular importance.

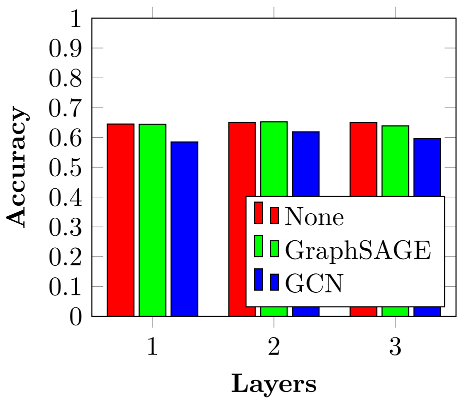


Table 6.1: A comparison of the discretised reference models to the reference models trained on the same dataset and using the same message-passing algorithm. All accuracies are test accuracies. The discretised models were trained and evaluated on the discretised datasets. There is no significant drop in accuracy by training and evaluating on a discretised dataset. Where no message passing is utilised, the discretised model does not change the predictions at all, and just passes through the input label received.

		Accuracy		
Dataset	Message-passing	Original	Discretised	Difference
ogbn-arxiv	none	55.4%	55.4%	0.00%
ogbn-arxiv	gcn	71.3%	70.6%	−0.70%
ogbn-arxiv	sage	71.3%	71.6%	0.03%

The reason one might suspect that the accuracy of models trained and evaluated on discretised datasets, as defined in Chapter 5.5.2, might be lower, is because when making the discretised dataset, no structural information in the graph is used, and as such the discrete labels it assigns to every node will be more inaccurate than a graph neural network would be able to predict, at least depending on the dataset. It might be possible that these incorrect labels will interfere with the the graph neural networks trained on these incorrect labels.

We find that the M-pre-arxiv dataset has an accuracy of 55.4%, meaning around half of the labels of the discrete-ogbn-arxiv dataset will be incorrect. The M-pre-deezer dataset fares slightly better, with an accuracy of 64.5%. This still leaves about a third of the discrete-deezer-europe dataset with incorrect labels.

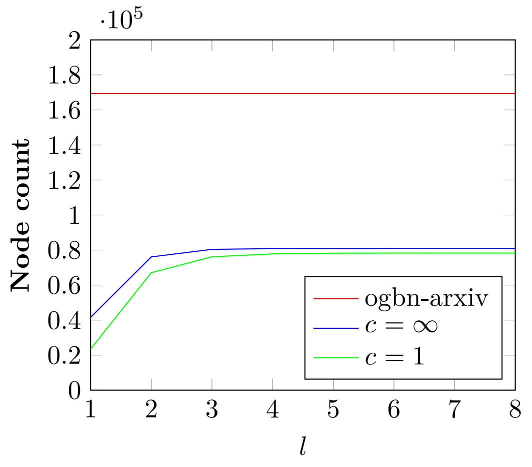
Luckily these incorrect labels do not change the accuracy of models trained on these discretised datasets in any significant way. As can be seen in Table 6.1, the accuracy of the discretised reference models is about the same as the accuracy of the normal reference models. There was no significant loss in accuracy by training on discretised dataset. This experiment was not conducted for the deezer-europe dataset, as the reference model did not improve its accuracy when a graph neural network was used over a classical neural network.

6.3 Performance of models trained on compressed graphs

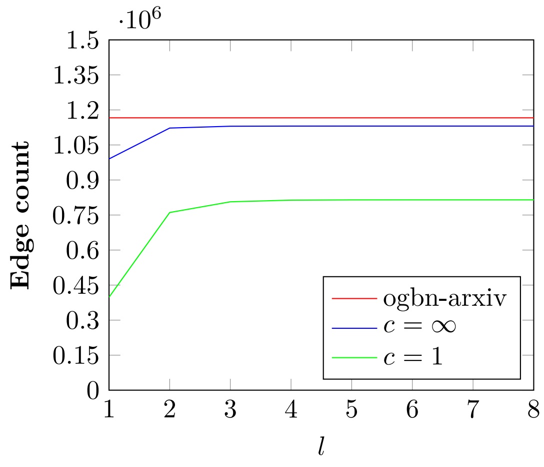
Now we know that it is feasible to train a graph neural network model on a discretised dataset, we have to evaluate whether it is still possible on compressed datasets. In this section we will first investigate how much small our graphs become by applying the compression algorithm, and then we will evaluate

Figure 6.2: The amount of nodes and edges in every compressed dataset compared to the original datasets, by the amount of compression rounds l for $c \in \{1, \infty\}$.

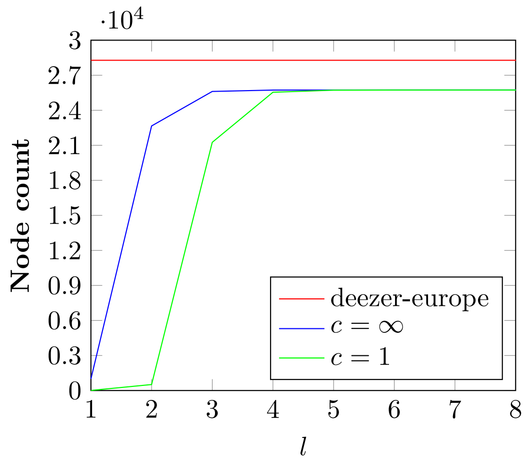
(a) The total amount of nodes in the compressed dataset COMP(discrete-ogbn-arxiv, l, c) compared to the amount of nodes in ogbn-arxiv.



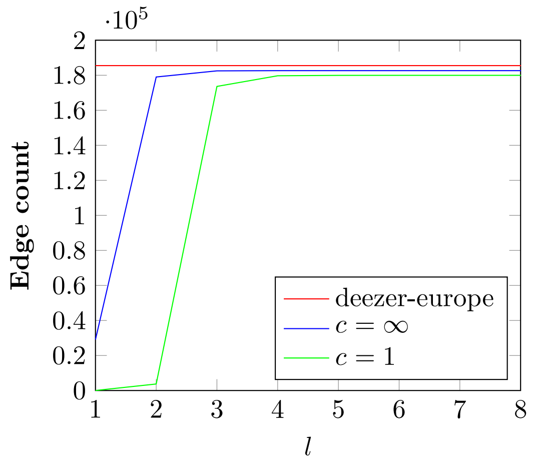
(b) The total amount of edges in the compressed dataset COMP(discrete-ogbn-arxiv, l, c) compared to the amount of edges in ogbn-arxiv.



(c) The total amount of nodes in the compressed dataset COMP(discrete-deezer-europe, l, c) compared to the amount of nodes in deezer-europe.



(d) The total amount of edges in the compressed dataset COMP(discrete-deezer-europe, l, c) compared to the amount of edges in deezer-europe.



the performance of the graph neural networks trained on the compressed datasets.

6.3.1 Compressed graphs

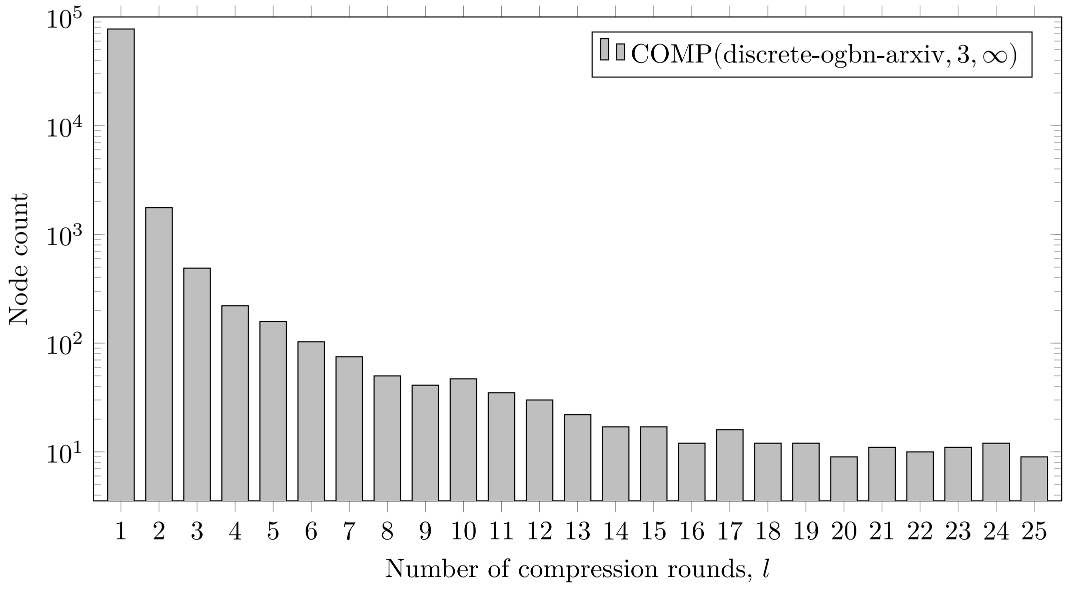
The next step is to verify that compression is at all possible. It is, after all, possible that the datasets only have nodes distinguishable from each other using Weisfeiler-Lehman or even l-round c-graded simulation.

Figure 6.2 plots how well the compression algorithm compresses the discrete datasets. Both datasets reached a stable labelling after only 7 rounds of compression.

In Figure 6.2a we see that the number of nodes in the compressed graph of ogbn-arxiv vary from around 14% of the original graph when using 1-round 1-graded bisimulation, to around 48% when using 4-round ∞-graded bisimulation. For edges the compression is less impressive, with for ∞-graded bisimulation, which as you might recall is equal to the Weisfeiler-Lehman isomorphism test, ends up at around 97%. This means that in general nodes with fewer edges must compress better than nodes with many edges. This also has intuitive sense to it: a node with fewer edges is more likely to have another node in the graph with the exactly the same neighbours.

