FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Andrei Lupasco

Deep Neural Networks for Graph Data Processing

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis:	doc. RNDr. Iveta Mrázová, CSc.
Study programme:	Computer Science with specialization in Artificial Intelligence

Prague 2024

To my supervisor, doc. RNDr. Iveta Mrázová, CSc., for her guidance and support throughout the work on the thesis.
To Fabiana, my dearest friend, for her encouragement and help with the graphics in the thesis.

Title: Deep Neural Networks for Graph Data Processing

Author: Andrei Lupasco

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: doc. RNDr. Iveta Mrázová, CSc., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Graph Neural Networks (GNNs) are a powerful tool for handling machine-learning tasks on graph-structured data. Because of the distinct nature of graph data, traditional neural networks are not directly applicable to it. Most GNNs are based on the Weisfeiler-Lehman (WL) algorithm for graph isomorphism testing. However, just like the WL algorithm, they are incomplete and incapable of distinguishing certain graph structures. In this thesis, we provide an overview of the current state of the art in GNNs, focusing on their application to molecular data. We then present a novel approach to GNNs based on an algorithm for planar graph isomorphism testing, which produces a unique, learnable graph representation. Any sequential model can then use this representation, thus bringing together the fields of GNNs and modern deep-learning techniques. We evaluate the performance of our model on a dataset of molecules and compare it against existing models.

Keywords: Graph Neural Networks, Deep Neural Networks, Graph Isomorphism, Planar Graphs, Molecular Data

Contents

Introduction

Recent research has proven Graph Neural Networks (GNNs) to be highly effective for processing graph-structured data, showcasing their ability to capture the intricate relationships and interactions within these data structures. As graphs arise naturally in a wide range of domains, from social networks to molecules, developing efficient algorithms and models for processing them is of great importance. In particular, the field of chemoinformatics and bioinformatics, which often treat molecules as graphs, can significantly benefit from the advancements in graph neural networks, as advances in this field could lead to breakthroughs in drug discovery, drug analysis, etc., which in turn could have a significant beneficial impact on the healthcare industry and, ultimately, on human health.

In this thesis, we explore the current state of the art in graph neural networks, with a focus on their application on molecular data. They have been already successfully applied to solve real-world problems [1], [2], [3]. We will explore the various types of graph neural networks, examining their benefits and addressing their limitations. Moreover, we will present a novel approach to graph neural networks, specialized in planar graphs, aiming to address some of the limitations of existing models, as well as constructing a bridge between the fields of graph neural networks and modern deep learning techniques. It achieves this by producing a graph representation that can be fed into a traditional deep learning model, thus benefiting from the advancements in the field of deep learning while still being able to process graph data.

The thesis is structured as follows: in Chapter 1, we provide a quick introduction to graph theory and planar graphs. Chapter 2 is dedicated to the theoretical background of deep learning, as well as the basics of graph neural networks. In Chapter 3, we present an algorithm for testing isomorphism of planar graphs, which is a crucial part of our research. We then proceed to explain our modification of this algorithm, which is used to generate a learnable graph representation, as well as the architecture of the model that uses this representation. Chapter 4 provides an overview of the software accompanying this thesis. Finally, in Chapter 5, we compare the performance of our model against existing models on a dataset of molecules and discuss the results.

1 Graphs

1.1 Graph Theory 101

Graphs represent a powerful mathematical tool for modeling relational data. They can naturally describe complex relationships between entities in a wide range of domains, from social networks to molecules. We define a graph as a tuple $G = (V, E)$, where:

- V is a set of vertices, also called nodes.
- $E \subseteq V \times V$ is a set of edges, i.e., connections between vertices.

In this thesis, we will consider only *undirected* graphs, which have the property that $(v_i, v_j) \in E \Leftrightarrow (v_j, v_i) \in E$. We say that 2 vertices $v_i, v_j \in V$ are *adjacent* if there is an edge between them, i.e., $(v_i, v_j) \in E$. The *neighborhood* of a vertex is a set $N(v_i) = \{v_j \in V | (v_i, v_j) \in E\}$. The *degree* of a vertex is the number of its neighbors, i.e., $d(v_i) = |N(v_i)|$. We call a *path* a sequence of vertices $v_1, v_2, \ldots, v_k$ such that v_i is adjacent to v_{i+1} for all $i \in \{1, 2, \ldots, k-1\}$. 2 vertices are *connected* if there exists a path between them. A graph is *connected* if all pairs of vertices are connected. A graph is *biconnected* if it cannot be disconnected by removing a single vertex. Respectively, a graph is *triconnected* if it cannot be disconnected by removing any 2 vertices.

A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$, $E' \subseteq E$. A *connected component* of a graph is a maximal connected subgraph, i.e., a subgraph that is connected and cannot be extended by adding more vertices from the original graph. A *biconnected component*, and respectively a *triconnected component*, is a maximal biconnected, and respectively triconnected, subgraph. An articulation point of a graph is a vertex whose removal disconnects the graph.

A *subdivision* of a graph is a graph resulting from the subdivision of its edges. The *subdivision of an edge* $(v, u) \in E$ is the operation of adding a new vertex w and replacing the edge (v, u) with the edges (v, w) and (w, u).

We say that a graph is *planar* if it can be drawn in the plane without any edges crossing, and such a drawing is called a *planar embedding*.

An *isomorphism* between 2 graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is a bijective function $\phi : V_1 \mapsto V_2$ such that $(v_i, v_j) \in E_1 \Leftrightarrow (\phi(v_i), \phi(v_j)) \in E_2$.

The *graph isomorphism problem* in computer science is the problem of, given 2 graphs, determining whether they are isomorphic.

For general graphs, the graph isomorphism problem is not an easy one. Despite the amount of research on this topic, to this day, there is no known polynomial-time algorithm for solving the graph isomorphism problem, nor is it known to be NP-complete. Thus, the graph isomorphism problem is believed to be in the complexity class NP-intermediate. Another known problem in this class is the integer factorization problem. However, for certain classes of graphs, the graph isomorphism problem can be solved in polynomial time. One such instance is the case of planar graphs, for which the problem can be solved in logarithmic time [4].

Traditionally, there are 2 approaches to storing graphs into the memory of a computer: the *adjacency matrix* and the *adjacency list*. The adjacency matrix is a $|V| \times |V|$ matrix, where the entry A_{ij} is 1 if there is an edge between vertices v_i and v_j, and 0 otherwise. The adjacency list is a list of lists, where each list corresponds to a vertex, and contains the vertices adjacent to it.

1.2 Why Planar Graphs?

Bioinformatics and cheminformatics often treat molecules as graphs, where atoms are vertices and bonds are edges. Using this representation, the vast majority of molecules, albeit not all, can be represented as planar graphs [5]. This is due to the fact that, by Kuratowski's theorem [6], a graph is planar if and only if it does not contain a subgraph that is a subdivision of K_5 or $K_{3,3}$, where K_5 is the complete graph on 5 vertices and $K_{3,3}$ is the complete bipartite graph on 3 vertices in each partition. It is difficult to synthesize a molecule with such a property, and thus for most practical purposes, molecules can be considered planar.

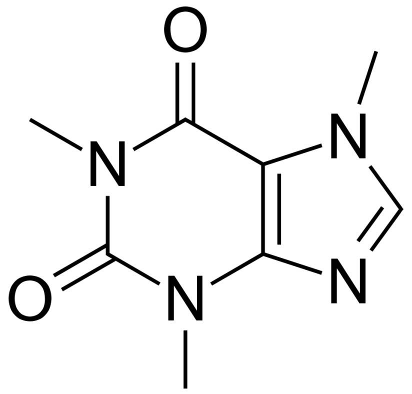
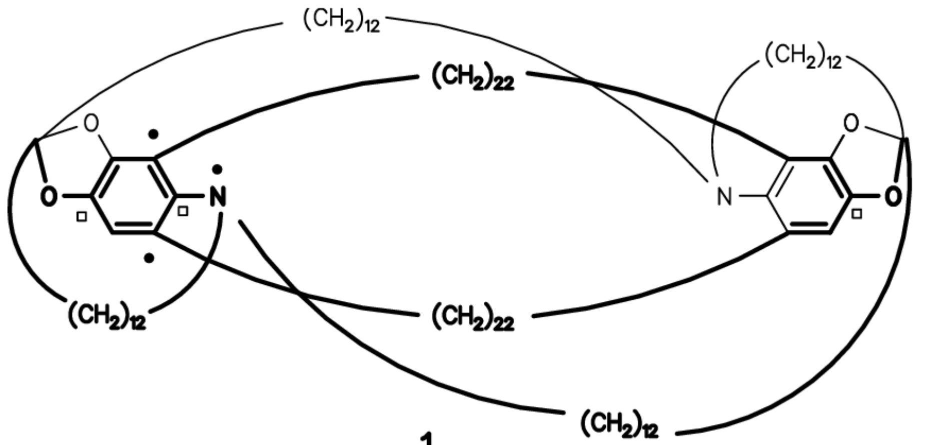
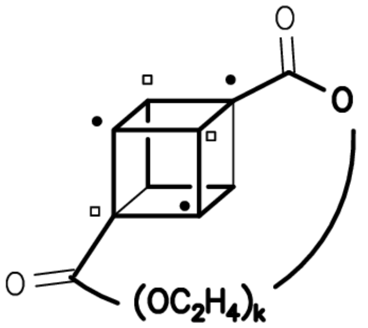


Figure 1.1 The structure of caffeine ($C_8H_{10}N_4O_2$), a molecule that can be represented as a planar graph.

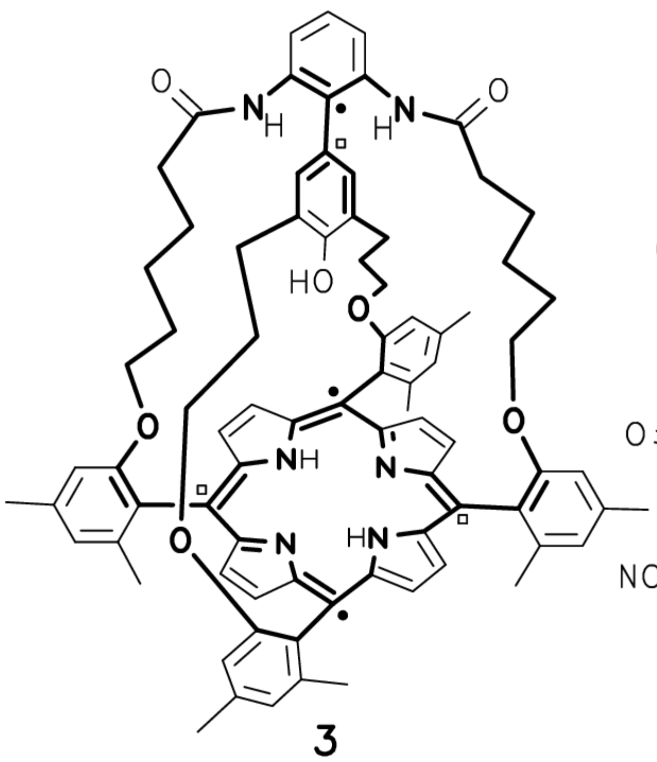
The planarity of molecules' graphs is a property that can be exploited in the development of algorithms, since the restriction to planar, as opposed to general graphs, often allows for more efficient algorithms. One such example is the above-mentioned polynomial-time algorithm for the graph isomorphism problem on planar graphs [4]. Thus, specializing in planar graphs could allow the usage of techniques unsuitable for general graphs, which could help improve the healthcare industry by simplifying the process of drug discovery, drug analysis, etc., a benefit that can not simply be overlooked.



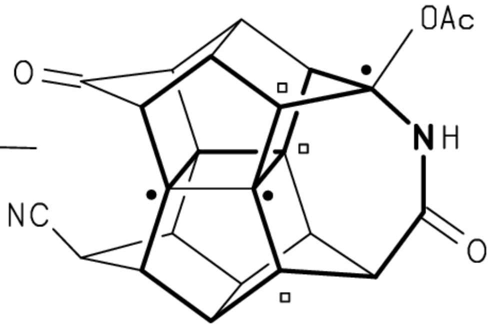
1



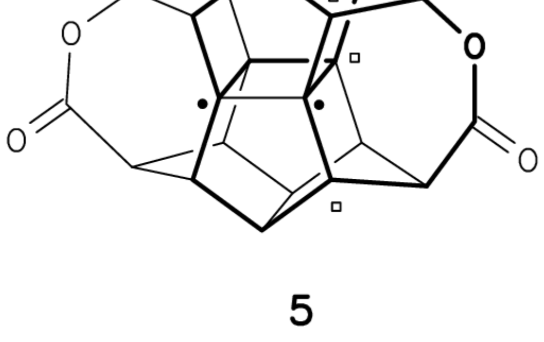
2



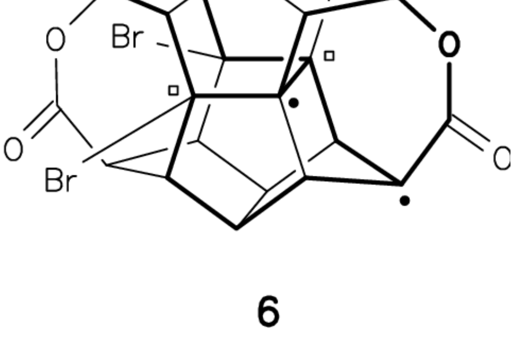
3



4



5



6

Figure 1.2 Examples of non-planar molecules. All contain subdivisions of $K_{3,3}$. Atoms and bonds included in the subdivisions are drawn in heavy lines. Image source: [5].

2 Deep Learning

2.1 Multilayer perceptron

The perceptron

Artificial neural networks are computational models inspired by the structure of the human brain, which is made out of interconnected neurons. A neuron is a cell consisting of several organelles. However we'll focus on only two of them: the dendrites and the axon. The dendrites recieve input signals from some neurons (in the form of electrical impulses), and the axon transmits the output signal to other neurons or to the muscles. See Figure 2.1 for a visual representation of a neuron.