For the discrete-deezer-europe dataset we see compression is much worse, except for lower compression rounds. For $l = 1$ and $c = 1$ the entire discrete-deezer-europe dataset gets compressed into just 8 nodes.

Figure 6.3: The size of every equivalence class during graph compression. The scale is logarithmic.



This is also the maximum amount of equivalence classes there could be. After all, a node either has no neighbours at all, a neighbour of the same sex, a neighbour of a different sex, or one neighbour of each sex. Given there are two sexes in the database, the total amount of possible equivalence classes is 8. For higher compression rounds, which are the ones we are most interested in, as they allow us to look deeper into the graph structure, the compression performs much worse. For both $c = 1$ and $c = \infty$ the number of nodes in the compressed graphs amount to around 91% of the full discrete graph, while the amount of edges amount to around 97% and 98% respectively.

When investigating what the compressed graphs look like, we find that for at least the discrete-ogbn-arxiv dataset, a lot of nodes end up in extremely small equivalence classes, as shown in Figure 6.3. For $\text{COMP}(\text{discrete-ogbn-arxiv}, 3, \infty)$, around 46% of nodes end up in a singleton equivalence class. On the other extreme end, we also have an incredibly big equivalence class. One equivalence class counts 11709 nodes, which represents around 6.91% of the total number of nodes in the graph.

Looking at the composition of the equivalence classes, we find they are extremely heterogeneous, i.e. they contain a lot of nodes with a different ground-truth label. Staying with $\text{COMP}(\text{discrete-ogbn-arxiv}, 3, \infty)$, we find typical equivalence classes have a lot of incorrect nodes in them. An example of such an equivalence class can be found in Table 6.2. In this particular dataset, only one of the predicted labels was correct according to the given classifiers. For one paper it predicted the cs.CV category, which is correct according to arXiv, but not according to the included classifier. The classifier can only predict one category, while the actual arXiv entry can have multiple correct categories.

6.3.2 Accuracy of post-compression models

The final experiment is to then train graph neural networks on these compressed graphs. These models were described in Chapter 5.5.4. A summary of all results was provided in Table 6.3. We tested these models on the full graphs, not the compressed graphs.

In general, none of the post-compression models outperform the pre-compression models their training graph was generated by. There were some exceptions to this rule, for instance the M-post-arxiv-none-cec-wl-1 model beat the reference model M-arxiv-none-1 by 7.7%, reaching an accuracy of 54.5%. However, this model does not use any message passing, so the benefit comes purely from the fact that the post-compression model can work on top of predictions already done by the pre-compression model that generated the discretised dataset it was trained on. No interesting graph structural features are used. In fact, it performs 0.9% worse than its pre-compression model.

No improvement is nearly a best-case scenario. Out of the post-compression models that actually used message passing, and thus were graph neural networks, the biggest improvement over the pre-compression

Table 6.2: This table shows a sample equivalence class from COMP(discrete-ogbn-arxiv, 3, ∞). It includes the title of the paper, the prediction made by the pre-compression model, and thus its label in the discrete dataset, its ground truth label, and other categories the paper belongs to according to `arxiv.org`, but that it was not labelled with in the dataset.

		Ground truth	
Name	Predicted	Labelled	Others
Automatic detection of passable roads after floods in remote sensed and social media data	cs.CV	cs.IR	
FUNSD: A Dataset for Form Understanding in Noisy Scanned Documents	cs.CV	cs.IR	cs.CV, cs.LG
wMAN: Weakly-supervised Moment Alignment Network for Text-based Video Segment Retrieval	cs.CV	cs.CV	cs.LG
DialogueGCN: A Graph Convolutional Neural Network for Emotion Recognition in Conversation	cs.CV	cs.CL	cs.LG

equivalent was by the M-post-arxiv-sage-cec-bi-2 model, but with just a 0.9% improvement in accuracy, this can hardly be considered significant. This moderate increase in accuracy still landed it 12.9% behind the equivalent reference model, M-arxiv-sage-bi-2. Most post-compression models performed extremely similar to their pre-compression equivalents, generating the exact same class predictions. This was especially the case in the post-compression models trained on the discrete-deezer-europe dataset.

Not a single post-compression model that used message-passing was able to beat its reference equivalent, and which of the four loss functions was used did not have a significant impact on the performance of the models.

With the current methodology, it appears not possible to train graph neural networks on compressed graphs. Part of this might be due to the pre-compression step. A lot of nodes receive inaccurate discrete labels, which incorrectly get merged into the same equivalence class. Around 44% of node labels in the discrete-ogbn-arxiv dataset are incorrect, and the example in Table 6.2 shows that for at least some equivalence classes, that means the incorrect labels are many. In this example, during training, the post-compression model would have had to predict three different categories, despite all of these papers being represented by one node in the compressed graph.

When the trained model is evaluated on the full uncompressed graph, it then has to somehow use the now uncompressed (and thus correct) structural information to make up for these errors in the training data, but given it was not trained to do so, there should be no reason to expect it will.

In the future it might be beneficial to compress the train set of nodes differently from the test set of nodes, using the ground-truth labels instead of the predictions made by the pre-compression model. This might lower the amount of heterogeneous equivalence classes.

There is also a difference in training between the post-compression models and all other models. While post-compression models were tasked with predicting a probability distribution of classes, those of the nodes in every equivalence class, the other models simply had to predict a single class. This task is inherently more difficult, which might make it harder for the post-compression models to learn.

Model		Accuracy		
Post-compression	Reference	Post-compression	Reference	Difference
`M-post-arxiv-none-cec-bi-1`	`M-arxiv-none-1`	54.47%	46.82%	7.65%
`M-post-arxiv-none-cec-bi-2`	`M-arxiv-none-2`	55.37%	53.26%	2.11%
`M-post-arxiv-none-cec-bi-3`	`M-arxiv-none-3`	55.37%	52.65%	2.72%
`M-post-arxiv-none-ced-bi-1`	`M-arxiv-none-1`	48.42%	46.82%	1.60%
`M-post-arxiv-none-ced-bi-2`	`M-arxiv-none-2`	54.86%	53.26%	1.60%
`M-post-arxiv-none-ced-bi-3`	`M-arxiv-none-3`	51.08%	52.65%	-1.57%
`M-post-arxiv-none-kld-bi-1`	`M-arxiv-none-1`	52.79%	46.82%	5.96%
`M-post-arxiv-none-kld-bi-2`	`M-arxiv-none-2`	55.37%	53.26%	2.11%
`M-post-arxiv-none-kld-bi-3`	`M-arxiv-none-3`	55.37%	52.65%	2.72%

M-post-arxiv-none-jsd-bi-1	M-arxiv-none-1	49.22%	46.82%	2.40%
M-post-arxiv-none-jsd-bi-2	M-arxiv-none-2	55.35%	53.26%	2.09%
M-post-arxiv-none-jsd-bi-3	M-arxiv-none-3	54.71%	52.65%	2.06%
M-post-arxiv-gcn-cec-bi-1	M-arxiv-gcn-1	51.95%	54.12%	-2.16%
M-post-arxiv-gcn-cec-bi-2	M-arxiv-gcn-2	55.41%	70.02%	-14.62%
M-post-arxiv-gcn-cec-bi-3	M-arxiv-gcn-3	54.92%	70.64%	-15.72%
M-post-arxiv-gcn-ced-bi-1	M-arxiv-gcn-1	36.10%	54.12%	-18.02%
M-post-arxiv-gcn-ced-bi-2	M-arxiv-gcn-2	53.82%	70.02%	-16.20%
M-post-arxiv-gcn-ced-bi-3	M-arxiv-gcn-3	51.13%	70.64%	-19.51%
M-post-arxiv-gcn-kld-bi-1	M-arxiv-gcn-1	50.88%	54.12%	-3.24%
M-post-arxiv-gcn-jsd-bi-1	M-arxiv-gcn-1	48.26%	54.12%	-5.86%
M-post-arxiv-sage-cec-bi-1	M-arxiv-sage-1	55.25%	59.47%	-4.23%
M-post-arxiv-sage-cec-bi-2	M-arxiv-sage-2	56.28%	69.14%	-12.86%
M-post-arxiv-sage-cec-bi-3	M-arxiv-sage-3	55.56%	70.06%	-14.50%
M-post-arxiv-sage-ced-bi-1	M-arxiv-sage-1	52.63%	59.47%	-6.85%
M-post-arxiv-sage-ced-bi-2	M-arxiv-sage-2	54.50%	69.14%	-14.64%
M-post-arxiv-sage-ced-bi-3	M-arxiv-sage-3	49.97%	70.06%	-20.09%
M-post-arxiv-sage-kld-bi-1	M-arxiv-sage-1	54.39%	59.47%	-5.09%
M-post-arxiv-sage-kld-bi-2	M-arxiv-sage-2	56.14%	69.14%	-13.01%
M-post-arxiv-sage-kld-bi-3	M-arxiv-sage-3	56.05%	70.06%	-14.01%
M-post-arxiv-sage-jsd-bi-1	M-arxiv-sage-1	51.02%	59.47%	-8.46%
M-post-arxiv-sage-jsd-bi-2	M-arxiv-sage-2	56.15%	69.14%	-12.99%
M-post-arxiv-sage-jsd-bi-3	M-arxiv-sage-3	55.24%	70.06%	-14.83%
M-post-arxiv-none-cec-wl-1	M-arxiv-none-1	54.53%	46.82%	7.70%
M-post-arxiv-none-cec-wl-2	M-arxiv-none-2	55.37%	53.26%	2.11%
M-post-arxiv-none-cec-wl-3	M-arxiv-none-3	55.37%	52.65%	2.72%
M-post-arxiv-none-ced-wl-1	M-arxiv-none-1	48.45%	46.82%	1.62%
M-post-arxiv-none-ced-wl-2	M-arxiv-none-2	54.93%	53.26%	1.67%
M-post-arxiv-none-ced-wl-3	M-arxiv-none-3	50.65%	52.65%	-2.01%
M-post-arxiv-none-kld-wl-1	M-arxiv-none-1	52.80%	46.82%	5.98%
M-post-arxiv-none-kld-wl-2	M-arxiv-none-2	55.37%	53.26%	2.11%
M-post-arxiv-none-kld-wl-3	M-arxiv-none-3	55.37%	52.65%	2.72%
M-post-arxiv-none-jsd-wl-1	M-arxiv-none-1	49.57%	46.82%	2.75%
M-post-arxiv-none-jsd-wl-2	M-arxiv-none-2	55.37%	53.26%	2.11%
M-post-arxiv-none-jsd-wl-3	M-arxiv-none-3	54.71%	52.65%	2.06%
M-post-arxiv-gcn-cec-wl-1	M-arxiv-gcn-1	52.99%	54.12%	-1.12%
M-post-arxiv-gcn-cec-wl-2	M-arxiv-gcn-2	55.10%	70.02%	-14.92%
M-post-arxiv-gcn-cec-wl-3	M-arxiv-gcn-3	55.02%	70.64%	-15.62%
M-post-arxiv-gcn-ced-wl-1	M-arxiv-gcn-1	47.04%	54.12%	-7.08%
M-post-arxiv-gcn-ced-wl-2	M-arxiv-gcn-2	53.71%	70.02%	-16.32%
M-post-arxiv-gcn-ced-wl-3	M-arxiv-gcn-3	49.69%	70.64%	-20.95%
M-post-arxiv-gcn-kld-wl-1	M-arxiv-gcn-1	51.63%	54.12%	-2.49%
M-post-arxiv-sage-cec-wl-1	M-arxiv-sage-1	55.24%	59.47%	-4.23%
M-post-arxiv-sage-cec-wl-2	M-arxiv-sage-2	55.69%	69.14%	-13.46%
M-post-arxiv-sage-cec-wl-3	M-arxiv-sage-3	55.81%	70.06%	-14.26%
M-post-arxiv-sage-ced-wl-1	M-arxiv-sage-1	50.65%	59.47%	-8.83%
M-post-arxiv-sage-ced-wl-2	M-arxiv-sage-2	54.63%	69.14%	-14.51%
M-post-arxiv-sage-ced-wl-3	M-arxiv-sage-3	53.17%	70.06%	-16.89%
M-post-arxiv-sage-kld-wl-1	M-arxiv-sage-1	54.43%	59.47%	-5.04%
M-post-arxiv-sage-kld-wl-2	M-arxiv-sage-2	56.06%	69.14%	-13.09%
M-post-arxiv-sage-kld-wl-3	M-arxiv-sage-3	56.07%	70.06%	-13.99%
M-post-arxiv-sage-jsd-wl-1	M-arxiv-sage-1	51.70%	59.47%	-7.77%
M-post-arxiv-sage-jsd-wl-2	M-arxiv-sage-2	56.15%	69.14%	-12.99%
M-post-deezer-none-cec-bi-1	M-deezer-none-1	64.47%	64.51%	-0.04%
M-post-deezer-none-cec-bi-2	M-deezer-none-2	64.47%	65.00%	-0.53%
M-post-deezer-none-cec-bi-3	M-deezer-none-3	64.47%	64.98%	-0.51%
M-post-deezer-none-ced-bi-1	M-deezer-none-1	64.47%	64.51%	-0.04%
M-post-deezer-none-ced-bi-2	M-deezer-none-2	64.47%	65.00%	-0.53%

M-post-deezer-none-ced-bi-3	M-deezer-none-3	64.47%	64.98%	-0.51%
M-post-deezer-none-kld-bi-1	M-deezer-none-1	64.47%	64.51%	-0.04%
M-post-deezer-none-kld-bi-2	M-deezer-none-2	64.47%	65.00%	-0.53%
M-post-deezer-none-kld-bi-3	M-deezer-none-3	64.47%	64.98%	-0.51%
M-post-deezer-none-jsd-bi-1	M-deezer-none-1	64.47%	64.51%	-0.04%
M-post-deezer-none-jsd-bi-2	M-deezer-none-2	64.47%	65.00%	-0.53%
M-post-deezer-none-jsd-bi-3	M-deezer-none-3	64.47%	64.98%	-0.51%
M-post-deezer-gcn-cec-bi-1	M-deezer-gcn-1	54.75%	58.49%	-3.73%
M-post-deezer-gcn-cec-bi-2	M-deezer-gcn-2	59.04%	61.88%	-2.84%
M-post-deezer-gcn-cec-bi-3	M-deezer-gcn-3	56.41%	59.61%	-3.20%
M-post-deezer-gcn-ced-bi-1	M-deezer-gcn-1	54.88%	58.49%	-3.61%
M-post-deezer-gcn-ced-bi-2	M-deezer-gcn-2	58.85%	61.88%	-3.03%
M-post-deezer-gcn-ced-bi-3	M-deezer-gcn-3	57.96%	59.61%	-1.65%
M-post-deezer-gcn-kld-bi-1	M-deezer-gcn-1	58.95%	58.49%	0.47%
M-post-deezer-gcn-kld-bi-2	M-deezer-gcn-2	57.93%	61.88%	-3.95%
M-post-deezer-gcn-kld-bi-3	M-deezer-gcn-3	59.55%	59.61%	-0.06%
M-post-deezer-gcn-jsd-bi-1	M-deezer-gcn-1	58.93%	58.49%	0.45%
M-post-deezer-gcn-jsd-bi-2	M-deezer-gcn-2	51.65%	61.88%	-10.22%
M-post-deezer-gcn-jsd-bi-3	M-deezer-gcn-3	59.06%	59.61%	-0.55%
M-post-deezer-sage-cec-bi-1	M-deezer-sage-1	64.47%	64.43%	0.04%
M-post-deezer-sage-cec-bi-2	M-deezer-sage-2	64.47%	65.25%	-0.78%
M-post-deezer-sage-cec-bi-3	M-deezer-sage-3	64.47%	63.89%	0.57%
M-post-deezer-sage-ced-bi-1	M-deezer-sage-1	64.47%	64.43%	0.04%
M-post-deezer-sage-ced-bi-2	M-deezer-sage-2	64.47%	65.25%	-0.78%
M-post-deezer-sage-ced-bi-3	M-deezer-sage-3	64.47%	63.89%	0.57%
M-post-deezer-sage-kld-bi-1	M-deezer-sage-1	64.47%	64.43%	0.04%
M-post-deezer-sage-kld-bi-2	M-deezer-sage-2	64.47%	65.25%	-0.78%
M-post-deezer-sage-kld-bi-3	M-deezer-sage-3	64.47%	63.89%	0.57%
M-post-deezer-sage-jsd-bi-1	M-deezer-sage-1	64.47%	64.43%	0.04%
M-post-deezer-sage-jsd-bi-2	M-deezer-sage-2	64.47%	65.25%	-0.78%
M-post-deezer-sage-jsd-bi-3	M-deezer-sage-3	64.47%	63.89%	0.57%
M-post-deezer-none-cec-wl-1	M-deezer-none-1	64.47%	64.51%	-0.04%
M-post-deezer-none-cec-wl-2	M-deezer-none-2	64.47%	65.00%	-0.53%
M-post-deezer-none-cec-wl-3	M-deezer-none-3	64.47%	64.98%	-0.51%
M-post-deezer-none-ced-wl-1	M-deezer-none-1	64.47%	64.51%	-0.04%
M-post-deezer-none-ced-wl-2	M-deezer-none-2	64.47%	65.00%	-0.53%
M-post-deezer-none-ced-wl-3	M-deezer-none-3	64.47%	64.98%	-0.51%
M-post-deezer-none-kld-wl-1	M-deezer-none-1	64.47%	64.51%	-0.04%
M-post-deezer-none-kld-wl-2	M-deezer-none-2	64.47%	65.00%	-0.53%
M-post-deezer-none-kld-wl-3	M-deezer-none-3	64.47%	64.98%	-0.51%
M-post-deezer-none-jsd-wl-1	M-deezer-none-1	64.47%	64.51%	-0.04%
M-post-deezer-none-jsd-wl-2	M-deezer-none-2	64.47%	65.00%	-0.53%
M-post-deezer-none-jsd-wl-3	M-deezer-none-3	64.47%	64.98%	-0.51%
M-post-deezer-gcn-cec-wl-1	M-deezer-gcn-1	58.53%	58.49%	0.04%
M-post-deezer-gcn-cec-wl-2	M-deezer-gcn-2	61.43%	61.88%	-0.45%
M-post-deezer-gcn-cec-wl-3	M-deezer-gcn-3	59.78%	59.61%	0.17%
M-post-deezer-gcn-ced-wl-1	M-deezer-gcn-1	54.69%	58.49%	-3.80%
M-post-deezer-gcn-ced-wl-2	M-deezer-gcn-2	61.26%	61.88%	-0.62%
M-post-deezer-gcn-ced-wl-3	M-deezer-gcn-3	59.82%	59.61%	0.21%
M-post-deezer-gcn-kld-wl-1	M-deezer-gcn-1	58.72%	58.49%	0.23%
M-post-deezer-gcn-kld-wl-2	M-deezer-gcn-2	61.20%	61.88%	-0.68%
M-post-deezer-gcn-kld-wl-3	M-deezer-gcn-3	59.69%	59.61%	0.08%
M-post-deezer-gcn-jsd-wl-1	M-deezer-gcn-1	58.53%	58.49%	0.04%
M-post-deezer-gcn-jsd-wl-2	M-deezer-gcn-2	61.31%	61.88%	-0.57%
M-post-deezer-gcn-jsd-wl-3	M-deezer-gcn-3	59.89%	59.61%	0.28%
M-post-deezer-sage-cec-wl-1	M-deezer-sage-1	64.47%	64.43%	0.04%
M-post-deezer-sage-cec-wl-2	M-deezer-sage-2	64.47%	65.25%	-0.78%
M-post-deezer-sage-cec-wl-3	M-deezer-sage-3	64.47%	63.89%	0.57%