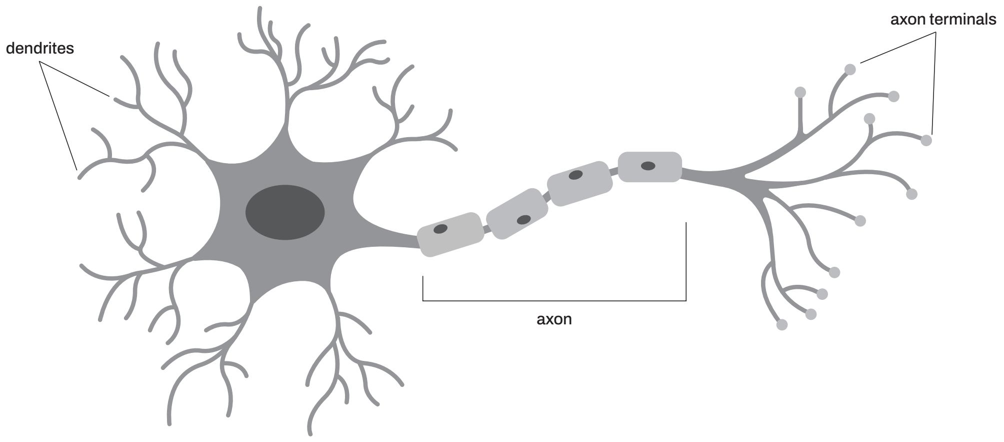


Figure 2.1 The structure of a neuron.

A historically important attempt at modeling the behavior of neurons was the *perceptron* model [7]. The perceptron is a simple model of a single neuron, which takes a vector of inputs $x \in \mathbb{R}^n$, and produces a single output $y \in \{0, 1\}$, i.e., a binary classification of the data. The output is computed by the following formula:

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

where w_i are the weights of the inputs, and b is the bias term.

Back to the analogy with the neuron, the input vector x corresponds to the strength of the electrical impulses received by the dendrites, while the weights w and the bias b correspond to the sensitivity of the said dendrites to the impulses.

The advantage of the model is that the weights (as well as the bias) are *learnable*, i.e., they can be adjusted during training in order to minimize the classification error.

However, the perceptron model has its limitations, as it can only classify linearly separable data. It was shown it cannot learn the XOR function [8], which is a simple example of a non-linearly separable function.

The multilayer perceptron

The limitations of the perceptron model have motivated the development of more complex models. The multilayer perceptron (MLP) [9] is a generalization of the perceptron model, which consists of multiple layers of perceptrons arranged in a feed-forward manner. The output of a perceptron in the i-th layer is used as input for the perceptrons in the $(i+1)$-th layer. The output of the last layer is the output of the network. See Figure 2.2 for an example. Moreover, to increase the expressive power of the model and to overcome the limitations of the perceptron, the output of each layer is passed through a non-linear *activation function*, such as the sigmoid, the hyperbolic tangent, or the rectified linear unit (ReLU).

The behavior of the i-th layer of an MLP can be described by the following formula:

$$h_i = \sigma(W_i h_{i-1} + b_i)$$

where h_i is the output of the i-th layer, W_i are the learnable weights of the layer (in the form of a matrix), b_i is the learnable bias vector, and σ is the activation function.

Contrary to the perceptron model, MLPs can be trained for a variety of tasks besides binary classification, such as regression, multi-class classification, etc. The training of an MLP can be performed in multiple ways, but by far, the most popular is the *backpropagation* algorithm, which is a form of the *gradient descent* algorithm [10]. It adjusts the weights and biases of the network in order to minimize a *loss function*, which quantifies the error of the network on the training data. The weights are adjusted by computing the gradient of the loss function with respect to the weights and updating the weights in the opposite direction of the gradient.

2.2 Convolutional neural networks

Convolutional Neural Networks (CNNs) [11], [12] have become a cornerstone of modern machine learning, particularly in tasks involving *structured data*, e.g., image, signal, and natural language processing. Structured data is data that has additional, hidden relationships between its elements. For example, in an image, neighboring pixels have highly correlated values. In a text, the meaning of a word depends more heavily on the words in the immediate vicinity rather than on the words that are far away. Convolutional neural networks are designed to exploit these relationships and do so efficiently. Despite the fact that CNNs were primarily developed for image processing, and they have been most successful in this domain, we will rather focus on convolutions applied to 1D data, such as time series, as they are more relevant to the topic of this thesis.

The efficiency of CNNs is best illustrated by the following example. Consider a sequence of 124 words, each represented by a vector of size 124. We want to map the whole sequence to a feature vector of size 124, which would represent the semantic meaning of the sequence. A valid option would be to flatten the

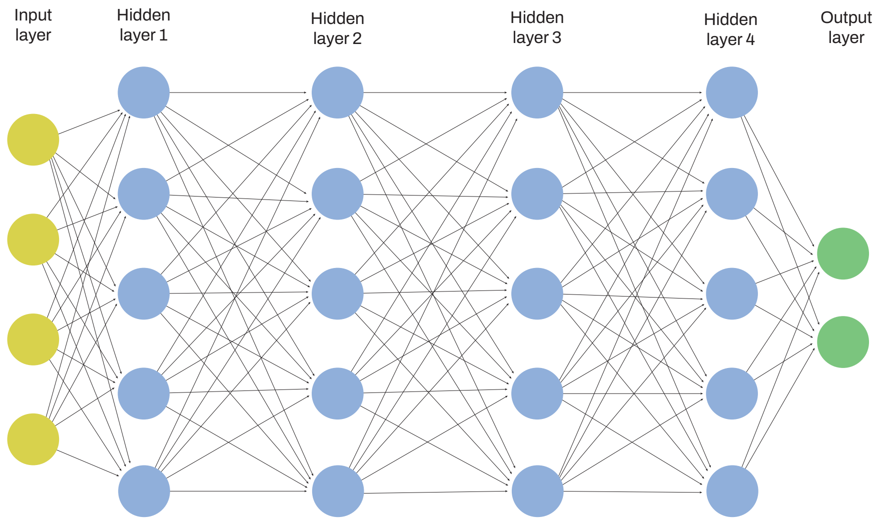


Figure 2.2 A multilayer perceptron consisting of 5 layers. We call the first 4 layers the *hidden layers* and the last layer the *output layer*.

sequence into a vector of size 124 × 124, and to pass it to a fully connected layer (MLP) with 124 outputs. However, this approach requires 124^3 parameters and quickly becomes infeasible for larger sequences.

We can improve the above method by exploiting the structure of the data. More specifically, connections between words that are far away from each other are less important than the connections between neighboring words, and thus, we may omit them. This is exactly what the convolution operation does. One way to think of this operation is that we're sliding a *filter* over the input sequence. For each position of the filter, we multiply the overlapping values of the sequence and the filter and sum up the results. Figure 2.3 provides a visual representation of the convolution operation.

Formally, given an input sequence of shape (L, C_{in}), where L is the length of the sequence and C_{in} is the number of channels (features) of the sequence, and a filter of shape (K, C_{in}), where K is the length of the filter, the output of the convolution operation is a sequence of shape $(L - K + 1, 1)$, and is computed as follows:

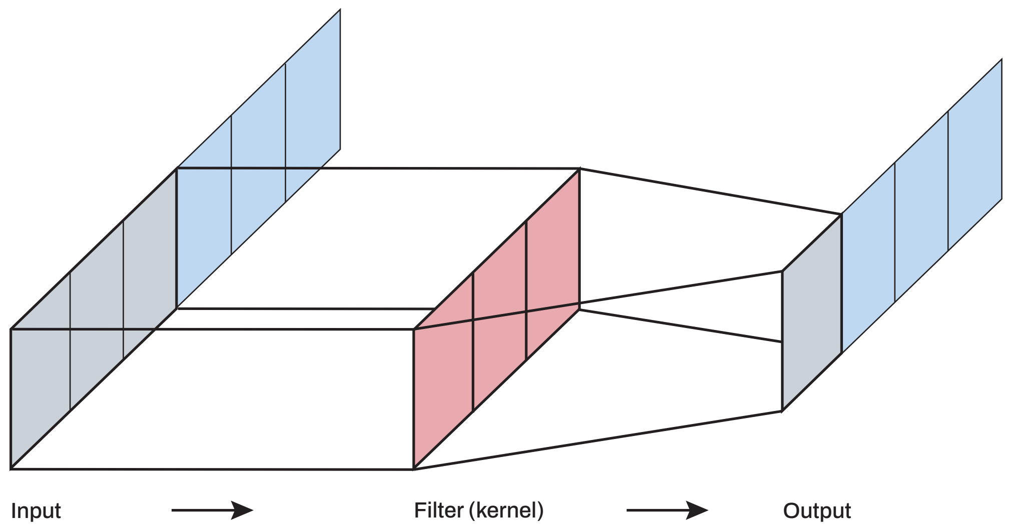
$$y_i = \sum_{j=0}^{K-1} x_{i+j}^T f_j = \sum_{j=0}^{K-1} \sum_{k=0}^{C_{in}-1} x_{i+j,k} f_{j,k}$$

where x is the input sequence, f is the filter, and y is the output sequence.

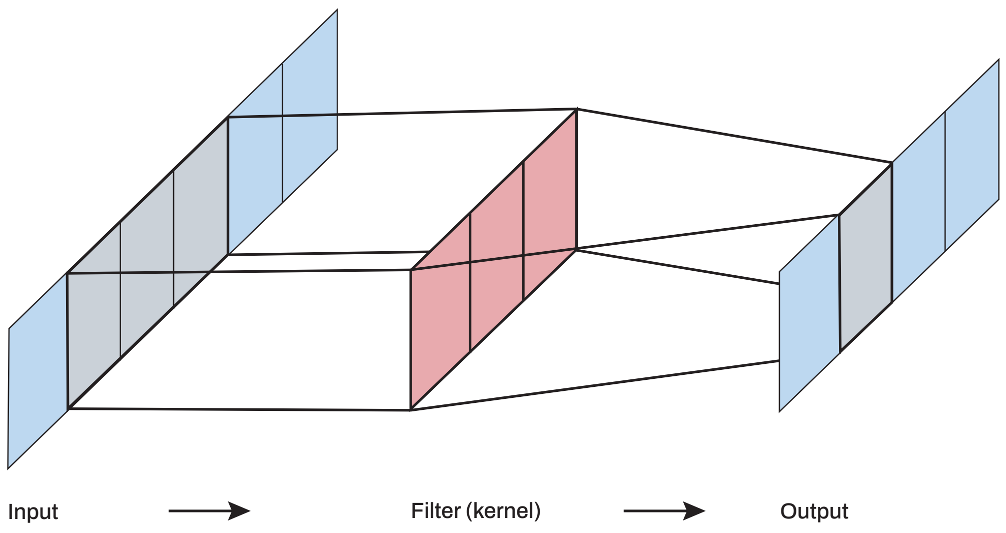
If we want multiple output channels, we can simply use multiple filters and stack their results. The output of the convolution operation is then a sequence of shape $(L - K + 1, C_{out})$, where C_{out} is the number of filter channels.

Sometimes, we want to keep the length of the output sequence the same as the input sequence. This can be achieved by using *padding*. Padding is the process of adding zeros to the beginning and the end of the input sequence, which allows us to get an output of shape (L, C_{out}).

For extremely long sequences, we can use a technique called *strided convolution*. Strided convolution is the process of skipping some of the positions of the filter,



(a)



(b)

Figure 2.3 A visualization of the sliding convolution operation.

which allows us to reduce the length of the output sequence. This can be useful when we want to reduce the computational cost of the convolution operation. It is important to note that as long as the size of the filter is larger than the stride, each position of the input sequence will be used in the computation of at least one output position.

Back to the example of the sequence of words, we can use a convolutional layer with a filter of size 3 (a matrix of shape $(3, 124)$ to account for the input channels). To achieve the desired number of channels in the output, we simply use 124 different filters. We also use padding to keep the length of the output sequence the same as the input sequence. This way, we only have $3 \times 124 \times 124 = 3 \times 124^2$ parameters, which is a significant improvement over the previous method. Moreover, contrary to the previous method, the number of trainable parameters does not depend on the length of the input sequence, but only on the size of the filter and the number of input/output channels.

2.3 Recurrent neural networks

Recurrent neural networks (RNNs) [13], [14], are a family of neural networks designed to work with *sequential data*, i.e., data that has a temporal structure. Examples of sequential data include time series, text, and audio. RNNs achieve this by introducing a *hidden state*, which is a lossy summary of the data seen so far and is computed at each time step alongside the output of the network. The hidden state is then used as input for the next time step (thus the name recurrent neural networks), which allows the network to remember the information from the previous time steps. Consult Figure 2.4 for a visual representation of the RNN neuron.

Formally, given a sequence of inputs $x^{(1)}, x^{(2)}, ..., x^{(T)}$, and an initial hidden state $h^{(0)}$, the outputs of the RNN, and the next hidden states are computed as follows:

$$(y^{(t)}, h^{(t+1)}) = f(x^{(t)}, h^{(t)})$$

where f simulates the RNN cell, which is a learnable differentiable function that takes the input $x^{(t)}$ and the current hidden state $h^{(t)}$, and produces the output $y^{(t)}$ and the next hidden state $h^{(t+1)}$. The output $y^{(t)}$ is then used for the task at hand, e.g., classification, regression, etc.

This model provides several advantages over traditional MLPs for sequential data:

1. The same transition function f with the same parameters can be used at each time step.
2. The size of the model is independent of the length of the sequence. This is because the same weights are used at each time step, and the hidden state is used to remember the information from the previous time steps.

RNNs are also trained using the backpropagation algorithm. To be able to use it, we need to *unfold* the RNN, i.e., unroll the network for a fixed number of time steps, and treat it as a feed-forward neural network (see Figure 2.5).

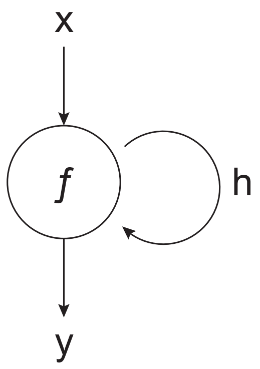


Figure 2.4 The structure of a single RNN neuron.