`M-post-deezer-sage-ced-wl-1`	`M-deezer-sage-1`	64.47%	64.43%	0.04%
`M-post-deezer-sage-ced-wl-2`	`M-deezer-sage-2`	64.47%	65.25%	-0.78%
`M-post-deezer-sage-ced-wl-3`	`M-deezer-sage-3`	64.47%	63.89%	0.57%
`M-post-deezer-sage-kld-wl-1`	`M-deezer-sage-1`	64.47%	64.43%	0.04%
`M-post-deezer-sage-kld-wl-2`	`M-deezer-sage-2`	64.47%	65.25%	-0.78%
`M-post-deezer-sage-kld-wl-3`	`M-deezer-sage-3`	64.47%	63.89%	0.57%
`M-post-deezer-sage-jsd-wl-1`	`M-deezer-sage-1`	64.47%	64.43%	0.04%
`M-post-deezer-sage-jsd-wl-2`	`M-deezer-sage-2`	64.47%	65.25%	-0.78%
`M-post-deezer-sage-jsd-wl-3`	`M-deezer-sage-3`	64.47%	63.89%	0.57%

Table 6.3: A table with all results. For every post-compression model, a reference model is shown too. The accuracy of both models and their difference in accuracy is given.

Chapter 7

Conclusion

In the writing of this thesis we learned a lot about graph learning, and especially graph neural networks, and both on a theoretical and a practical level.

We studied statistics that can be calculated or learned on graphs. We learned about alternatives to graph neural networks in the form of shallow embeddings, as well as graph neural networks themselves. We learned about the limitations in the expressiveness of graph neural networks, and exploited this to device a compression algorithm for graphs.

We gained practical experience in creating and training graph neural networks by applying the theoretical concepts to real experiments.

Even though we showed it is possible to train a graph neural network on a graph which has its labels replaced by discrete predictions made by a pre-compression model, this success did not repeat itself when training the post-compression models on the compressed graphs. While this for the most part, did not make the performance of the post-compression models worse when compared to the pre-compression models used to create their compressed datasets, it did stop any further improvements in accuracy being made, rendering the entire post-compression model redundant.

Part of this could be explained by the fact that the equivalence classes used during training the post-compression models are very heterogeneous. Another part could be that by requiring the post-compression model to predict accurate probability distributions as opposed to single classes, the task was too different from the pre-compression models and reference models.

7.1 Future work

While the results of the experiments presented in this thesis were discouraging, a lot of questions remain open for the future.

It is worth investigating how well the reference models would perform on the l-round c-graded compressed graphs. This could be done by for every node in the original graph, finding out what equivalence class it belongs to, and pairing it with the prediction of that equivalence class. The pre-compression model should not be able to distinguish this compressed graph from the original, and as such accuracy should not suffer.

Another direction to continue in is to study how much the post-compression model is handicapped by incorrect predictions of the pre-compression model. This could be done by substituting the incorrect predictions of the pre-compression model with the ground truths, at least those known in the training set, training new post-compression models on these ground truth based compressed datasets, and measuring the difference in accuracy.

Finally, a discrete dataset could be predicted or generated so no pre-compression model is needed any more. The discrete dataset could be compressed straight away, which would make it more fair to compare post-compression models and reference models.

Chapter 8

Dutch summary / Nederlandse samenvatting

In dit hoofdstuk voorzien we een Nederlandse samenvatting voor de thesis. We houden veelal dezelfde structuur als de rest van de thesis aan.

8.1 Introductie

Deer learning en artificiële intelligentie zijn veelbesproken onderwerpen vandaag de dag. Er bestaan veel verschillende soorten neurale netwerken die elk voorspellingen kunnen maken over een bepaalden typen data. Zo bestaan er convolutionele neurale netwerken die goed zijn in het verwerken van data in de vorm van afbeeldingen, en recurrente neurale netwerken die dan weer beter werken op tekstinvoer.

8.1.1 Neurale netwerken en deep learning

Deep learning is een aftakking van machine learning, en gebruikt neurale netwerken bestaande uit meerdere lagen om een voorstelling van de data te leren die bijzonder geschikt is om een bepaalde taak uit te voeren [LBH15]. Zo een model trainen gebeurt aan de hand van een voorbeeldfunctie, die voor een beperkte hoeveelheid datapunten de correcte voorstelling produceert. Het model zal deze functie leren nabootsen.

De vorm van de output data hangt af van het probleem dat het model hoort op te lossen. Classificatietaken zullen een model trainen om voor ieder datapunt een probabiliteitsdistributie over verschillende klassen te voorspellen. Een classificatietaak over afbeeldingen zal zo bijvoorbeeld moeten voorspellen of een foto een hond dan niet kat bevat.

Classificatietaken zijn het soort taken waar we ons op focussen in deze thesis. Er bestaan uiteraard ook andere soorten taken, denk maar aan gezichtsherkenning of het vertalen van teksten.

Er zijn ook verschillenden soorten neurale netwerken. Convolutionele neurale netwerken, zoals eerder vermeld, kunnen zeer goed omgaan met afbeeldingen, en recurrente neurale netwerken zijn dan weer sterker wanneer we te maken hebben met tekstinvoer.

8.1.2 Graaf neurale netwerken

Het soort data waar we ons in deze thesis mee bezighouden is graaf data. Veel wereldse objecten kunnen natuurlijk worden voorgesteld als grafen. Moleculen zijn niet slechts atomen, maar ook de bindingen tussen deze atomen. Een persoon op een sociaal netwerk is niet enkel zijn of haar profielfoto en biografie, maar ook zijn of haar vrienden en kennissen. Deze thesis kan worden voorgesteld door niet enkel de woorden zelf, maar ook alle literatuur die geciteerd wordt. De structuur van deze grafen omvat kritieke informatie over de objecten van deze grafen.

Wanneer we een klassiek neuraal netwerk trainen om de categorie van een academische paper te voorspellen, met enkel de titel en het abstract, dan zal de accuraatheid van dit netwerk redelijk goed zijn.

Dit is in de praktijk al gedaan, met een accuraatheid tot wel 61% over 40 categorieën [Hu+20]. Echter, wanneer we de structurele informatie erbij betrekken, zoals de citatiegraaf van de verschillende papers, dan is het mogelijk deze accuraatheid verder te verhogen. Dit is niet onlogisch, papers die elkaar citeren zullen waarschijnlijk ook in eenzelfde categorie behoren.

Graaf neurale netwerken zijn neurale netwerken die dit soort structurele informatie kunnen gebruiken om voorspellingen te maken. Voor de eerder besproken taak zal een graaf neuraal netwerk die de citatiegraaf van papers ter beschouwing neemt de accuraatheid kunnen verhogen van 61% naar wel 76% [Chi+21].

8.1.3 Grafen comprimeren

Al deze extra informatie kan niet zomaar zonder problemen worden gebruikt. Er kan veel informatie verstopt zitten in de structuur van een graaf, en al deze informatie gebruiken zal ook processorkracht vereisen. Dit is misschien niet zo een groot probleem voor het gebruiken van een model, maar voor zo een model te trainen zullen we vaak vele maal over alle data moeten itereren.

Ook zijn graaf neurale netwerken beperkt in hoe ze deze structurele data kunnen gebruiken. Ze zullen enkel de lokale structurele informatie voor iedere knoop gebruiken voor het maken van een voorspelling voor die knoop. Ook zullen ze niet alle vormen van structuur kunnen onderscheiden.

In deze thesis bestuderen we of we grafen kunnen comprimeren tot kleinere grafen, met een beperkte of zelf geen impact op de voorspellingskracht van een model getrainde op zo een gecomprimeerde graaf. We doen dit door een gecomprimeerde graaf op te stellen waarvan de structuur niet te onderscheiden is van de structuur van de originele graaf, of toch niet door een neuraal netwerk. Deze gecomprimeerde graaf kan dan gebruikt worden tijdens het trainen, wat het trainen significant zou kunnen versnellen.

8.1.4 Bijdragen

In deze thesis bestuderen we bestaande literatuur over de expressiviteit van een graaf neuraal netwerk. Vervolgens gebruiken we deze vergaarde kennis om een algoritme op te stellen dat een gecomprimeerde graaf kan bouwen die structureel erg lijkt op de originele graaf. Ten slotte presenteren we een reeks experimenten om de effectiviteit van het compressiealgoritme te toetsen.

8.2 Definities en notatie

Hier bespreken we kort de fundamentele definities en een reeks notaties die we gebruiken in deze thesis.

8.2.1 Grafen

We definiëren een graaf G als een tupel $G = (V, E, L)$ waar:

- V een set kopen is,
- $E \subseteq V^2$ een multiset gerichte bogen is,
- Σ een set labels is, en
- $L : V \to \Sigma$ een functie is die elke knoop naar een label vertaalt.

We gebruiken ook de volgende notaties wanneer we werken met grafen:

- $v \in G$ zegt dat er een knoop v bestaat in de graaf G,
- $(v_1 \to v_2) \in G$ zegt dat er in de graaf G een boog bestaat van een knoop v_1 naar een knoop v_2,
- $inv(v) = \{\!\{v_1 | (v_1 \to v) \in E\}\!\}$ is de multiset van bogen naar v, en
- $\text{indeg}(v) = |inv(v)|$ is het aantal bogen naar v.

Een multiset wordt voorgesteld als een normale set, maar met dubbele haakjes, bijvoorbeeld $\{\!\{1, 2, 2, 3, 3\}\!\}$. Voor de multipliciteit van een multiset te beperken gebruiken we een subscript notatie. Bijvoorbeeld, de multiset $\{\!\{1, 2, 2, 3, 3, 3\}\!\}|_{\leq 2}$ is gelijk aan de eerder voorgestelde multiset $\{\!\{1, 2, 2, 3, 3\}\!\}$.

8.2.2 Klasseerders

In deep learning is het belangrijk dat we een graaf kunnen opsplitsen naar een trainset, een validatieset en een testset. Aangezien we werken met classificatietaken in deze thesis, definiëren we een een klasseerder D als een functie $D : \text{dom}(V) \to \mathbb{Q}^d$ waar $\text{dom}(D) \subset V$ de set knopen is waarvoor de klasseerder is gedefinieerd, d het aantal klassen is, en $\mathbb{Q}^d$ een probabiliteitsdistributie over de verschillende klassen is. Voor de datasets in de thesis is de klasse waartoe elke knoop behoord gekend, dus zal deze probabiliteitsdistributie een vector zijn met de waarde 1 voor de correcte klasse, en 0 voor alle andere klassen. Iedere graaf zal zo drie klasseerders hebben: D_{train} bevat de traindata, D_{valid} de validatie data en D_{test} de testdata.

8.3 Graaf neurale netwerken

De informatie in dit onderdeel komt voornamelijk uit “Graph representation learning” van Hamilton [Ham20].

Een graaf neuraal netwerk maakt gebruik van een message-passing algoritme. Het netwerk begint met een reeks voorstellingen van knopen, 1 per knoop. Elke laag van het graaf neuraal netwerk zal dan een aggregatiestap gevolgd door een update stap doen. De aggregatiestap zal de de voorstelling van de huidige knoop en al zijn buren combineren, en de update stap zal dan een nieuwe voorstelling bouwen.

We noteren de voorstelling voor een knoop v die we bekomen na n lagen in het netwerk als h_v^n. We definiëren deze inductief.

$$\begin{aligned} h_v^0 &= \text{een initiële voorstelling van een knoop} \\ h_v^n &= \text{upd}_n(h_v^{n-1}, \text{aggr}_n(v)) \end{aligned}$$

Er bestaan een reeks verschillende invullingen voor de aggregatiestap en de updatestap. We beschrijven in deze samenvatting slechts de meest fundamentele vormen. Voor beide.

De simpelste aggregatiestap is simpelweg de sommatie van de voorstellingen van alle buren, zoals in Formule 8.1.

$$h_v^n = \text{upd}_n(h_v^{n-1}, \sum_{(u \to v) \in E} h_u^{n-1}) \tag{8.1}$$

We kunnen deze aggregatie ook normaliseren. Een manier om dit te doen is door symmetische normalisatie, zoals in Formule 8.2.

$$h_v^n = \text{upd}_n\left(h_v^{n-1}, \sum_{(u \to v) \in E} \frac{h_u^{n-1}}{\sqrt{\text{indeg}(v) \cdot \text{indeg}(u)}}\right) \tag{8.2}$$

De simpelste update stap is dan gewoon de huidige voorstelling van de huidige knoop optellen bij de geaggregeerde voorstelling van alle buren, zoals voorgesteld in Formule 8.3. We vermenigvuldigen ook beiden met een leerbare matrix W_{self}^{n-1} en W_{neigh}^{n-1}, en voegen een biasterm b_n toe.

$$h_v^n = \sigma\left(W_{self}^{n-1} \times h_v^{n-1} + W_{neigh}^{n-1} \times \text{aggr}_n(v) + b_n\right) \tag{8.3}$$

Deze updatefunctie kan nog vereenvoudigd worden door de voorstelling van de huidige knoop te behandelen in de aggregatiestap, door een virtuele zelfboog toe te voegen, en deze dan niet meer apart te beschouwen in de updatefunctie. Combineren we dit met symmetrische normalisatie, krijgen we een graaf convolutioneel netwerk. zoals beschreven in Formule 8.4.

$$h_v^n = \sigma\left(W_{neigh}^{n-1} \times \sum_{(u \to v) \in E} \frac{h_u^{n-1}}{\sqrt{\text{indeg}(v) \cdot \text{indeg}(u)}} + b_{n-1}\right) \tag{8.4}$$

8.4 Expressiviteit van graaf neurale netwerken

Het Weisfeiler-Lehman isomorfisme algoritme tracht te testen of twee grafen al dan niet isomorf zijn. Twee grafen $G_1 = (V_1, E_1, L_1)$ en $G_2 = (V_2, E_2, L_2)$ zijn isomorf als er een bijectieve functie $f : V_1 \to V_2$ bestaat waarvoor de volgende twee eigenschappen gelden [HHH06]:

1. **Label-gelijkheid**: $\forall v_1 \in V_1 : L(v_1) = L(f(v_1))$.
2. **Boog-gelijkheid**: $\forall (v_1, u_1) \in E_1 : (f(v_1), f(u_1)) \in E_2$.

De Weisfeiler-Lehman test of twee grafen al dan niet isomorf kunnen zijn door een canonieke labelling te maken van beide grafen [She+11; WL68]. Twee grafen slagen de test als ze dezelfde canonieke labelling hebben, en falen de test als ze dit niet hebben. Wanneer twee grafen de test falen kunnen ze nooit isomorf zijn, maar slechts slagen op de test is niet genoeg om effectief isomorf te zijn. In sommige gevallen kunnen twee grafen slagen voor de Weisfeiler-Lehman isomorfisme test, en toch niet isomorf zijn.

Het algoritme maakt een canonieke labelling door een sequentie van functies $K_0^G, K_1^G, K_2^G, ..., K_\infty^G$ te berekenen, waarbij K_i^G alle knopen in G vertaalt naar "canonieke labels" uit het alfabet T_∞. In de praktijk zal T_∞ gewoon de set natuurlijke getallen zijn. We kunnen deze functies berekenen als beschreven in Formule 8.5. Het is hier belangrijk dat de functie hash een injctieve functie is met codomein T_∞.

$$\begin{aligned} K_0^G(v) &= \text{hash}(L(v)) \\ K_i^G(v) &= \text{hash}\left(K_{i-1}^G(v), \{\!\{K_{i-1}^G(u) | u \in inv(v)\}\!\}\right) \end{aligned} \tag{8.5}$$

Wanneer een labelling K_l^G gelijk is aan een labelling K_{l+1}^G, dan zeggen we dat de labelling K_l^G stabiel is. We definiëren dat $K_\infty^G = K_l^G$ met K_l^G stabiel. Zo een K_l^G zal altijd bestaan.

Wanneer K_l^G en K_l^H aan elkaar gelijk zijn, slagen grafen G en H voor de Weisfeiler-Lehman isomorfisme test. We zeggen dat deze grafen niet te onderscheiden zijn door Weisfeiler-Lehman.

We kunnen echter het begrip isomorfisme verder generaliseren. Gegeven twee grafen $G = (V_G, E_G, L_G)$ en $H = (V_H, E_H, L_H)$ waar L_G en L_H eenzelfde codomein Σ delen, zeggen we dat twee gewortelde grafen (G, v) and (H, u) **c-graded, l-rooted bisimilair** zijn, genoteerd als $(G, v) \sim_{\#}^{c,l} (H, u)$, wanneer er aan de volgende vereisten voldaan wordt [Ott19]:

- **label-equivalentie**: $L_G(v) = L_H(u)$

Wanneer $l > 0$ moet er ook aan de volgende vereisten voldaan worden:

- **c-graded voorwaarts**: $\forall k \in \mathbb{N}, k \leq min\left(c, \text{indeg}_G(v)\right)$ en alle paargeweis-verschillende $v_1...v_k \in \text{inv}_g(v)$ zijn er paargeweis-verschillende $u_1...u_k \in inv_H(u)$ zodat $\forall i \in \mathbb{N}, i \leq k : (G, v_i) \sim_{\#}^{c,l-1} (H, u_i)$.
- **c-graded achterwaarts**: $\forall k \in \mathbb{N}, k \leq min\left(c, \text{indeg}_H(u)\right)$ en alle paargeweis-verschillende $u_1...u_k \in inv_H(v)$ zijn er paargeweis-verschillende $v_1...v_k \in \text{inv}_g(v)$ zodat $\forall i \in \mathbb{N}, i \leq k : (H, u_i) \sim_{\#}^{c,l-1} (G, v_i)$.

Wanneer er een bijectieve functie $f : V_G \to H_G$ bestaat zodat $\forall v_G \in V_G : (G, v_G) \sim_{\#}^{c,l} (H, f(v_G))$, en omgekeerd $\forall v_H \in V_H : (G, f(v_H)) \sim_{\#}^{c,l} (H, v_U)$, zeggen we dat beide grafen c-graded l-round bisimilair zijn. Wanneer $l = 1$ en $c = 1$, zeggen we dat de twee grafen **bisimilair** zijn.

Twee grafen zijn c-graded l-round bisimilair als en slechts als ze niet te onderscheiden zijn door Weisfeiler-Lehman.

We definiëren een procedure LAB(G, l, c) die l-round c-graded labellings produceert.

$$\begin{aligned} \text{LAB}(G, 0, c)(v) &= \text{hash}(L(v)) \\ \text{LAB}(G, i, c)(v) &= \text{hash}\left(\text{LAB}(G, i-1, c)(v), \{\!\{\text{LAB}(G, i-1, c)(u) | u \in inv(v)\}\!\}|_{\leq c}\right) \end{aligned} \tag{8.6}$$

De expressiviteit van een graaf neuraal netwerk is sterk gekoppeld aan de Weisfeiler-Lehman isomorfisme test. De test lijkt erg fel op het message-passing algoritme, maar gebruikt een hash functie in plaats van een *upd* functie. Het is zo dat het message-passing algoritme slechts zo expressief is als de Weisfeiler-Lehman isomorfisme test wanneer de *upd* functie zich gedraagt als een hash functie [Sat20].