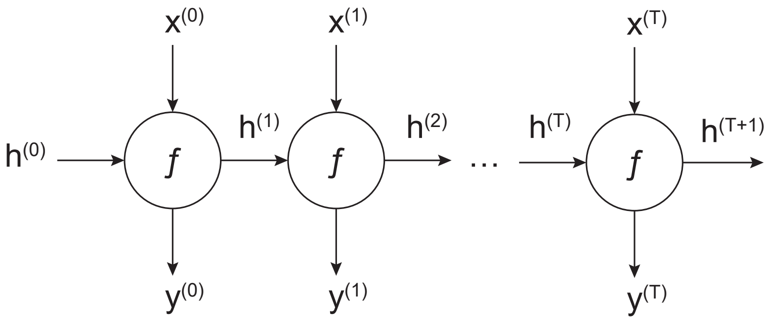


Figure 2.5 The unfolded RNN for T time steps. The same weights and function f are used at each time step.

However, when the input sequence gets too long, simple RNNs suffer from the vanishing gradient problem, or its counterpart, the exploding gradient problem. In other words, the gradients of the loss function with respect to the weights of the network become too small or too large, which makes the training of the network unstable. To this day, the most efficient RNN architecture addressing this issue is the *gated RNN*, which we will discuss further.

LSTM

The Long Short-Term Memory (LSTM) [15] architecture elegantly solves the problem of vanishing/exploding gradients by introducing a more complex cell structure (gates), and a second hidden state, called the *cell state*. The gates correspond to matrix multiplications with the learnable weights, followed by a sigmoid or a hyperbolic tangent activation function. The output of the gates is then used to control the flow of information in the cell.

The behavior of the LSTM cell is described by the following formulae:

$$\begin{aligned}
f_t &= \sigma(W_f \cdot [h_t, x_t] + b_f)\\
i_t &= \sigma(W_i \cdot [h_t, x_t] + b_i)\\
o_t &= \sigma(W_o \cdot [h_t, x_t] + b_o)\\
\tilde{C}_t &= \tanh(W_C \cdot [h_t, x_t] + b_C)\\
C_{t+1} &= f_t * C_t + i_t * \tilde{C}_t\\
h_{t+1} &= o_t * \tanh(C_t)
\end{aligned}$$

where f_t, i_t, o_t are the forget, input, and output gates, respectively, $\tilde{C}_t$ is the candidate cell state, C_t is the cell state, and h_t is the hidden state, as well as the output of the cell. W_f, W_i, W_o, W_C are the learnable weights of the gates, and b_f, b_i, b_o, b_C are the learnable biases.

The forget gate f_t controls how much of the previous hidden state h_t is kept in the new cell state C_{t+1}. The input gate i_t controls how much of the previous hidden state h_t is added to the new cell state. The output gate o_t controls how much of the previous hidden state is used to compute the new one, h_{t+1}. Figure 2.6 depicts in detail the structure of the LSTM cell.

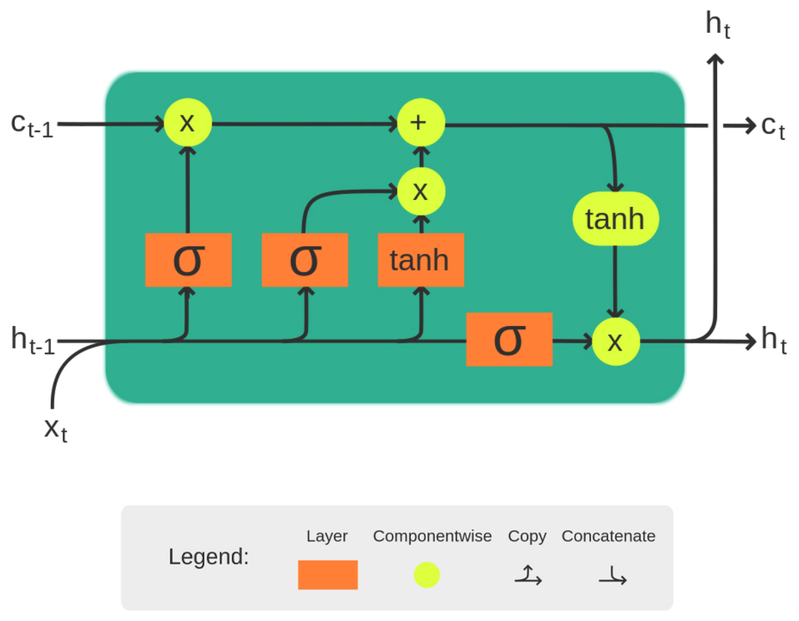


Figure 2.6 The structure of the LSTM cell.

GRU

The Gated Recurrent Unit (GRU) [16] is a simpler version of the LSTM cell, which has been shown to be as expressive as the LSTM, while having fewer parameters. The GRU cell has only 2 gates, the reset gate r_t and the update gate z_t, and a single hidden state h_t.

The behavior of the GRU cell is described by the following formulae:

$$\begin{aligned}
z_t &= \sigma(W_z \cdot [h_t, x_t] + b_z) \\
r_t &= \sigma(W_r \cdot [h_t, x_t] + b_r) \\
\tilde{h}_t &= \tanh(W_h \cdot [r_t * h_t, x_t] + b_h) \\
h_{t+1} &= (1 - z_t) * h_t + z_t * \tilde{h}_t
\end{aligned}$$

where z_t is the update gate, r_t is the reset gate, $\tilde{h}_t$ is the candidate hidden state, and h_t is the hidden state, as well as the output of the cell. W_z, W_r, W_h are the learnable weights of the gates, and b_z, b_r, b_h are the learnable biases. Figure 2.7 depicts in detail the structure of the GRU cell.

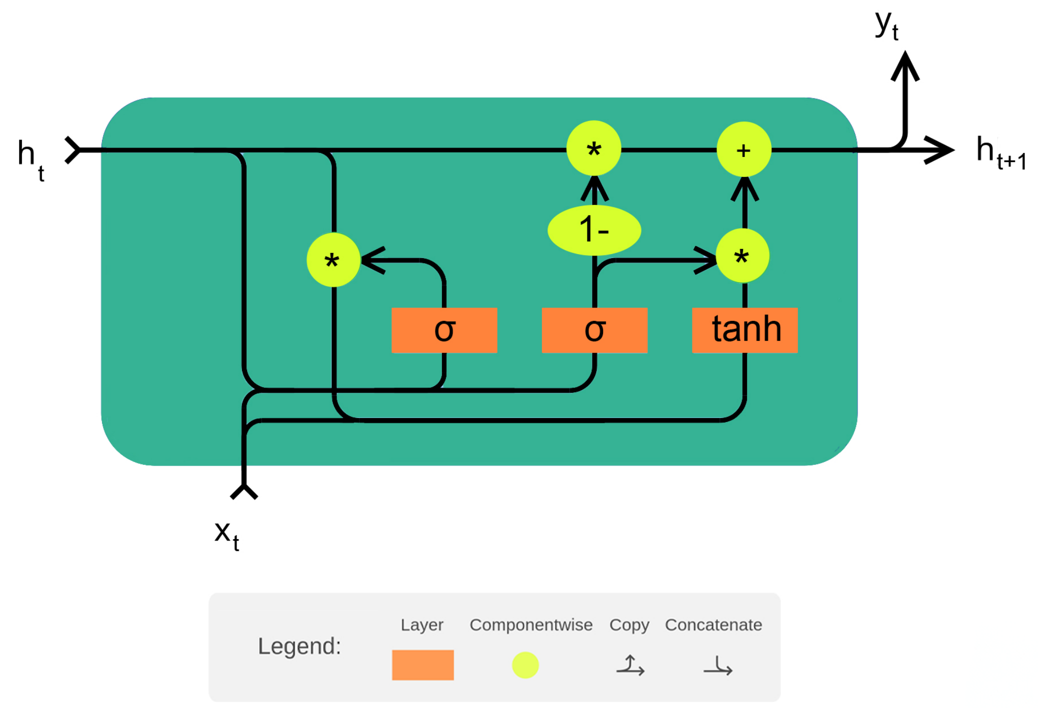


Figure 2.7 The structure of the GRU cell.

2.4 A Primer on Graph Neural Networks

2.4.1 What are graph neural networks?

Graph Neural Networks (GNNs) are a family of neural networks designed to work with graphs. Unless we're talking about generative GNNs, the input of a GNN is a graph. In the deep learning setting, a graph is usually represented by its adjacency matrix. Moreover, in order to correspond with real-world data, the graph may have additional information. For example, each vertex of the graph may have a feature vector, which represents its properties. Analogously, the same can be true about edges, but we will restrict our discussion to the case where the edges have no features. A GNN aims to learn a function on the graph, i.e., its structure and its node features, and to produce an output, which is usually a classification, regression, or clustering of the graph.

Formally, for a graph $G = (V, E)$, with the adjacency matrix A, and the node feature matrix X (each row of X is a feature vector of a vertex), the goal of a GNN is to learn a function f such that $y = f(A, X)$, where y is the output of the network.

2.4.2 Why are graphs different?

In computer science, graphs are usually represented via their adjacency matrices. One way to make it an input for a neural network is to flatten the adjacency matrix into a vector. Formally, given a graph $G = (V, E)$, and its adjacency matrix A, we could use as input the vector $x = A[1] \oplus A[2] \oplus ... \oplus A[|V|]$, where $A[i]$ is the i-th row of the adjacency matrix, and $\oplus$ is the concatenation operator. The issue with this approach is that x depends on the ordering of the vertices that was chosen when constructing A, which was completely arbitrary and bears no information about the structure of the graph itself. In other words, given a permutation matrix P, a second adjacency matrix $A' = PAP^T$ would produce a completely different vector x', even though they represent the same graph. The deep learning models that we have described so far are not invariant to the ordering of the input, and thus they would not be able to learn from such a representation.

Formally, a function f that takes an adjacency matrix as input, and produces a vector $z = f(A)$, has to obey one of the following properties:

- **Permutation invariance:** $f(PAP^T) = f(A)$ for any permutation matrix P.
- **Permutation equivariance:** $f(PAP^T) = Pf(A)$ for any permutation matrix P.

Intuitively, permutation invariance means that the function f is indifferent to the ordering of the vertices, while permutation equivariance means that the function f cares about the ordering, but the output is permuted in a consistent way with the input.

The same is true for graph neural networks. Since their goal is to approximate a given function on a graph (or its adjacency matrix), they have to be invariant or equivariant to the ordering of the vertices.

2.4.3 Scarselli's graph neural network

The first graph neural network was proposed by Scarselli et al. [17]. Its innovative idea was to introduce a *hidden state* for each vertex of the graph and update the hidden state of each vertex by using the hidden states of its neighbors. Initially, the hidden state of each vertex is its feature vector. The hidden state is then used to compute the output of the network. The model is then a composition of two functions: the *transition function* and the *output function*. Let $G = (V, E)$ be a graph with the adjacency matrix A and the node feature matrix X. Let h_v and x_v be the hidden state and the feature vector of the vertex v, respectively. The model can then be formalized as follows:

1. **Transition function f:** $h_v^{(t)} = f(h_v^{(t-1)}, \{h_u^{(t-1)} | u \in N(v)\}, x_v, \{x_u | u \in N(v)\})$

2. **Output function g:** $y_v^{(t)} = g(h_v^{(t)}, x_v)$

Intuitively, the transition function f computes the new hidden state of the vertex v by aggregating its neighbors' hidden states and features. The output function g then computes the output corresponding to the vertex v based on its hidden state and its feature vector.

The transition function f is of special interest, as it has inspired the development of many other GNNs. It is defined as:

$$h_v^{(t)} = \sum_{u \in N(v)} \phi(h_u^{(t-1)}, x_u),$$

where ϕ is a learnable parametric function.

The permutation invariance of the model is obtained by the fact that the transition function f is a sum of the hidden states of the neighbors, and summation is permutation invariant. Many other GNNs are based on this idea, and they differ in the choice of the function ϕ.

Moreover, to assure that h_v is uniquely defined, the transition function f has to be a *contractive mapping*, i.e., $||f(h_v^{(t-1)}, \{h_u^{(t-1)} | u \in N(v)\}, x_v, \{x_u | u \in N(v)\}) - f(h_{v'}^{(t-1)}, \{h_u^{(t-1)} | u \in N(v')\}, x_{v'}, \{x_u | u \in N(v')\})|| \leq ||h_v^{(t-1)} - h_{v'}^{(t-1)}|| \forall v, v' \in V$, where $|| \cdot ||$ is the Euclidean norm. By Banach's fixed-point theorem, a contractive mapping has a unique fixed point, which is going to be the unique hidden state of the vertex v. To obtain the said fixed point, the transition function f is applied iteratively on the hidden states of the vertices, until convergence. This makes the model recurrent in nature. However, this idea, unlike the previous ones, was not adopted by the later GNNs.

2.4.4 Message-passing neural networks

The model proposed by Scarselli et al. [17] was the first of its kind, and it has inspired many other models. Despite their differences, most of them share the ideas present in Scarselli's model, and they can be generalized via the paradigm of *message-passing neural networks* [18], [19].

The idea behind message-passing neural networks is once again to introduce a hidden state for each vertex of the graph. The hidden state will be iteratively updated during each learning step (epoch). The message-passing update rule can be formalized in the following way:

$$\begin{aligned} h_v^{(t+1)} &= \text{UPDATE}^{(t)}(h_v^{(t)}, \text{AGGREGATE}^{(t)}(\{h_u^{(t)} | u \in N(v)\})) \\ & \text{UPDATE}^{(t)}(h_v^{(t)}, m_{N(v)}^{(t)}), \end{aligned}$$

where $h_v^{(t)}$ is the hidden state of the vertex v at the t-th iteration, UPDATE is a differentiable function, and AGGREGATE is a differentiable, permutation invariant function. $m_{N(v)}^{(t)}$ represents the message being passed from the neighbors of the vertex v to it at the t-th iteration. Both UPDATE and AGGREGATE are learnable functions that are continuously updated during the training of the network, and the superscript (t) denotes their version at the t-th iteration.