Twee grafen die niet te onderscheiden zijn door Weisfeiler-Lehman zullen ook niet te onderscheiden zijn door message-passing. Als de Weisfeiler-Lehman isomorfisme test dezelfde labellings zal toewijzen aan twee knopen na l ronden, dan zal een graaf neuraal netwerk met l message-passing lagen dezelfde voorstelling toewijzen aan deze twee knopen [Sat20].

8.5 Compressie

We kunnen de vergaarde kennis over de expressiviteit van een graaf neural netwerk gebruiken om een compressiealgoritme op te stellen die een gecomprimeerde graaf maakt die moeilijk te onderscheiden is van de originele graaf. Echter, we moeten niet enkel de graaf comprimeren, maar ook de bijhorende klasseerders, anders zal het niet mogelijk worden een graaf neuraal netwerk op deze geocomprimeerde graaf te trainen.

8.5.1 Graaf compressie

We kunnen een gecomprimeerde graaf opstellen door elke twee knopen die niet te onderscheiden zijn door Weisfeiler-Lehman te vervangen door een knoop die beide knopen vertegenwoordigd. We zeggen dat twee knopen tot dezelfde equivalentieklasse behoren als en slechts als ze beiden dezelfde labelling hebben in $\text{LAB}(G, l, c)$. We definiëren de functie $\text{eqclass}_{K^G}(v)$ voor een labelling K^G. Ze berekent de grootste subset S bestaande uit knopen uit de graaf G zodat $\forall v \in S, \forall u \in V_G : K^G(v) = K^G(u) \implies u \in S$.

We definiëren een algoritme $\text{COMP}(G, l, c)$ dat een graaf $G = (V, E, L)$ comprimeert tot een kleinere graaf $G' = (V', E', L')$. Grafen G en G' zijn l-round, c-graded bisimilair. De Nederlandstalige definitie van dit algoritme wordt gegeven in Algoritme 2.

Algorithm 2 Graaf compressie

```
input: Een graaf $G = (V, E, L)$, $l$, $c$
output: Een gecomprimeerde graaf $G' = (V', E', L')$

1:  function COMP(G, l, c)
2:      $V' \leftarrow \{\}$
3:      $E' \leftarrow \{\!\{\}\!\}$
4:      $K \leftarrow LAB(G, l, c)$                    ▷ Een canonieke labelling voor de graaf.
5:      $S \leftarrow \{\}$                            ▷ Een set equivalentieklassen.
6:
7:      for $v \in V$ met $indegree(v)$ van laag naar hoog do
8:          if $K(v) \notin S$ then      ▷ Kies een vertegenwoordiger voor iedere equivalentieklasse.
9:              $V' \leftarrow V' \cup \{v\}$
10:             $S \leftarrow K(v)$
11:         end if
12:     end for
13:
14:     $L' \leftarrow L$, met het domein beperkt tot $V'$
15:
16:     for $(u', (v \to u')) \in V' \times E$ do
17:         Zoek $v' \in V'$ zodat $K(v) = K(v')$.   ▷ $v$ en $v'$ tot dezelfde equivalentieklasse behoren.
18:         $E' \leftarrow E' \cup \{\!\{(v', u')\}\!\}$
19:     end for
20:
21:     $E' \leftarrow E'|_{\leq c}$
22:     $G' \leftarrow (V', E', L')$
23:     return $G'$
24: end function
```

Het algoritme kiest voor iedere equivalentieklasse een vertegenwoordiger $u' \in V$. Vervolgens itereren we, voor iedere vertegenwoordiger u', over alle inkomende buren $v \in V$, en vinden we de vertegenwoordiger

v' van de equivalentieklasse waar v tot behoort. We voegen de boog $v' \to u'$ toe aan de gecomprimeerde graaf. Ten slotte beperken we de multipliciteit van de bogen van de gecomprimeerde graaf tot c. Dit kan gedaan worden zonder dat de l-round c-graded bisimilariteit tussen de ongecomprimeerde graaf en de gecomprimeerde graaf breekt.

We kiezen steeds de knoop met de minste inkomende buren als vertegenwoordiger voor een equivalentieklasse. Deze heuristiek levert vaker een zo klein mogelijke graaf op, maar is niet altijd een optimale oplossing.

Iedere knoop in de gecomprimeerde graaf G vertegenwoordigd een equivalentieklasse van knopen in de ongecomprimeerde graaf G.

8.5.2 Compressie van de klasseerders

Willen we een graaf neuraal netwerk trainen op een gecomprimeerde graaf, hebben we ook een aantal klasseerders nodig voor deze gecomprimeerde graaf.

We kunnen een klasseerder voor een ongecomprimeerde graaf vertalen naar een klasseerder voor een gecomprimeerde graaf. Gegeven een klasseerder D en een gecomprimeerde graaf G' definiëren we een gecomprimeerde klasseerder $D' : V \to \mathbb{Q}^d$. Voor elke knoop $v \in G'$ definiëren we $D'(v)$ gelijk aan de probabiliteitsdistributie over $\{\!\{D(v) | v \in \text{eqclass}_{K^G}(v')\}\!\}$.

8.6 Methodologie

In de thesis proberen we beperkingen in de expressiviteit van een graaf neuraal netwerk te gebruiken om grote grafen te comprimeren tot kleine grafen waarop er makkelijk geleerd kan worden, zonder relevantie informatie te verliezen.

We bespreken de gebruikte datasets, de gebruikte modellen, en de trainparameters.

8.6.1 Datasets

Voor de experimenten gebruikten we twee datasets: ogbn-arxiv [Hu+20] en deezer-europe Rozemberczki and Sarkar.

De ogbn-arxiv dataset is een graaf. De knopen van de graaf zijn papers, met als labels een vector embedding met grootte 128, van hun titel en hun abstract. Elke paper heeft een boog naar een ander paper die het citeert. De klasseerders horende bij deze graaf proberen te voorspellen tot welke categorie op `arxiv.org` een paper behoort. Er zijn zo 40 verschillende categoriën in totaal.

De deezer-europe dataset heeft als knopen de verschillende gebruikers van Deezer; dit is een muziekstreamingdienst. Elke gebruiker heeft een boog vanuit elke andere gebruiker die hem of haar volgt. De gebruikers hebben als label een vector embedding met als grootte 128 componenten van de artiesten die ze volgen. De taak die bij deze dataset hoort is het voorspellen van het geslacht van elke gebruiker, man of vrouw.

Deze datasets kunnen niet onmiddellijk gecomprimeerd worden. De labels van de nodes zijn allen uniek, waardoor dat ze door het compressiealgoritme ook allen een unieke labelling krijgen toegewezen. We moeten eerst zorgen dat de labels van de datasets discreet zijn. Dit doen we door eerst een klassiek neuraal netwerk te bouwen dat al tracht de categorieën van elke knoop te voorspellen, en dan een nieuwe gediscretiseerde graaf op te bouwen, identiek aan de originele dataset, maar met voor elke knoop zijn voorspelde categorie. We noemen deze datasets discrete-ogbn-arxiv en discrete-deezer-europe voor beide datasets respectievelijk.

Beide gediscretiseerde modellen werden gecomprimeerd door $\text{COMP}(G, l, c)$ op zes verschillende manieren: met invullingen 1, 2 en 3 voor l en invullingen 1 en ∞ voor c.

8.6.2 Model opbouw

Hoewel we in de experimenten klassieke neurale netwerken en graaf neurale netwerken gebruiken, stellen we één structuur voor waar alle modellen onder vallen. Deze structuur is gebasseerd op de modellen gebruikt door Hu et al. [Hu+20]. Ieder model zal een reeks message-passing lagen hebben, gevolgd door een finale laag die de uitvoer transformeert naar een probabiliteitsdistributie.

Concreet zal iedere message-passing laag een aggregatie- en updatestap doen, gevolgd door een batch-normalisatie en een ReLU activatie stap. De batch-normalisatie zal zorgen dat de standaardafwijking tussen alle componenten van alle uitvoeren van de aggregatiestap gelijk is. Dit versnelt het leren. Welke aggregatiestap er gebruikt wordt hangt af van het type netwerk. We beschouwen in deze thesis een graaf convolutionele (GCN) stap, een GraphSAGE stap [HYL17], en een normale lineaire transformatie. Dit laatste gebruikt geen booginformatie, en dus zal ieder neural netwerk dat dit model volgt met een lineaire transformatie als aggregatie- en updatestap dus geen graaf neuraal netwerk zijn.

8.6.3 Loss functies

We gebruiken vier verschillende loss functies tijdens het trainen. De cross-entropy wordt gedefinieerd in Formule 8.7. We beschouwen twee varianten van deze loss functies: een variant die werkt met volwaardige probabiliteitsdistributies, en een variant die werkt met een probabiliteitsdistributie waarvan alle componenten 0 zijn, behalve de ground-truth. We zoemen ze deze loss functies de cross-entropy loss en de cross-entropy klasseerder loss.

we gebruiken ook de Kullback-Leibler divergence loss functie zoals gedefinieerd in Formule 8.9, en de Jensen-Shannon divergence loss functie zoals gedefinieerd in Formule 8.10. De Jensen-Shannon divergence loss functie is een variant van de Kullback-Leibler divergence loss functie, maar geeft hetzelfde resultaat als we de ground-truth en de voorspelling zouden verwisselen.

$$\mathcal{L}_{ce}(p,q) = -\sum_{i=1}^{n} \log(p_i) \cdot q_i \tag{8.7}$$

$$kl(p,q) = \sum_{i=1}^{n} p_i \cdot \log\left(\frac{p_i}{q_i}\right) \tag{8.8}$$

$$\mathcal{L}_{kl}(p,q) = kl(q,p) \tag{8.9}$$

$$m = \frac{(p+q)}{2}$$

$$\mathcal{L}_{js}(p,q) = \frac{kl(p,m) + kl(q,m)}{2} \tag{8.10}$$

Omdat onze ground-truth vaak veel 0-componenten zal bevatten, namelijk voor alle klassen die niet correct zijn, stellen we dat alle termen van de sommatie in Formule 8.8 waarbij $p_i = 0$, de hele term 0 bedraagt.

8.6.4 Varianten van modellen

Voor de experimenten in deze thesis worden er veel modellen gebruikt. We kunnen ze indelen in drie categoriën:

- Referentiemodellen: Deze modellen worden getraind op de originele, ongecomprimeerde grafen. Ze dienen als een baseline om de accuraatheid die we kunnen behalen op de datasets te toetsen.
- Discrete Referentiemodellen: Deze modellen worden getraind op de gediscretiseerde maar ongecomprimeerde grafen. Ze dienen om te testen of trainen op gediscretiseerde grafen überhaupt mogelijk is.
- Precompressiemodellen: Deze modellen worden getraind op de originele grafen, en hun voorspellingen worden gebruikt om de discrete datasets op te bouwen.
- Postcomressiemodellen: Deze modellen worden getraind op de gecomprimeerde grafen. Wanneer deze modellen even goed of beter presteren dan de discrete referentiemodellen, dan weten we dat het compressiealgoritme mogelijk nuttig is.

8.7 Implementatie

De modellen werden geïmplementeerd in PyTorch [Pas+19]. Dit is een Python library voor het maken van neurale netwerken. PyG, een extentie van PyTorch, is een library voor het maken van graaf neurale netwerken [FL19].

Het trainen gebeurde op de infrastructuur van het Vlaams Supercomputer Centrum (VSC)[1].

Het compressiealgoritme werd geïmplementeerd in de Rust programmeertaal[2]. Een reeks libraries werden gebruikt, waaronder ndarray[3] en ndarray-npy[4] voor het inlezen en uitschrijven van datasets op een manier dat Python libraries begrijpen. De dashmap[5] en rayon[6] libraries werden gebruikt voor het parallelliseren van het algoritme.

8.8 Resultaten

We overlopen de resultaten van de experimenten.

8.8.1 Referentiemodellen

De experimenten met de referentiemodellen zijn belangrijk want ze bevestigen dat de opbouw van de modellen correct is. We vonden dat de referentiemodellen aardig presteerde wanneer we ze vergeleken met andere papers getraind op dezelfde datasets.

Ook willen we bevestigen dat het gebruiken van een graaf neuraal netwerk wel degelijk betere resultaten oplevert dan het gebruiken van een normaal neuraal netwerk. Dit was niet altijd het geval. Hoewel we voor de ogbn-arxiv dataset een verbetering zagen met 18% accuraatheid door het gebruiken van een graaf neuraal netwerk met 3 lagen, voor zowel GCN en GraphSAGE, was er helemaal geen verbetering voor de deezer-europe dataset. Dit maakt al onmiddellijk duidelijk niet alle grafen nuttige data voor een graaf neuraal netwerk bevatten in hun structuur.

8.8.2 Discrete referentiemodellen

De discrete referentiemodellen werden getraind op de gediscretiseerde graaf. Dit betekent dus dat er eerst een klassiek neuraal netwerk wordt getraind die de graaf al probeert de klasseren, en deze voorspellingen dan gebruikt worden om een gediscretiseerde graaf op te stellen waar we een graaf neuraal netwerk op trainen. De vraag is dan of deze aanpak nog steeds even goede resultaten oplevert als de referentiemodellen, en we vonden ook dat dit het geval is. Deze modellen presteren even goed als de referentiemodellen.

8.8.3 Compressie

De volgende stap is het comprimeren van de grafen, en kijken in welke maten het comprimeren van grafen mogelijk is.

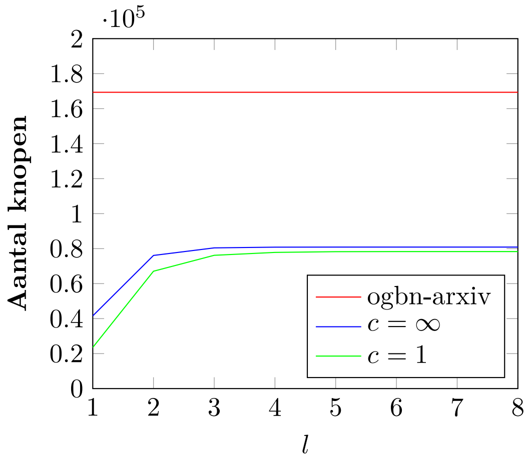
Comprimeren we de getrainde grafen, dan vinden we dat compressie mogelijk is, maar niet even goed werkt tussen verschillende datasets. In Tabel 8.1 geven we een overzicht van het aantal knopen en bogen in de gecomprimeerde grafen.

In het algemeen is de ogbn-arxiv dataset beter comprimeerbaar dan de deezer-europe dataset. De deezer-europe dataset is bijna niet comprimeerbaar, en de enige compressie die er gebeurt is het gevolg van mathematische beperkingen. Bijvoorbeeld, voor 1-graded 1-round bisimulatie wordt de volledige deezer-europe dataset gecomprimeerd tot slechts 8 knopen. Dit is het maximum aantal knopen dat er kan bestaan in zo een dataset, de dataset heeft immers maar twee mogelijke labels, man of vrouw. Elk van deze labels kan dus ofwel verbonden zijn met een man, ofwel een vrouw, ofwel beiden, of wel geen van beiden. Dat zijn vier mogelijke combinaties, en twee mogelijke geslachten, wat een maximum van 8 verschillende equivalentieklassen maakt.

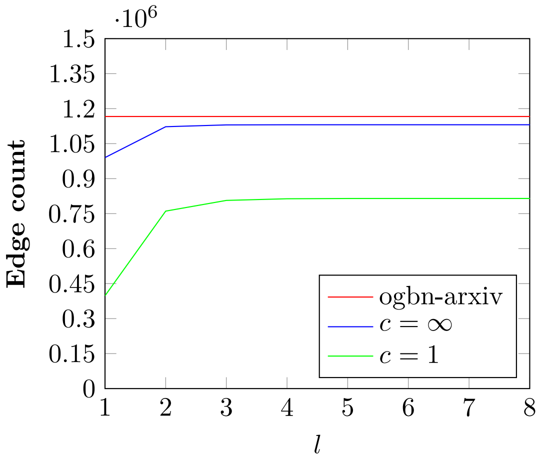
[1] `https://www.vscentrum.be/`
[2] `https://www.rust-lang.org/`
[3] `https://github.com/rust-ndarray/ndarray`
[4] `https://github.com/jturner314/ndarray-npy`
[5] `https://github.com/xacrimon/dashmap`
[6] `https://github.com/rayon-rs/rayon`

Figure 8.1: Het aantal nodes en edges in elke gecomprimeerde dataset, na l compressieronden en voor $c \in \{1, \infty\}$.

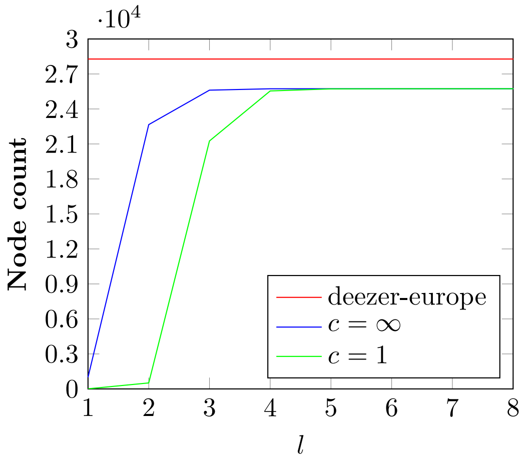
(a) Het totaal aantal knopen in de gecomprimeerde dataset COMP(discrete-ogbn-arxiv, l, c) vergeleken met het aantal knopen in ogbn-arxiv.



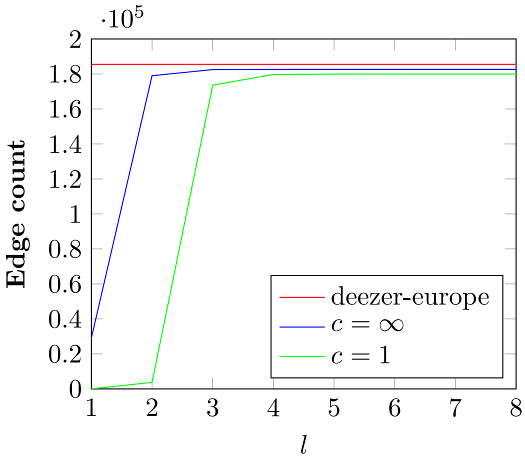
(b) Het totaal aantal bogen in de gecomprimeerde dataset COMP(discrete-ogbn-arxiv, l, c) vegeleken met het aantal knopen in ogbn-arxiv.



(c) Het totaal aantal knopen in de gecomprimeerde dataset COMP(discrete-ogbn-arxiv, l, c) vergeleken met het aantal knopen in deezer-europe.



(d) Het totaal aantal bogen in de gecomprimeerde dataset COMP(discrete-ogbn-arxiv, l, c) vegeleken met het aantal knopen in deezer-europe.



Wanneer we de gecomprimeerde grafen van iets dichter bij bestuderen vinden we al snel dat de meeste equivalentieklassen enorm klein zijn. Voor de graaf $\text{COMP}(\text{discrete-ogbn-arxiv}, 3, \infty)$ behoren 46% van alle knopen tot een equivalentieklasse met maar één lid. Voor deze knopen is er dus volledig geen compressie.