During the k-th iteration of the training algorithm, for a vertex v, the AGGREGATE function computes, based on the set of hidden states of the neighbors of v, a message $m_{N(v)}^{(k)}$. The message is supposed to summarize the information present in v's neighborhood. The UPDATE function then computes the new hidden state of v based on the message $m_{N(v)}^{(k)}$ and the current hidden state $h_v^{(k)}$. This way, the state of v is updated to contain the information its neighbors had at $k-1$-th iteration. Moreover, its neighbors' hidden states at $k-1$-th iteration are containing the information from their respective neighbors at $k-2$-th iteration, and so on. In such a manner, the information is propagated through the graph, and if done long enough, the hidden states of the vertices are updated to contain the information from the whole graph. Note that we have defined the AGGREGATE function to take a set as input, which means that it is permutation invariant. Figure 2.8 provides an overview of the message-passing scheme.

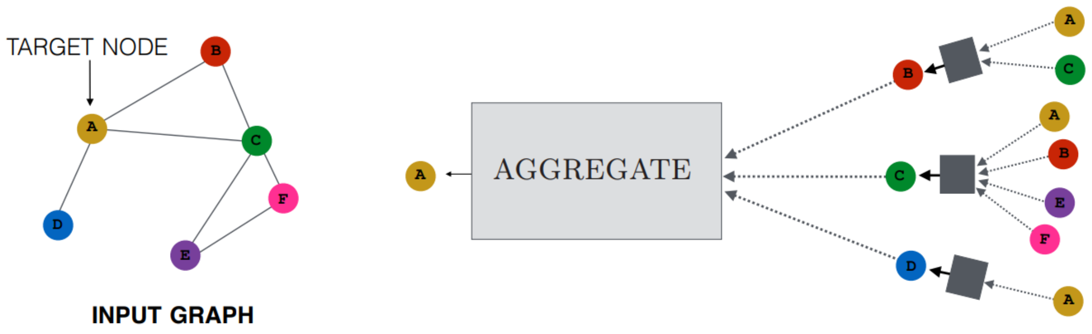


Figure 2.8 Overview of how a vertex aggregates information from its neighbors during multiple iterations. Source: [19].

The basic message-passing model

Many implementations of the AGGREGATE and UPDATE functions are possible. One of the simplest models is the following:

$$
\begin{aligned}
m_{N(v)}^{(t)} &= \sum_{u \in N(v)} h_u^{(t)} \\
h_v^{(t+1)} &= \sigma(\phi(h_v^{(t)}) + \psi(m_{N(v)}^{(t)})),
\end{aligned}
$$

where σ is a non-linear activation function, and ϕ and ψ are learnable functions (usually MLPs). The AGGREGATE function is a simple sum of the hidden states of the neighbors. Since summation is commutative, the function is permutation invariant.

Graph Convolutional Neural Networks

One of the most successful models obeying the message-passing paradigm is the Graph Convolutional Neural Network (GCN) [20]. It adds two ideas to the message-passing model:

1. **Self-loops:** An edge is added from each vertex to itself, thus extending the definition of a neighborhood of a vertex to include it as well. This allows the vertex to propagate its own information in the AGGREGATE phase.

2. **Normalization:** The aggregation operation is normalized by the degree of the vertex. The intuition behind this is to prevent vertices with many neighbors from having hidden states with too large magnitudes and thus dominating the learning process.

The GCN model can be formalized as follows:

$$\begin{aligned} m_{N(v)}^{(t)} &= \sum_{u \in N(v) \cup \{v\}} \frac{1}{\sqrt{|N(v)||N(u)|}} h_u^{(t)} \\ h_v^{(t+1)} &= \psi(m_{N(v)}^{(t)}), \end{aligned}$$

where $|N(v)|$ is the degree of the vertex v, and ψ is a learnable function. Notice that the UPDATE function is now a simple MLP, which takes the aggregated message as input. Because of the self-loops, there is no need for the UPDATE function to take the hidden state of the vertex as input, as it is already included in the message. Moreover, the AGGREGATE function is normalized by the square root of the product of the degrees of the vertices.

Graph Attention Networks

Another popular model based on the message-passing paradigm is the Graph Attention Network (GAT) [21]. It was inspired by the success of the attention mechanism in the field of natural language processing [22], [23], and it expands the idea of *attention* to graph neural networks. The model is trying to assign an attention coefficient, i.e., a weight, for each neighbor of a vertex, which denotes its importance. These coefficients are then used during the aggregation phase as follows:

$$\begin{aligned} m_{N(v)}^{(t)} &= \sum_{u \in N(v)} \alpha_{v,u} h_u^{(t)} \\ h_v^{(t+1)} &= \psi(m_{N(v)}^{(t)}), \end{aligned}$$

where ψ is a learnable function, and $\alpha_{v,u}$ is the attention coefficient between the vertices v and u.

There are many approaches to computing the attention coefficients. One of the most popular is the following:

$$\alpha_{v,u} = \frac{\exp(\phi(h_v, h_u))}{\sum_{t \in N(v)} \exp(\phi(h_v, h_t))},$$

where ϕ is a learnable function, usually an MLP, and exp is the exponential function. It is important to note that the attention coefficients are normalized by the sum of the exponential of the similarities between the vertex v and its neighbors. This way, the attention coefficients are in the range $[0, 1]$, and they sum up to 1.

In addition to this, as inspired by the transformers architecture [23], the GAT model can be extended to use multiple attention heads. In brief, each attention head has its individual learnable parameters, and thus its individual attention coefficients. On the other hand, they all compute a hidden state of lower dimensionality. The hidden states of the attention heads are then concatenated and passed through an MLP to produce the final hidden state of the vertex.

Graph Pooling

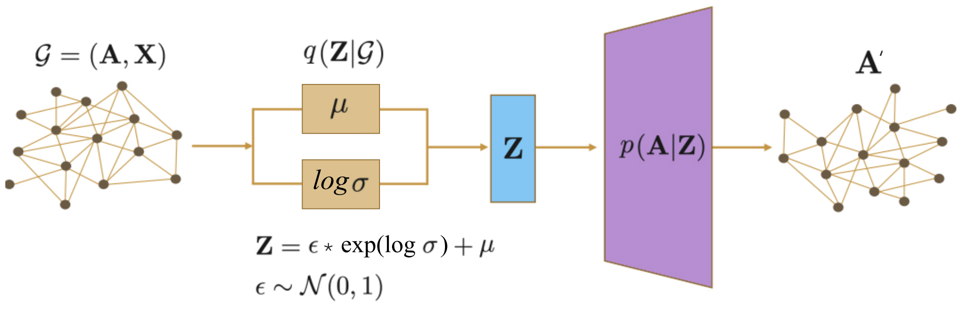
The message-passing paradigm we have described, together with all its variations, are computing a hidden state for each vertex of the graph. However, in many cases, we are interested in the whole graph, rather than in its individual vertices, e.g., to perform a graph classification task. Thus, the following question arises naturally: "How can we summarize the information from the hidden states of the vertices into a single vector, which represents the whole graph?".

The techniques that address this question are called *graph pooling* methods. The main principle they must obey is that they have to work in a permutation invariant fashion, just like the message-passing models. As seen previously, a simple yet efficient way to summarize the information from the hidden states of the vertices would be to simply sum them up, or perform a normalized sum, e.g., the average. We will stick to the summation pooling technique.

2.4.5 Graph Autoencoders

One more task we're interested in when it comes to graphs is the task of *graph reconstruction*, which itself is closely related to the task of *graph generation*.

Generative graph neural networks were also inspired by successes in the field of deep generative models. One such prominent example are the variational autoencoder models (VAEs) [24], which have inspired the development of the Graph Autoencoders (GAEs) and Variational Graph Autoencoders (VGAEs) [25], [19]. The goal of the VGAE model is to train a *probabilistic decoder* model $p(A|Z)$, which would be able to construct the adjacency matrix A (and thus a graph), given some information Z about the graph, which we call a *latent variable*. Figure 2.9 provides a visual representation of the VGAE model.



$\mathcal{G} = (\mathbf{A}, \mathbf{X})$
$q(\mathbf{Z}|\mathcal{G})$
μ
$\log\sigma$
$\mathbf{Z}$
$p(\mathbf{A}|\mathbf{Z})$
$\mathbf{A}'$
$\mathbf{Z} = \epsilon * \exp(\log\sigma) + \mu$
$\epsilon \sim \mathcal{N}(0,1)$

Figure 2.9 The structure of the Variational Graph Autoencoder model.

However, to be able to train the decoder model, we need to have a *probabilistic encoder* model $q(Z|G)$, which would be able to infer the latent variable Z from the graph G. The encoder model is usually a graph neural network, which computes the hidden state of the graph, and then maps it to the latent variable Z. When the training is done, the decoder model $p(A|Z)$ can then act independently, and generate new graphs by sampling $Z \sim p_0(Z)$ from some prior distribution $p_0(Z)$. More formally, the VGAE model consists of the following components:

- A **probabilistic decoder** model $p(A|Z)$, which takes as input a latent variable Z, and outputs the adjacency matrix A.
- A **probabilistic encoder** model $q(Z|G)$, which takes as input a graph G, and outputs the latent variable Z.
- A **prior distribution** $p_0(Z)$ over the latent space of the variable Z. A common choice is the standard normal distribution, i.e., $Z \sim \mathcal{N}(0, I)$.

Then, given a dataset of graphs $D = \{G_1, G_2, ..., G_N\}$, the goal of the VGAE model is to minimize the following loss function, called the *evidence likelihood lower bound* (ELBO):

$$\mathcal{L} = \sum_{G_i \in D} \mathbb{E}_{q(Z|G_i)}[p(G_i|Z)] - \text{KL}(q(Z|G_i)||p_0(Z)),$$

where KL is the Kullback-Leibler divergence between the encoder distribution $q(Z|G)$ and the prior distribution $p_0(Z)$. The intuition behind the ELBO loss is that it is trying to maximize the similarity between the decoder and the encoder, while minimizing the divergence between the encoder and the prior distribution.

The encoder model

The encoder uses two separate GNNs (as discussed in the previous section), usually Graph Convolution Networks. They aim to learn the parameters of the distribution $q(Z|G)$, which is a Gaussian distribution with the mean μ and the variance σ^2. The encoder model is then defined as follows:

$$\begin{aligned} \mu &= \text{GNN}_\mu(A, X) \\ \log \sigma &= \text{GNN}_\sigma(A, X), \end{aligned}$$

where GNN_μ and GNN_σ are arbitrary graph neural networks, and A and X are the adjacency matrix and the node feature matrix of the graph $G = (V, E)$ taken as input, respectively. In this case, $\mu \in \mathbb{R}^{|V| \times d}$ and $\sigma \in \mathbb{R}^{|V| \times d}$, are matrices, where d is the dimension of the latent variable Z. They are matrices instead of vectors to allow for the latent variable to be different for each vertex of the graph.

Next, the latent variable Z is computed as:

$$Z = \epsilon * \exp(\log \sigma) + \mu,$$

where ϵ is a random sample from the standard normal distribution, i.e., $\epsilon \sim \mathcal{N}(0, I)$, and $*$ denotes the element-wise multiplication.

The decoder model

Given the latent variable $Z \in \mathbb{R}^{|V| \times d}$, the decoder model is learning the posterior distribution of adjacency matrices conditioned on the latent variable. In other words, its goal is to predict the likelihood of the presence of an edge between each pair of vertices, given the latent variable Z. In the original paper [25], the authors have used a fairly simple technique for the decoder. The likelihood of the presence of an edge between the vertices v and u is computed as:

$$p(A'[v,u] = 1 | z_v, z_u) = \sigma(z_v^T z_u),$$

where σ is the sigmoid function, and z_v and z_u are the latent variables of the vertices v and u, respectively.

Next, we assume that the presence of an edge between each pair of vertices is independent, and we compute the likelihood of the adjacency matrix A as:

$$p(A'|Z) = \prod_{v,u \in V} p(A'[v,u] = 1 | z_v, z_u).$$

Non-probabilistic graph autoencoders

A simplification of the VGAE model is the Graph Autoencoder (GAE) model. The latent variable Z, and the reconstructed adjacency matrix A' are computed as follows:

$$\begin{aligned} Z &= \mathrm{GNN}(A, X) \\ A' &= \sigma(ZZ^T), \end{aligned}$$

where GNN is a graph neural network, and σ is the sigmoid function.

2.5 GNNs and Graph Isomorphism

Graph Isomorphism

Recall that an *isomorphsim* between two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is a bijective function $\phi : V_1 \mapsto V_2$, s.t. $(v_i, v_j) \in E_1 \Leftrightarrow (\phi(v_i), \phi(v_j)) \in E_2$. The problem of testing for graph isomorphism in computer science is the problem of determining whether such a bijection exists between 2 given graphs. Intuitively, for two graphs to be isomorphic, they must be essentially identical. Isomorphic graphs represent the same underlying structure, the same relationships between vertices, but they may have a different ordering of the vertices. Formally, we may say that two graphs G_1, G_2, with their adjacency matrices A_1 and A_2, as well as node feature matrices X_1 and X_2, are isomorphic if there exists a permutation matrix P, such that $A_1 = PA_2P^T$ and $X_1 = PX_2$.

It is important to note that the ordering of the vertices we choose when representing a graph via its adjacency matrix (or any other permutation-sensitive representation) is completely arbitrary, and it bears no information about the structure of the graph itself. Despite the simple definition of the graph isomorphism problem, it is computationally difficult to solve. One naive approach would be to check all possible permutation matrices P, and stop when we find one that satisfies the above conditions. However, the time complexity of such an approach would take $O(|V|!)$ time, which makes the algorithm infeasible for graphs on more than a few dozen vertices.

Moreover, no polynomial-time algorithm is known for the graph isomorphism problem, nor is it known for it to be NP-complete, and therefore it is believed to be in the complexity class NP-intermediate (NPI) [26]. For example, the integer factorization problem is another problem suspected to be in NPI. However, there are practical algorithms that can solve the graph isomorphism problem for many classes of graphs. Most of them do so by computing a canonical code for a graph, such that two graphs are isomorphic if and only if they have the same canonical code. One of the most well-known algorithms for this is the Weisfeiler-Leman (WL) test [27].

Graph Isomorphism and the power of GNNs

The idea of graph isomorphism is helpful for quantifying the power of graph neural networks. Most GNNs, at some point during the execution (usually in the late stages), compute a representation of the input graph $z_G \in \mathbb{R}^d$, where d is the dimension of the representation. This representation is then used for the task at hand, such as node classification, graph classification, etc. We may then pose the question: "Can this representation z_G be used in testing for graph isomorphism?". More specifically, a "perfect" GNN would be able to compute a representation z_G such that $z_{G_1} = z_{G_2}$ if and only if G_1 and G_2 are isomorphic. In other words, the representation z_G would be a *canonical code* for the graph G.

Clearly, such a "perfect" practical GNN can't exist unless P=NP. Nevertheless, the idea of asserting the expressivity of GNNs by testing for graph isomorphism may be useful, especially in the context when most isomorphism tests are specifically producing canonical codes for the graphs.

The Weisfeiler-Leman Algorighm

The Weisfeiler-Leman (WL) algorithm [19] is a simple, yet efficient way of testing graph isomoprhism. For a given graph G, the algorithm computes a canonical code of the graph, such that two graphs are isomorphic if and only if they have the same canonical code. We will present an outline of a version of the algorithm, which ignores edge features, and requires discrete vertex features, e.g., vertex labels. Figure 2.10 provides an example of the iterative labeling process of the WL algorithm.

The algorithm works as follows:

1. Given graph G as input, we assign an initial label to each vertex, $l_0(v_i) = \zeta(v_i)$, where $\zeta(v_i)$ is the initial feature vector of the vertex v_i. If the graph does not have vertex features, or they are not discrete, set $l_0(v_i) = deg(v_i)$, where $deg(v_i)$ is the degree of the vertex v_i, i.e., the number of neighbors of v_i.

2. Iteratively assign a new label to each vertex. We compute the label of the vertex v_i at iteration t as follows:

$$l_t(v_i) = \text{Hash}(l_{t-1}(v_i), \{l_{t-1}(v_j) | v_j \in N(v_i)\})$$

where $N(v_i)$ is the set of neighbors of v_i, and Hash is a hash function that produces a new, unique label from the previous labels.

3. Repeat step 2 until convergence, i.e., until no new labels are assigned.

4. Construct the canonical code of the graph as the multiset of the labels of the vertices.

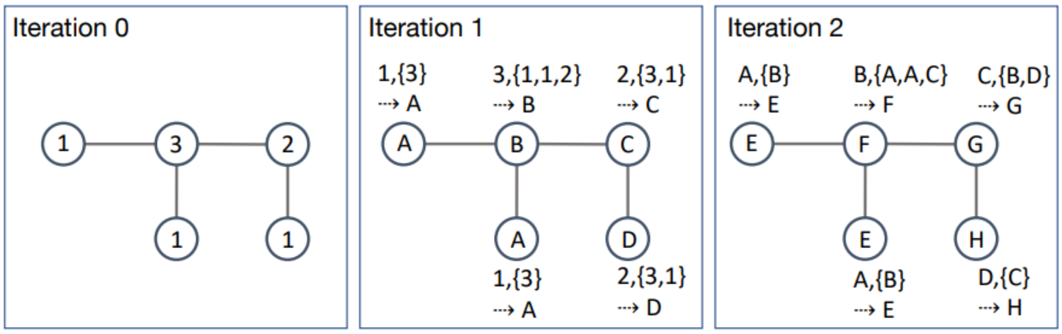


Figure 2.10 Example of the iterative labeling process of WL. Source: [19].

While the WL algorithm is quite successful, it is not complete. There are known examples of non-isomorphic graphs that the algorithm fails to distinguish. Figure 2.11 provides an example of such graphs.

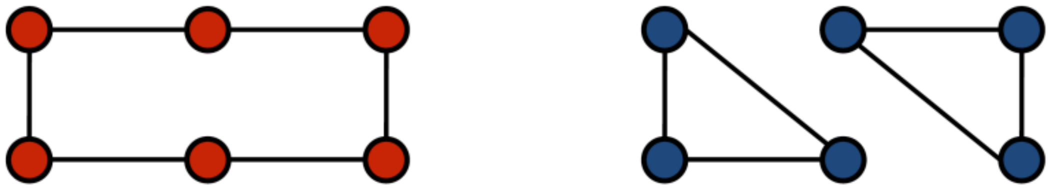


Figure 2.11 Example of non-isomorphic graphs that WL fails to distinguish. Source: [19].

Limitations of Message-passing neural networks

The Weisfeiler-Leman algorithm is similar to a message-passing neural network. Both of them iteratively use the features of a vertex, together with its neighbors, to compute a new feature for the vertex. The *Hash* function from the WL algorithm can be seen as a combination of both *Aggregate* and *Update* functions from the message-passing neural networks. In fact, message-passing neural networks have been heavily inspired by the WL algorithm, and are an attempt to generalize it to continuous and differentiable vertex features.

However, just like the WL algorithm, message-passing neural networks have their limitations. This can be formalized by the following theorem:

Theorem ([19], [28], [29]). Define a message-passing graph neural network to be any GNN that consists of K message-passing layers of the following form:

$$h_v^{(k+1)} = \text{UPDATE}^{(k)}(h_v^{(k)}, \text{AGGREGATE}^{(k)}(\{h_u^{(k)} | u \in N(v)\})),$$

where *UPDATE* is a differentiable function, and *AGGREGATE* is a differentiable, permutation invariant function. Moreover, assume that the initial vertex features are discrete and the edges have no features. Then, for any K, $h_v^{(K)} \neq h_u^{(K)}$ only if the vertices v and u have different labels after K iterations of the WL algorithm.

In other words, the expressive power of message-passing neural networks is upper-bounded by the power of the Weisfeiler-Leman algorithm. This means that, just like the WL algorithm, message-passing based GNNs can't distinguish between certain non-isomorphic graphs. This result is motivating the development of other types of GNNs, such as the one presented in this thesis.

3 Learnable Planar Graph Representations

3.1 The KHC algorithm

3.1.1 Why KHC?

Recent research has produced many polynomial time algorithms for planar graph isomorphism [30], [4], with the best-known result being in log space [4]. Most of them do so by computing a *canonical code* for a graph, such that two graphs are isomorphic if and only if they have the same code. Despite these thrilling theoretical results, the practicality of these algorithms is often limited, as they are not easily implementable in practice. Moreover, these algorithms actively modify the input graph during execution, which makes adapting them to a learnable setting difficult.

Contrary to these algorithms, the one proposed by Kukluk, Holder, and Cook [31], hereafter referred to as the KHC algorithm, is simpler in both its original form, as well as modifying it to output a representation of the graph suitable for deep learning. Just like other algorithms, it computes a canonical code for a graph. Its downside, however, is the fact that for a graph $G = (V, E)$, the algorithm runs in $O(|V|^2)$, which is theoretically worse than the best-known algorithms. We believe that this is a negligible downside, as most molecule datasets contain graphs with a number of vertices in the order of tens. Thus, a quadratic time complexity is reasonable, while the exponential time complexity of naive algorithms is not.

A recent paper [32] has also taken inspiration from the KHC algorithm by building a neural network that mimics the algorithm's behavior. However, we take a different approach by modifying the algorithm itself to output a representation of the graph, which can then be fed into a traditional deep learning model.

3.1.2 An overview of the KHC algorithm

The KHC algorithm for constructing the canonical code of a connected graph can be summarized as follows:

Algorithm 1: A short summary of the KHC algorithm

```
  Input: A connected graph $G = (V, E)$
  Output: The canonical code of the graph, a list of symbols
1 Decompose $G$ into a tree of biconnected components;
2 Decompose each biconnected component into an SPQR-tree and its
    triconnected components;
3 Compute the canonical code of each triconnected component;
4 In a bottom-up manner, merge the codes of the triconnected components
    of each SPQR-tree node to get the code of the biconnected compoennts;
5 Still, in a bottom-up manner, merge the codes of the biconnected
    components to get the code of $G$;
```

In what follows, we will describe each algorithm step in more detail. However, feel free to refer to the original paper [31] for a more in-depth explanation.

Decomposing into biconnected components

The first step of the KHC algorithm is to decompose the input, a connected graph, into its blocks-and-cuts tree [33]. This data structure is a tree, where the nodes can be of 2 types: *blocks* and *cut vertices*. A block is a biconnected component of the graph, and a cut vertex is an articulation point, i.e., a vertex whose removal would disconnect the graph. It is important to note that the articulation points appear both in cut vertices and blocks, as they may also be a part of a biconnected component. There is an edge between a block and a cut vertex if the articulation point corresponding to the cut vertex is also part of the block. See Figures 3.1 and 3.2 for an example of a graph and its blocks-and-cuts tree.

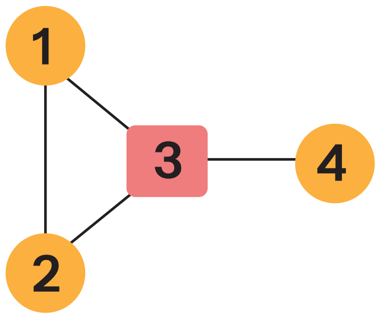


Figure 3.1 An example of a graph and its single articulation point, namely the vertex labeled “3”.

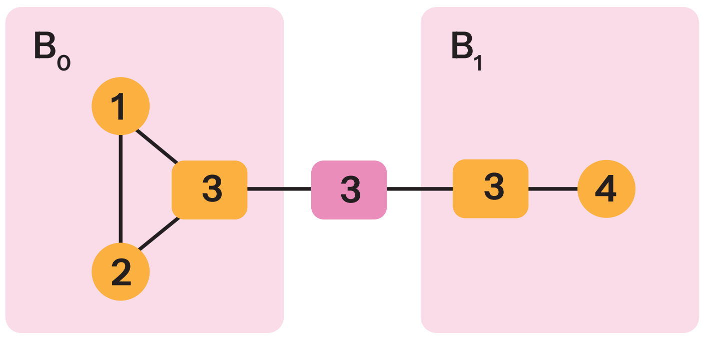


Figure 3.2 The blocks-and-cuts tree of the graph from Figure 3.1. The blocks are labeled as B_0 and B_1. The middle “3” is the cut vertex.

The code of a graph from the codes of its articulations and biconnected components

The algorithm computes the code of the input graph by merging the codes of its biconnected components and articulation points. However, their respective codes are computed in different ways. The biconnected components will be further decomposed into triconnected components to compute their codes. On the other hand, the articulation points will serve as a way to merge the codes of the biconnected components into the code of the whole graph. Thus, we will store the codes of the articulation points in a dictionary and will incrementally build their codes as we merge the codes of the biconnected components. The codes of the biconnected components will be computed and immediately added to their respective cut vertices. This step of the algorithm can be summarized as follows:

```
Algorithm 2: Merging the codes of the biconnected components
   Input: A connected graph $G = (V, E)$
   Output: The canonical code of the graph, a list of symbols
 1 Build $T$, the blocks-and-cuts tree of $G$;
 2 Initialize a dictionary $A$, from the articulation points to their codes;
 3 Initialize a dictionary $B$, from the biconnected components to their codes;
 4 while Number of nodes in $T > 1$ do
 5     foreach $C$ - leaf in $T$, a biconnected component do
 6         $B[C]$ = FindBiconnectedCode(C, A);
 7     end
 8     foreach $a$ - articulation point adjacent to a leaf $C$ in $T$ do
 9         $A[a]$append("$(_A$");
10         Append the codes of $a$'s neighbors which are leaves in $T$ to $A[a]$ in
            lexicographical order;
11         $A[a]$append("$)_A$");
12     end
13     Delete all leaves from $T$;
14     Delete from $T$ all the cut vertices with degree 1;
15 end
16 $v$ = the remaining node in $T$;
17 if $v$ is a cut vertex then
18     return $A[v]$;
19 if $v$ is a block then
20     return $B[v]$;
```

SPQR-trees

A *SPQR-tree* [34] is a tree data structure used to decompose a biconnected graph into triconnected components. In the KHC algorithm, it is used to compute the canonical code of a biconnected component by merging the codes of its triconnected components. We will follow Gutwenger's definition and implementation of a linear time construction of SPQR-trees [35]. The first step of this implementation is to make the graph directed by replacing each edge with a pair of directed edges in opposite directions.

Depending on its type (to be elaborated later), a vertex of the SPQR-tree represents a class of triconnected components. The component it refers to is called the *skeleton* of the vertex. The skeletons are directed multigraphs, i.e., they can have multiple edges between the same pair of vertices. It is important to note that the same vertex can appear in multiple skeletons. The edges of the skeletons of the SPQR-tree are of two types: *real* and *virtual.* A real edge connects two vertices inside a skeleton if and only if the said edge was present in the original graph. A virtual edge, on the other hand, is meant to represent a deeper connection between two vertices, see [35] for more details. Moreover, each virtual edge inside a skeleton has a twin virtual edge in the skeleton of another, connecting the same two vertices, in the same direction. The edges of the SPQR-tree are edges in between pairs of twin edges.

The nodes of the SPQR-tree are of three types:

- **S-nodes** represent a cycle graph on three or more vertices. "S" stand for "Series".
- **P-nodes** represent a dipole multigraph, i.e., a graph on two vertices with one or more parallel edges. "P" stand for "Parallel".
- **R-nodes** represent a triconnected component other than the above 2. "R" stand for "Rigid".

Originally, there was also a **Q-node** type associated with a trivial graph on two vertices with a single edge. The implementation we use omits this type since it is redundant, as it can be thought of as a special case of a P-node. See Figure 3.3 for an example of a biconnected graph and its corresponding SPQR-tree.

The code of a biconnected component from the codes of its triconnected components

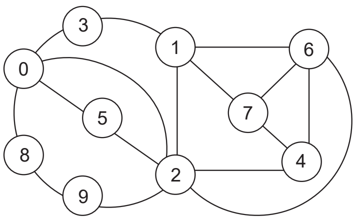
The algorithm will compute the canonical code of a biconnected component by traversing its corresponding SPQR-tree. We start from the *center* of the tree, which is defined as the middle node in every longest path in the tree. There may be two centers, in which case we compute the codes for both centers and choose the lexicographically smallest one.