Kijken we naar de compositie van de equivalentieklassen vinden we dat ze ook erg heterogeen zijn. De labels van de knopen die we gebruiken om de equivalentieklassen op te bouwen zijn niet allemaal correct, immers als ze dat wel waren was er geen reden meer om de grafen te comprimeren en een model op deze gecomprimeerde graaf te trainen. Dit zorgt dat er vaak knopen met een ander ground-truth label tot eenzelfde equivalentieklasse gaan behoren, omdat het precompressiemodel eenzelfde voorspelling voor hun had gedaan. Dit kan uiteraard de leerperformantie negatief beïnvloeden.

8.8.4 Postcompressiemodellen

Ten slotte trainde we een reeks modellen op de gecomprimeerde grafen. We testten deze modellen op de volledige, originele grafen, niet op de gecomprimeerde grafen.

We vonden dat de postcompressiemodellen en pak slechter presteerden dan de referentiemodellen, en vaak zelfs niet beter presteerden dan de precompressiemodellen. Als we de postcompressiemodellen beschouwen die een graaf neuraal netwerk gebruiken, dan vinden we de beste verbetering over het precompressiemodel bij een GraphSAGE model van 2 lagen diep, getraind met een cross-entropy klasseerder loss op een $\text{COMP}(\text{discrete-ogbn-arxiv}, 2, 1)$ gecomprimeerde graaf. Echter, deze verbetering is slechts 0.9%, en nog steeds 12% minder dan een equivalent referentiemodel. Geen enkel postcompressiemodel dat gebruik maakte van een graaf neuraal netwerk deed het beter dan een equivalent referentiemodel.

Het lijkt dus onmogelijk een graaf te comprimeren volgens de huidige methodologie en deze graaf toch nog te kunnen gebruiken om een model te trainen. Hier kunnen verschillende oorzaken voor zijn. Zo is er uiteraard het probleem dat veel knopen in een 'foute' equivalentieklasse terechtkomen, met veel andere knopen met een andere ground truth. Dit komt omdat het precompressiemodel geen correcte voorspelling kan maken voor 100% van de knopen. Het precompressiemodel verbeteren is hier ook geen goede optie, want dan zou er geen postcompressiemodel meer nodig zijn. Wel kan er wellicht gezocht worden naar een dataset waar de labels van iedere knoop al discreet zijn, om zo de graaf te kunnen comprimeren zonder heterogene equivalentieklassen te krijgen. De ogbn-arxiv dataset heeft verder ook nog een probleem dat veel papers in deze dataset eigenlijk tot meerdere categorieën behoren, maar in de dataset werd er steeds maar éém vam deze categorieën gebruikt als ground-truth. Het precompressiemodel voorspelde daardoor vaak een categorie die wel juist was, maar niet volgens de ground-truth.

Ook kunnen we voor de knopen in de trainset de voorspellingen van het precompressiemodel vervangen door de correcte klassen. Dit kan verder de heterogeniteit van de equivalentieklassen verlagen.

Ook is er een verschil in het trainen tussen de postcompressiemodellen en alle andere modellen. De postcompressiemodellen moeten een volwaardige probabiliteitsdistributie proberen te voorspellen, terwijl alle andere modellen slechts een enkele klasse moeten proberen te voorspellen, die wordt voorgesteld door een probabiliteitsdistributie. Het kan zijn dat deze taak gewoon inherent moeilijker is.

Ten slotte is het ook zo dat het postcompressiemodel getraind wordt op een gecomprimeerde dataset waar niet de volledige graafstructuur correct zal zijn. Ze wordt echter getest op de volledige graaf, waar de structuur volledig correct is. Hoewel de gecomprimeerde graaf voor een graaf neuraal netwerk zeer moeilijk te onderscheiden zou moeten zijn, kan het best dat het verschil in datasets gewoon te groot voor het postcompressiemodel om de informatie in de ongecomprimeerde, gediscretiseerde graaf informatie correct te kunnen gebruiken.

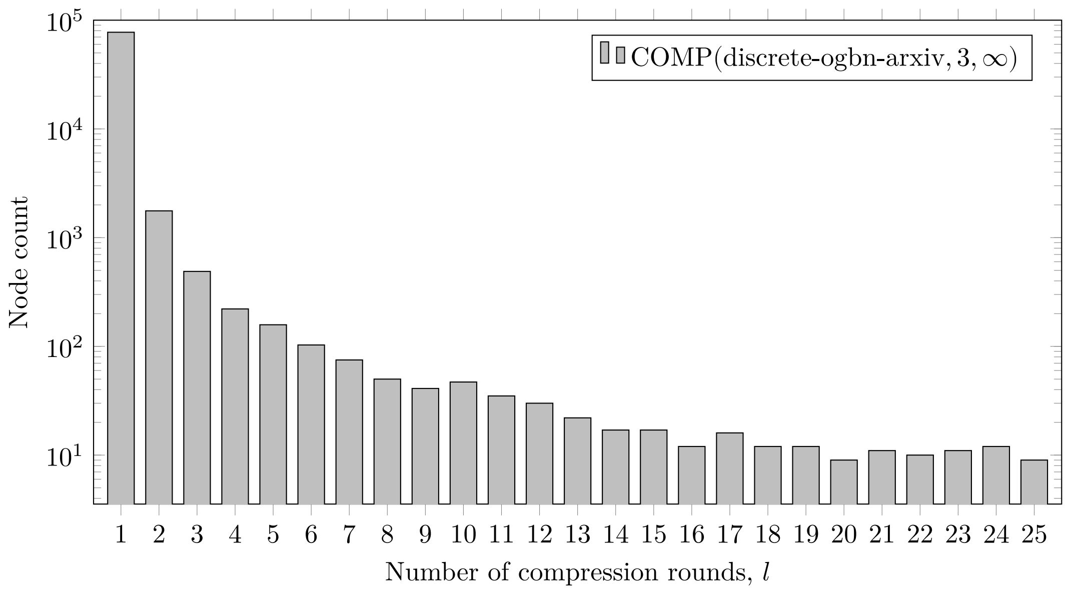
Een overzicht van alle postcompressiemodellen is gegeven in Tabel 6.3.

8.9 Conclusie

In het schrijven bouwde we veel ervaring op met graaf neurale netwerken, op een theoretisch en praktisch niveau.

De experimenten leerden ons dat graaf compressie zoals voorgesteld in de thesis niet voldoende werkt om effectief bruikbaar te zijn. De voornamelijkste oorzaken zijn waarschijnlijk de heterogeniteit van de equivalentieklassen die gebruikt worden tijdens het comprimeren, die zelf dan weer het resultaat zijn

Figure 8.2: De grootte van iedere equivalentieklasse tijdens het comprimeren van een graaf, op een logaritmische schaal.



van incorrecte voorspellingen door het precompressiemodel, en beperkingen in de datasets, die slechts 1 categorie kunnen toewijzen per paper, ook voor papers die tot meerdere categorieën behoren.

Er is nog veel onderzoek mogelijk naar graaf compressie:

De impact van incorrecte voorspellingen door het precompressiemodel kan onderzocht worden door een graaf te gebruiken met correcte en discrete labels, en hierop postcompressiemodellen te trainen.

Ook kan er gekeken worden naar hoe goed de referentiemodellen presteren op de gecomprimeerde grafen. De neurale netwerken zouden geen verschil moeten kunnen zien tussen de originele grafen en l-round c-graded gecomprimeerde grafen.

Bibliography

[WL68] Boris Weisfeiler and Andrei Leman. “The reduction of a graph to canonical form and the algebra which appears therein”. In: *NTI, Series* 2.9 (1968), pp. 12–16.

[HHH06] Shu-Ming Hsieh, Chiun-Chieh Hsu, and Li-Fu Hsu. “Efficient method to perform isomorphism testing of labeled graphs”. In: *International Conference on Computational Science and Its Applications*. Springer. 2006, pp. 422–431.

[She+11] Nino Shervashidze et al. “Weisfeiler-lehman graph kernels.” In: *Journal of Machine Learning Research* 12.9 (2011).

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), pp. 436–444.

[HYL17] Will Hamilton, Zhitao Ying, and Jure Leskovec. “Inductive representation learning on large graphs”. In: *Advances in neural information processing systems* 30 (2017).

[Abu+19] Sami Abu-El-Haija et al. “Mixhop: Higher-order graph convolutional architectures via sparsified neighborhood mixing”. In: *international conference on machine learning*. PMLR. 2019, pp. 21–29.

[FL19] Matthias Fey and Jan Eric Lenssen. “Fast graph representation learning with PyTorch Geometric”. In: *arXiv preprint arXiv:1903.02428* (2019).

[Ott19] Martin Otto. “Graded modal logic and counting bisimulation”. In: *arXiv preprint arXiv:1910.00039* (2019).

[Pas+19] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[Ham20] William L Hamilton. “Graph representation learning”. In: *Synthesis Lectures on Artifical Intelligence and Machine Learning* 14.3 (2020), pp. 1–159.

[Hu+20] Weihua Hu et al. “Open graph benchmark: Datasets for machine learning on graphs”. In: *Advances in neural information processing systems* 33 (2020), pp. 22118–22133.

[RS20] Benedek Rozemberczki and Rik Sarkar. “Characteristic functions on graphs: Birds of a feather, from statistical descriptors to parametric models”. In: *Proceedings of the 29th ACM international conference on information & knowledge management*. 2020, pp. 1325–1334.

[Sat20] Ryoma Sato. “A survey on the expressive power of graph neural networks”. In: *arXiv preprint arXiv:2003.04078* (2020).

[Chi+21] Eli Chien et al. “Node Feature Extraction by Self-Supervised Multi-scale Neighborhood Prediction”. In: *arXiv preprint arXiv:2111.00064* (2021).