The algorithm will then traverse the SPQR-tree from the center to the leaves, in order to compute the codes of the triconnected components. Depending on the type of the node, and whether it is a center or not, we will compute the code of the triconnected component in different ways. Figure 3.4 shows the codes of the triconnected components of the graph from Figure 3.1. Figure 3.5 shows the codes of the biconnected components of the same graph.

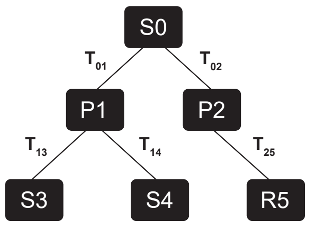
Weinberg's algorithm

The KHC algorithm uses as a subroutine Weinberg's algorithm [36] for computing the canonical code of a triconnected planar graph, and thus we will also present an overview of Weinberg's algorithm.

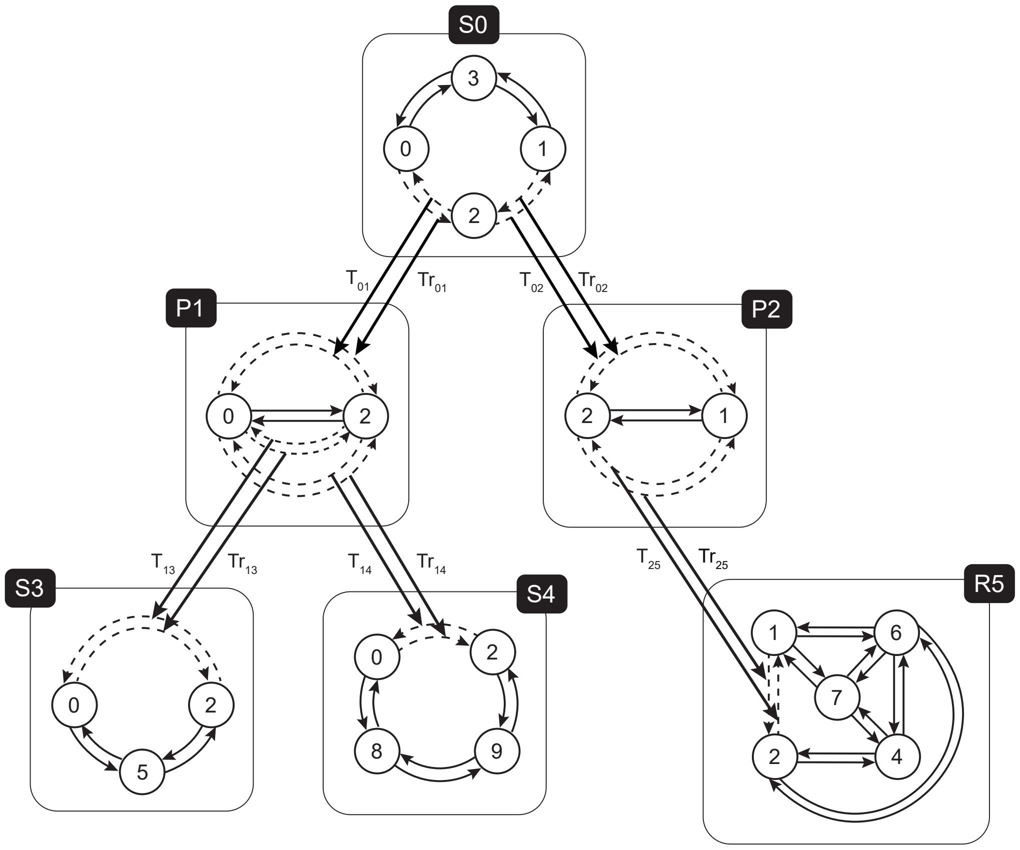
Recall that a *planar embedding* of a graph is a drawing of the graph in the plane such that no edges cross. In computer science, we represent such an embedding



(a) A biconnected graph



(b) SPQR-tree



(c) Detailed view of the skeletons and the edges of the SPQR-tree

Figure 3.3 An example of a biconnected graph and its corresponding SPQR-tree

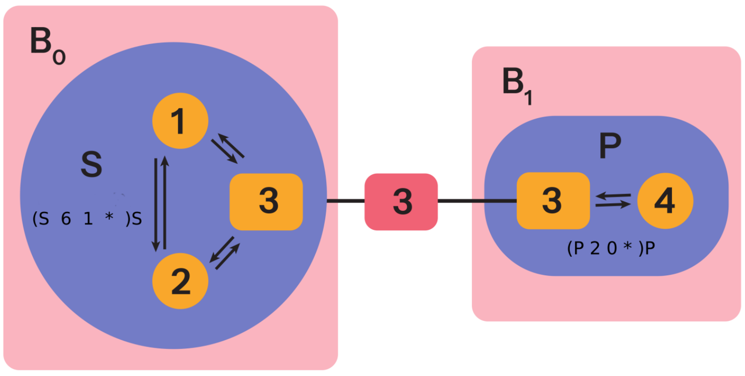


Figure 3.4 The codes of the triconnected components of the graph from 3.1.

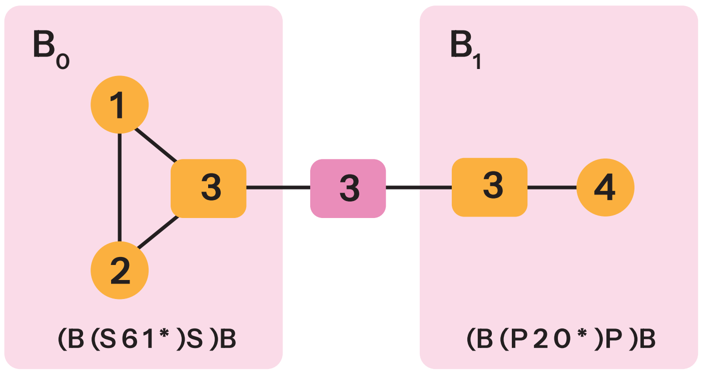


Figure 3.5 The codes of the biconnected components of the graph from 3.1.

as a dictionary, where each vertex is associated with a list of its neighbors. In this list, the neighbors are ordered in the counterclockwise order in which they appear in the embedding. Figure 3.6 shows an example of the two unique planar embeddings of the same graph.

By Whitney's theorem [37], a triconnected planar graph (which is not a cycle graph or a dipole graph) has a unique planar embedding, up to equivalence. Weinberg's algorithm [36] takes advantage of this fact to compute a canonical code for a triconnected planar graph.

The algorithm starts by substituting every edge of the graph with a pair of directed edges in opposite directions. This way, we obtain a directed, strongly connected graph, for which the in-degree of each vertex is equal to its out-degree. According to Kőnig [38], this is a necessary and sufficient condition for a directed graph to have an Eulerian circuit, i.e., a path that visits every edge exactly once (in the direction of the edge), and returns to the starting vertex. We use this fact to compute two codes (to exhaust the equivalence of embeddings) for each edge of the digraph, each code being associated to a particular Eulerian circuit. We call these codes *code going right* and *code going left* for the edge in question. We will describe the procedure of computing the *code going right* for an edge; the

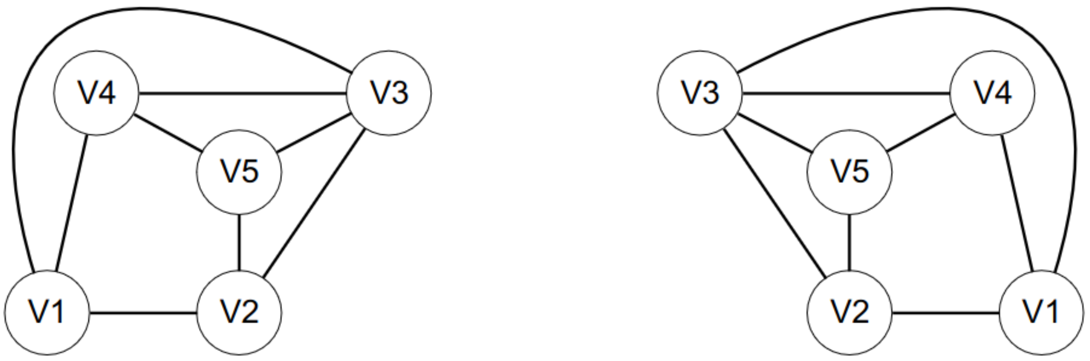


Figure 3.6 Two unique planar embeddings of the same graph. Source: [31].

computation of the *code going left* is symmetric. We define the *first neighbor to the right* of a vertex v after reaching it via an edge (u, v) to be the neighbor of v in the adjacency list of v from the planar embedding that appears immediately after u.

While traversing the Eulerian circuit, we will assign new labels to the vertices, and add them to the code, obeying the following rules:

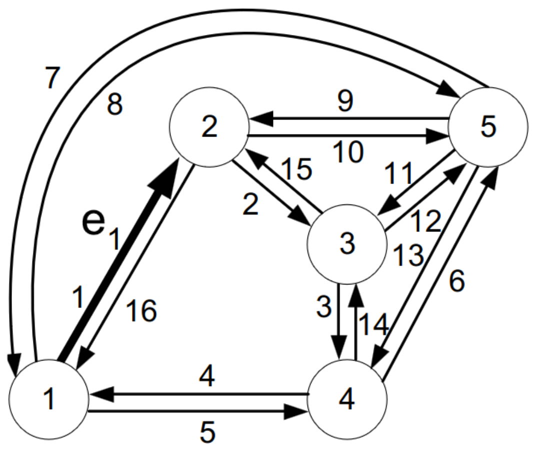
- The starting vertex is labeled 1.
- When we traverse an edge, if the target vertex has not been labeled yet, we assign it the smallest natural number that has not been used yet.
- Otherwise, we assign the target vertex the label it already has.

The Eulerian circuit for an edge (v_1, v_2) is travered in the following way:

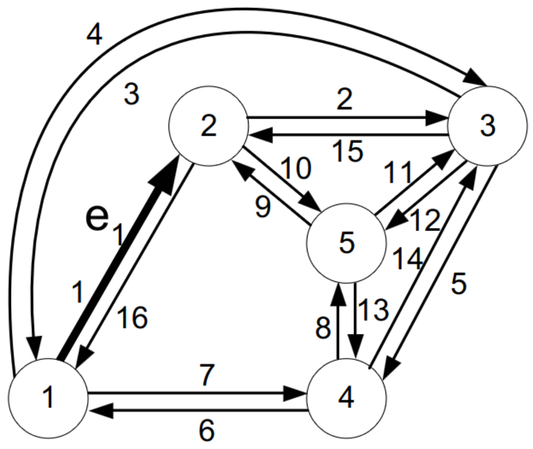
- Start ar v_1 and first traverse the edge (v_1, v_2).
- When reached a vertex v_i via the edge (v_{i-1}, v_i), we do the following:
  - If v_i has not been visited yet, exit it via the first neighbor to the right.
  - If v_i has been visited, and the reverse edge (v_i, v_{i-1}) has not been traversed yet, traverse it.
  - Otherwise, exit v_i via the first **unused** neighbor to the right.

Obviously, we stop when we have traversed all the edges of the digraph.

This way, we compute the codes *going right* and *going left* for each edge of the digraph. We choose the lexicographically smallest code to be the canonical code of the whole triconnected graph. It is able to uniquely identify the graph, up to isomorphism. Figure 3.7 shows an example of the application of the algorithm on a planar embedding.



(a) e_1 code going right=
1 2 3 4 1 4 5 1 5 2 5 3 5 4 3 2 1



(b) e_1 code going left=
1 2 3 1 3 4 1 4 5 2 5 3 5 4 3 2 1

Figure 3.7 An example of a triconnected planar graph and its canonical code computed by Weinberg's algorithm. Source: [31].

Algorithm 3: Computing the code of an edge inside a S-node

```
def code_S_edge(e_in, stop_edge, virtual_edge_codes, skeleton, A):
    Input: The starting edge e_in of the tour of the S-node,
    the edge stop_edge where the tour stops, the precomputed codes of
    the virtual edges,
    the skeleton of the S-node,
    the dictionary of articulation points
    Output: The code associated to the edge e_in
    Initialize code as an empty string;
    code.append("$(_S$");
    /* The number of edges in the skeleton                         */
    code.append(skeleton.size());
    Initialize tour_edge_codes as an empty list;
    Initialize tour_counter as 0;
    while True do
        if e_in is a virtual edge then
            code.append(tour_counter);
            tour_edge_codes.extend(virtual_edge_codes[e_in]);
        end
        v = the end vertex of e_in;
        if v is an articulation point then
            code.append(tour_counter);
            code.append("$*$");
            code.append(A[v]);
        end
        e_in = the next edge from v in the same direction as e_in;
        tour_counter += 1;
        if e_in == stop_edge then
            break;
        end
    end
    code.extend(tour_edge_codes);
    code.append("$)_S$");
    return code;
end
```

Algorithm 4: Computing the code of a S-node

```
1  def code_S_center(center, A, spqr_tree):
       Input: A central S-node,
       the dictionary of articulation points,
       the SPQR-tree
       Output: The code of a central S-node
2      Initialize virtual_edge_codes as an empty dictionary;
3      foreach e - virtual edge in the skeleton of center do
4          virtual_edge_codes[e] = code_virtual(e, center, A, spqr_tree);
5      end
6      return min{code_S_edge(next edge after e, next edge after e,
         virtual_edge_codes, center, A) | edge e ∈ skeleton of center}
7  end
8  def code_S_non_center(e_in, node, A, spqr_tree):
       Input: An S-node that is not a center,
       the dictionary of articulation points,
       the SPQR-tree
       Output: The code of a non-central S-node
9      Initialize virtual_edge_codes as an empty dictionary;
10     foreach e - virtual edge in the skeleton of node do
11         virtual_edge_codes[e] = code_virtual(e, node, A, spqr_tree);
12     end
13     return code_S_edge(next edge after e_in, e_in,
         virtual_edge_codes, node, A)
14 end
```

Algorithm 5: Computing the code of a vertex inside a P-node

```
def code_P_vertex(source_vertex, virtual_edge_codes, skeleton, A):
    Input: The source vertex of the tour of the P-node,
    the precomputed codes of the virtual edges,
    the skeleton of the P-node,
    the dictionary of articulation points
    Output: The code associated to the vertex source_vertex
    Initialize code as an empty string;
    code.append("$(_P$");
    /* The number of edges in the skeleton                    */
    code.append(skeleton.size());
    code.append(Number of virtual edges in the skeleton);
    tour_codes = codes of virtual edges from virtual_edge_codes that are
      going out of source_vertex;
    Append the codes from tour_codes to code in lexicographical order;
    if source_vertex is an articulation point then
        code.append("$*$");
        code.append(A[source_vertex]);
    end
    /* The other vertex of the dipole                         */
    if sink_vertex is an articulation point then
        code.append("$*$");
        code.append(A[sink_vertex]);
    end
    code.append("$)_P$");
    return code
end
```

Algorithm 6: Computing the code of a P-node

```
1  def code_P_center(center, A, spqr_tree):
       Input: A central P-node,
       the dictionary of articulation points,
       the SPQR-tree
       Output: The code of a central P-node
2      Initialize virtual_edge_codes as an empty dictionary;
3      foreach e - virtual edge in the skeleton of center do
4          virtual_edge_codes[e] = code_virtual(e, center, A, spqr_tree);
5      end
6      return min{code_P_vertex(source vertex of e, virtual_edge_codes,
         center, A) | vertex source_vertex ∈ skeleton of center}
7  end
8  def code_P_non_center(e_in, node, A, spqr_tree):
       Input: A P-node that is not a center,
       the dictionary of articulation points,
       the SPQR-tree
       Output: The code of a non-central P-node
9      Initialize virtual_edge_codes as an empty dictionary;
10     foreach e - virtual edge in the skeleton of node do
11         virtual_edge_codes[e] = code_virtual(e, node, A, spqr_tree);
12     end
13     return code_P_vertex(source vertex of e_in, virtual_edge_codes,
         node, A);
14 end
```

Algorithm 7: Computing the code of an edge inside a R-node

```
1  def code_R_edge(e_in, virtual_edge_codes, skeleton, A):
       Input: The starting edge e_in of the tour of the R-node,
       the precomputed codes of the virtual edges,
       the skeleton of the R-node,
       the dictionary of articulation points
       Output: The code associated to the edge e_in
2      Initialize code as an empty string;
3      code.append("$(_R$");
4      Apply Weinberg's algorithm to the skeleton of the R-node to get the
         canonical code of the triconnected component;
5      Compute both the code going right and the code going left starting
         from e_in;
6      if at any vertex v during Weinberg's algorithm we encounter an
         articulation point then
7          code_weinberg.append("∗");
8          code_weinberg.append(A[v]);
9      end
10     code.extend(min{code going right, code going left});
11     code.append("$)_R$");
12     return code;
13 end
```

Algorithm 8: Computing the code of a R-node

```
def code_R_center(center, A, spqr_tree):
    Input: A central R-node,
    the dictionary of articulation points,
    the SPQR-tree
    Output: The code of a central R-node
    Initialize virtual_edge_codes as an empty dictionary;
    foreach e - virtual edge in the skeleton of center do
        virtual_edge_codes[e] = code_virtual(e, center, A, spqr_tree);
    end
    return min{code_R_edge(e, virtual_edge_codes, center, A) | edge
     e ∈ skeleton of center}
end
def code_R_non_center(e_in, node, A, spqr_tree):
    Input: An R-node that is not a center,
    the dictionary of articulation points,
    the SPQR-tree
    Output: The code of a non-central R-node
    Initialize virtual_edge_codes as an empty dictionary;
    foreach e - virtual edge in the skeleton of node do
        virtual_edge_codes[e] = code_virtual(e, node, A, spqr_tree);
    end
    return code_R_edge(e_in, virtual_edge_codes, node, A)
end
```

The output of the KHC algorithm is a list of integers and special symbols, which can then be used to test for graph isomorphism.

Each element of the list represents one of the following:

- A natural number. Depending on the context, it may represent different things, e.g., the number of vertices in the skeleton of a S-node, the id of a virtual edge, etc.
- A special symbol, as follows:
  - $(_S$, $)_S$ - the beginning and end of the code of a S-node
  - $(_P$, $)_P$ - the beginning and end of the code of a P-node
  - $(_R$, $)_R$ - the beginning and end of the code of a R-node
  - $(_B$, $)_B$ - the beginning and end of the code of a biconnected component
  - $(_A$, $)_A$ - the beginning and end of the code of an articulation point
  - $*$ - the encounter of an articulation point

3.2 A modification of the KHC algorithm

3.2.1 C-KHC: Handling disconnected graphs

Many datasets of graphs, and even molecules, contain disconnected graphs. This may seem counterintuitive, as a molecule should be connected, but it depends on what an edge represents in the corresponding datasets. For example, some datasets may consider only covalent bonds to be edges, thus an abundance of other types of bonds may lead to a disconnected graph.

The KHC algorithm is designed to work on connected graphs; therefore, we must modify it to handle disconnected graphs. We achieve this in the following way:

Algorithm 9: C-KHC: Adjusted KHC for disconnected graphs

```
  Input: A graph $G = (V, E)$
  Output: The canonical code of the graph, a list of symbols
1 Decompose $G$ into connected components $G_1, G_2, \dots, G_k$;
2 for $i = 1$ to $k$ do
3     Compute the canonical code of $G_i$ using the KHC algorithm;
4     Add special symbols $(_C$, $)_C$ to the beginning and end of the code to
        mark the code of a connected component;
5 end
6 Sort the codes of the connected components in lexicographical order;
7 Concatenate the codes of the connected components, in the order of the
    sorted list;
```

Theorem 1. Let G_1, G_2 be 2 planar graphs, and let z_1, z_2 be their canonical codes computed using the adjusted KHC algorithm. Then G_1 is isomorphic to G_2 if and only if $z_1 = z_2$.

Proof. Clearly, by adding the special symbols $(_C$, $)_C$ to the beginning and end of the code of each connected component, the KHC algorithm maintains the property that the code of a graph is unique up to isomorphism.

$\Rightarrow$: Let C_1 be the set of connected components of G_1, and C_2 the set of connected components of G_2. Since G_1 is isomorphic to G_2, there exists a bijection $f : C_1 \mapsto C_2$ such that for each connected component $c_i \in C_1$, $f(c_i)$ is isomorphic to c_i. Let $khc(c_i)$ be the canonical code of a connected component c_i computed using the modified KHC algorithm. By the correctness of the KHC algorithm, we have that $khc(c_i) = khc(f(c_i)) \forall c_i$. Thus, the sets $Z_1 = \{khc(c_i) | c_i \in C_1\}$ and $Z_2 = \{khc(c_i) | c_i \in C_2\}$ are equal, and therefore a string made out of the concatenation of their elements in a lexicographical order is the same for both G_1 and G_2.

$\Leftarrow$: Let $z_1 = z_2$. Each substring of both z_1 and z_2 that is enclosed in $(_C$, $)_C$, without any of these two characters inside the respective substring, represents the canonical code of a connected component. Let Z_1 be the list of these substrings in z_1, and Z_2 the list of these substrings in z_2, in the order of their appearance in the respective string. Since $z_1 = z_2$, we have that $Z_1 = Z_2$. Each element of Z_1 is the canonical code of a connected component of G_1, and each element of Z_2 is the canonical code of a connected component of G_2. Thus, by the correctness of the KHC algorithm, and because $Z_1 = Z_2$, we have that each connected component of G_1 is isomorphic to the corresponding connected component of G_2. Therefore, G_1 and G_2 are unions of isomorphic connected components and thus are isomorphic themselves.

□

3.2.2 L-KHC: KHC for learnable representations of planar graphs

The canonical code computed by the KHC algorithm is a list of integers and special symbols, which is unsuitable for deep learning techniques. We propose one more modification to the KHC algorithm, which will be called L-KHC, which will output a representation of a planar graph that is suitable for deep learning, and which still maintains the property that two graphs are isomorphic if and only if they have the same representation. We achieve this in 2 steps:

- Whenever adding a symbol C to the code, be it an integer or a special symbol, we instead add a pair of the form (C, F), where F is a feature vector associated with the symbol C. For example, when adding a special symbol $(_S$, we instead add $((_S, F)$, where F is a feature vector (a one-hot encoding) associated with the special symbol $(_S$. The KHC algorithm, however, needs its elements to be totally ordered, as it uses lexicographical orders of sub-codes to compute the canonical code. To achieve this, we define a total order on the set of all pairs. Thus, for 2 pairs (C_1, F_1) and (C_2, F_2), we have that $(C_1, F_1) < (C_2, F_2)$ if and only if $C_1 < C_2$, and $(C_1, F_1) = (C_2, F_2)$ if and only if $C_1 = C_2$. In other words, the second element of the pair is completely ignored during the computation of the canonical code, and will only be used in the deep learning model. We then define the total order of the codes in the following way: 2 codes z_1 and z_2 are equal if and only if they have the same length, and for each i, the i-th element of z_1 is equal to the i-th element of z_2.
- The codes of S, P, and R nodes encompass a traversal of the associated triconnected component, where each vertex is visited at least once. We may thus add the features of all nodes to the code, by adding a pair of the form $(0, F_v)$ for each vertex v visited during the traversal, where F_v is the feature vector associated with the vertex v. Since 0 is the same key for all vertices, and because 0 cannot be produced anywhere else in the code, this addition will not affect the code, and we will prove it formally in the following theorem.

Theorem 2. Let G_1, G_2 be two planar graphs, and let z_1, z_2 be their representations computed using the modified, learnable version of the KHC algorithm. Then G_1 is isomorphic to G_2 if and only if $z_1 = z_2$.

Proof. It is clear that the first item of the modification maintains this property, for the second member of the pair is completely ignored during the computation of the canonical code. We will then focus on the second item of the modification, more specifically on the addition of the features of the vertices to the code.

Let A be the set of possible codes that the original version of the KHC algorithm can output. Let B be the set of possible codes that C-KHC can output. We'll prove that our modification, namely $f : A \mapsto B$, is a bijection. Let $z_1, z_2 \in A$ be 2 codes such that $z_1 \neq z_2$. Since for any $z \in A$, $f(z)$ is obtained by adding elements of the form $(0, F)$, we have that $f(z_1) \neq f(z_2)$. Thus, f is injective. Let $t \in B$. t is obtained by adding elements of the form $(0, F)$ to the code during the traversal of a triconnected component, without changing the lexicographical order of the respective code. Thus, there exists a code $z \in A$ such that $f(z) = t$, and therefore f is surjective. Since f is both injective and surjective, it is a bijection. By the correctness of the original KHC algorithm, it then follows the statement of the theorem.

$\square$

To obtain a learnable representation of a planar graph, we will simply use L-KHC algorithm, which outputs a list z of pairs for a graph G. We then define the feature vector associated with G to be $x = [F_1, F_2, \ldots, F_k]$, where F_i is the feature vector associated wtih the i-th element of z, and k is the length of z.

Since the graph level feature x consists of a vector of features of both vertices and special symbols, we need a way to map them to a common vector space. We will use simple one-hot embeddings for the special symbols. Assume a dataset of graphs contains vertex features of dimension d_v. Then, the dimension of $F_i \in x$ is $d_v + 14$, as there are 13 special symbols, and the 14-th additional dimension is for integers. Figure 3.8 shows the feature vector associated with the special symbol of the beginning of a P-node, and Figure 3.9 shows the feature vector associated with a vertex.

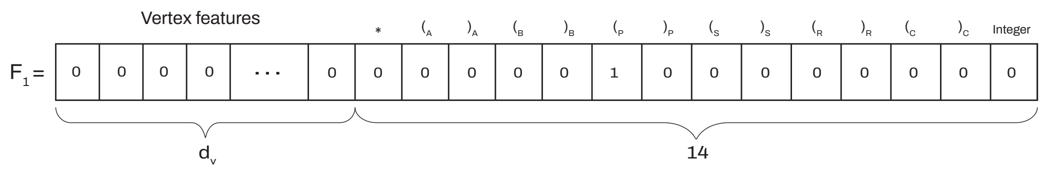


Figure 3.8 The feature vector associated with the special symbol of the beginning of a P-node.

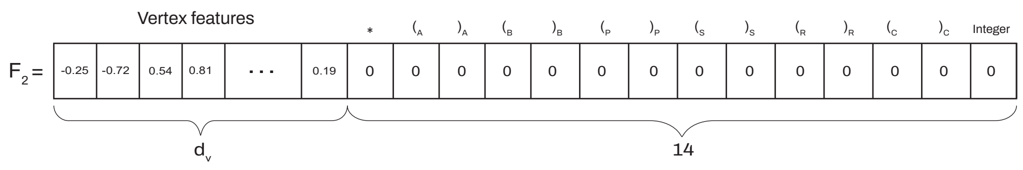


Figure 3.9 The feature vector associated with a vertex.

3.3 Planar GNN

Traditional GNNs are a separate branch of deep learning. Although some GNN architectures were inspired by advances in deep learning, e.g., the graph attention networks, graph convolutions, etc., the branch was developing independently of the whole field of machine learning. The novelty of the Planar GNN architecture that we propose is that it can help reunifying the fields of graph neural networks and the rest of deep learning. This is because L-KHC outputs a representation of a planar graph, which is analogous to the adjacency matrix or adjacency vector representations, but which can then be used by any sequential neural network architecture, e.g., RNNs, transformers, etc.

The Planar GNN pipeline consists of 2 steps: preprocessing and the Planar GNN architecture itself. The preprocessing step is the L-KHC algorithm, which for a given graph dataset, outputs a vector of the representations of the graphs in the dataset. This step is visualized in Figure 3.10. This procedure must be done only once for any given dataset, as the representations can be saved and reused for further experiments. As for the Planar GNN architecture, its only requirement is to be able to process sequential data. The architecture we propose is shown in Figure 3.11. It can briefly be described as follows:

1. An input to the model is a sequence of vectors of shape (L, F_{in}), where L is the length of the sequence, and F_{in} is the dimension of the input features.
2. Several linear layers are applied to the input sequence in order to obtain a sequence of vectors of shape (L, F_l), where F_l is the dimension of the output features. This is done in order to increase the dimensionality of the input features and to allow the model to learn more complex patterns.
3. The sequence of vectors is then passed through a recurrent neural network, e.g., an LSTM, GRU, etc. We then discard all but the last hidden state of the RNN, which is a vector of shape $(F_r, 1)$. This vector summarizes all the information about the input graph.
4. The vector is then passed through a series of linear layers in order to obtain the desired output. The output can be a vector of shape $(F_{out}, 1)$, where F_{out} is the dimension of the output features. Alternatively, the output can be a scalar, a binary value, etc., depending on the task.

Back to the relationship between graph isomorphism and graph neural networks, the Planar GNN architecture is a so-called “Perfect GNN”, as it can perfectly distinguish between non-isomorphic graphs. This is because the L-KHC algorithm, which serves as the preprocessing step of the architecture, outputs a representation of a planar graph that is unique up to isomorphism.

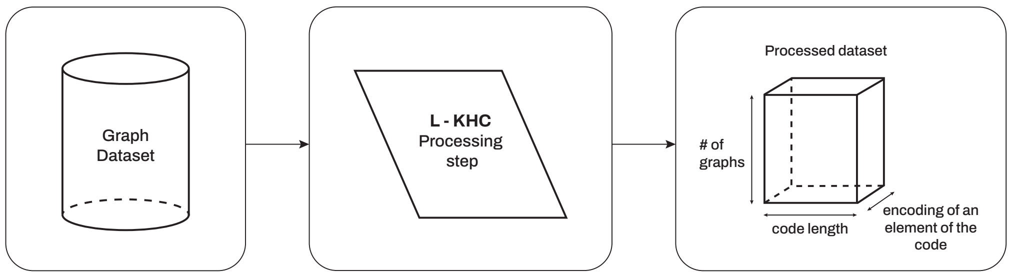


Figure 3.10 The preprocessing step of the Planar GNN architecture.

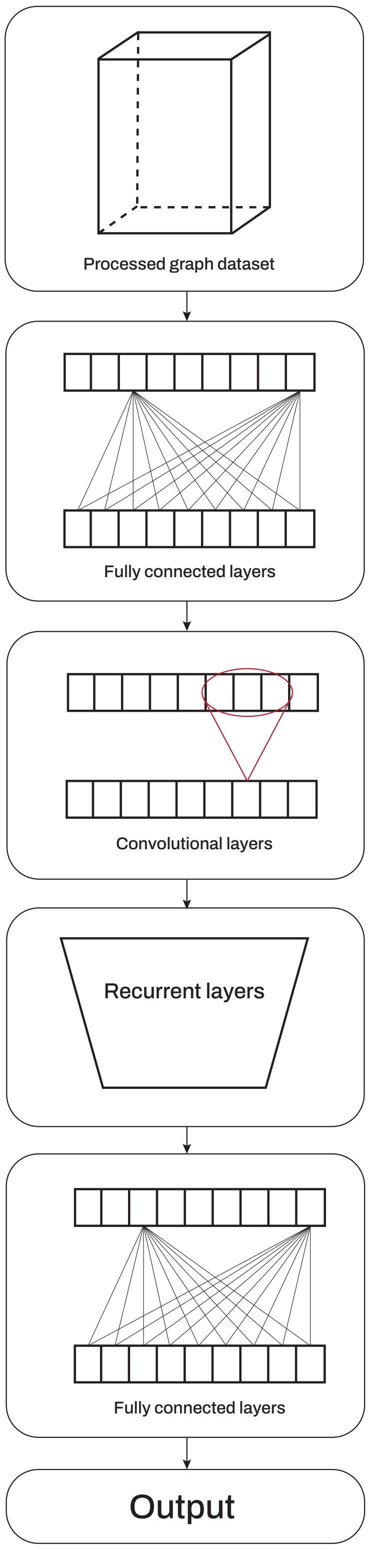


Figure 3.11 The Planar GNN architecture.

4 Implemented Software

4.1 GraphMindKeras

Introduction

This text outlines the software package called "GraphMindKeras," (A.1) that we have developed as a support tool for this Bachelor's thesis. This software project aimed to provide a toolkit for the study and development of Graph Neural Networks. It consists of well-known, classical GNN models, such as the Graph Convolutional Network (GCN) and Graph Attention Network (GAT).

The toolkit is written in Python 3 [39], uses the Keras framework [40], and is implemented with the help of only Keras operations. We chose Keras, because we wanted to address a large pool of potential users, and not limit it to a specific back-end framework, e.g., PyTorch [41] or TensorFlow [42]. In conclusion, we didn't choose a back-end, at least at this stage. On the contrary, Keras is back-end agnostic, meaning it can run on top of any of the aforementioned frameworks, which boosts sustainability in the realm of rapidly evolving deep learning frameworks. Thus, the users of GraphMindKeras can choose the back-end that best suits their needs, and the code will run just fine regardless. The choice of the back-end is done simply by specifying an environment variable. We provide examples for both TensorFlow and PyTorch back-ends (A.1). Despite other back-end agnostic GNN libraries, such as DeepGraph [43], GraphMindKeras is better integrated with its back-end, providing the user more flexibility and control over the training process.

We focus on data from TUDataset [44], which is a collection of graph datasets, but any dataset of graphs obeying the same format (to be specified later) can be used.

Memory layout

Traditionally, in machine learning, graphs are represented via their adjacency matrix, which facilitates the implementation of many algorithms. The drawback of this representation is that it is not memory efficient, as it requires $\Theta(N^2)$ memory (i.e., both worst-case and best-case), regardless of the number of edges in a graph, where N is the number of nodes in the graph. In practice, however, most graphs are sparse, and thus the adjacency matrix is mostly empty, leading to a waste of memory. To address this issue, we have used a more memory-efficient representation, the edge list, which requires $\Theta(E)$ memory, where E is the number of edges in the graph. Therefore, when taking into account the requirements for machine learning, a graph is represented via the following tensors:

- **Node features**: a tensor of shape (N, F_v), where N is the number of nodes in the graph, and F_v is the dimension of the node features.
- **Edge list**: a tensor of shape $(E, 2)$, where E is the number of edges in the graph, and each row represents an edge, with the first element being the source node, and the second element being the target node.
- **Node degrees** *(only for convolution)*: a tensor of shape $(N, 1)$, where each element represents the degree of the corresponding node.

When working with a dataset of graphs, the hardware usually requires that all graphs are of the same size. This considerably facilitates computations on a GPU. We achieve this by padding the corresponding tensors with zeros. Thus, we pad the edge list with zeros, and its final shape is $(E_{max}, 2)$, where E_{max} is the maximum number of edges in the dataset. Now, since we have used the 0-th vertex to denote the padding, i.e., the lack of a connection in the edge list, we require that the 0-th (first) row of the node features and node degrees tensors to be filled with zeros as well. This is because the dummy 0-th vertex shouldn't have any features. Additionally, we still have to pad the latter two tensors with zeros, to account for the variable size of graphs, and their final shapes are $(N_{max} + 1, F_v)$ and $(N_{max} + 1, 1)$, respectively, where N_{max} is the maximum number of nodes in the dataset, and "+1" is due to empty 0th row.

To facilitate the loading of datasets into memory, we provide the **read_dataset()** function. It reads a dataset in the TUDataset format [44] from the hard-disk, processes it, and returns the node features, edge list, node degrees, and graph features tensors representing the dataset.

A graph dataset in the TUDataset format consists of multiple files, as follows:

- **A.txt**: each line corresponds to an edge, and the first element is the source node, and the second element is the target node.
- **graph_indicator.txt**: the value on the i-th line is the graph id of the i-th node.
- **graph_labels.txt**: the value on the i-th line is the class label of the i-th graph.
- **node_labels.txt**: the value on the i-th line is the class label of the i-th node.
- more optional files, adding more properties to the nodes or edges, or even the graphs.

Figure 4.1 provides an example of the above-discussed TUDataset format encoding.

Keras GNN layers

The main building block of Keras models is layers (**keras.layers**). They represent an abstract operation on some tensors, e.g., a matrix multiplication, concatenation of tensors, an activation function, etc. Keras layers can form complex networks, together forming a model (**keras.Model**), which in turn orchestrates the layers.

To obey this framework, we have implemented all our GNN operations as classes inheriting from **keras.layers.Layer**. This allowed them to be integrated into the whole Keras ecosystem, and thus their API is the same as the one of traditional Keras layers.

A goal of of this project was to keep the memory footprint minimal, and to achieve this, the layers operate in a gather-scatter fashion. They first gather the necessary information from the input tensors, perform the intended operations, and scatter the results back into output tensors. At no point during the execution

A.txt	**graph_labels.txt**
1,2	
1,3	
2,3	
3,4	
4,5	1
3,5	0
6,7	
7,8	
8,9	
7,9	
9,10	
graph_indicator.txt	**node_labels.txt**
1	0
1	1
1	2
1	1
1	0
2	0
2	1
2	1
2	1
2	0

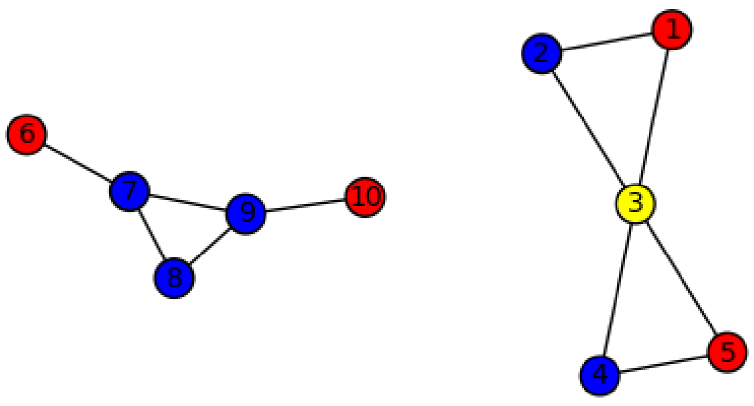


Figure 4.1 An example of a dataset consisting of the butterfly and bull graphs in the TUDataset format. Different colors represent different node labels.

of the layers, a data structure of size $\Theta(N^2)$ is created, e.g., the adjacency matrix, attention coefficients, etc.

The following GNN layers are implemented. They all accept the common Keras layer arguments, such as **name**, **trainable**, etc.:

- **GatherNodes()**. The constructor accepts no additional arguments (other than the default Keras layer arguments). The **call()** method accepts as input a tuple of 2 tensors. The first tensor is the node features tensor of shape (N, F_v), and the second tensor is the edge list tensor of shape $(E, 2)$. The method returns a tensor of shape $(E, 2, F_v)$, where each row represents the source and target node features of an edge.

- **ReduceGatheredNodesSum()**. The constructor accepts no additional arguments. The **call()** method accepts as input a tuple of 3 tensors. The first tensor is the tensor returned by the **GatherNodes** layer, the second tensor is the node features tensor of shape (N, F_v), and the third tensor is the edge list tensor of shape $(E, 2)$. The method returns a tensor of shape (N, F_v), where each row represents the sum of the features of the neighbors of the node at the corresponding index.

- **ReduceNodesSum()**. The constructor accepts no additional arguments. The **call()** method accepts as input a single tensor: the node features tensor of shape (N, F_v). The method returns a tensor of shape $(F_v, 1)$, which represents the sum of the node features of the whole graph.

- **ApplyOverBatch(layer)**. The constructor accepts an instance of a Keras layer as an argument. The **call()** method accepts as inputs the same arguments as the input layer, but with an additional dimension at the beginning, which represents the batch size. **ApplyOverBatch** unstacks the batch of graphs, and iteratively applies the input layer over each graph. It then stacks the results back together.

- **SingleGraphConvolution(units, activation, use_bias)**. The constructor accepts three additional arguments: the number of output units (the output dimension), the activation function to be applied to the output, and a boolean flag indicating whether to use a bias term. The **call()** method applies the convolution operation [20] over a single graph. It accepts as input a tuple of 3 tensors: the node features tensor of shape (N, F_v), the edge list tensor of shape $(E, 2)$, and the node degrees tensor of shape $(N, 1)$. The method returns a tensor of shape $(N, \textbf{units})$. It uses internally the **GatherNodes** and **ReduceGatheredNodesSum** layers.

- **SingleHeadAttention(output_dim, activation, use_bias)**.
The constructor accepts three additional arguments: the output dimension, the activation function to be applied to the output, and a boolean flag indicating whether to use a bias term. The **call()** method applies the attention operation [21] over a single graph. It accepts as input a tuple of 2 tensors: the node features tensor of shape (N, F_v), and the edge list tensor of shape $(E, 2)$. The method returns a tensor of shape $(N, \textbf{output\_dim})$. It uses internally the **GatherNodes** and **ReduceGatheredNodesSum** layers.

- **MultiHeadAttention(output_dim, num_heads, activation, use_bias)**. The constructor accepts 4 additional arguments: the output dimension, the number of heads, the activation function to be applied to the output, and a boolean flag indicating whether to use a bias term. The **call()** method applies the multi-head attention operation [21] over a single graph, by internally applying **num_heads SingleHeadAttention** layers, and concatenating the results. It accepts as input a tuple of 2 tensors: the node features tensor of shape (N, F_v), and the edge list tensor of shape $(E, 2)$. The method returns a tensor of shape $(N,$ **output_dim**$)$.

Examples

A few examples of how to use the toolkit on a dataset of graphs from TUDataset are provided in the **examples** folder (A.1). It includes examples of both the GCN and GAT layers, as well as the **read_dataset()** function. To prove that the toolkit is back-end agnostic, we have provided examples for both TensorFlow (filenames with a "tf" suffix) and PyTorch (filenames with a "torch" suffix) back-ends.

Demo

A typical usage of "GraphMindKeras" looks as follows:

1. Download a dataset from the TUDataset website (A.2). Alternatively, use a dataset of graphs obeying the same format.
2. Load the dataset using the **read_dataset()** function.
3. Define the inputs to the neural network by specifying their shapes via the **keras.Input** layer.
4. Build the model via the Keras API, by specifying all the layers and their connections.
5. Compile the model by specifying the optimizer, loss function, and metrics.
6. Train the model by calling the **.fit()** method.
7. Test the model on unseen data by calling the **.evaluate()** method.
8. Save the model by calling the **.save(path)** method.
9. Load the model by calling the **keras.models.load_model(path)** method.
10. Use the model for inference by calling the **.predict()** method.

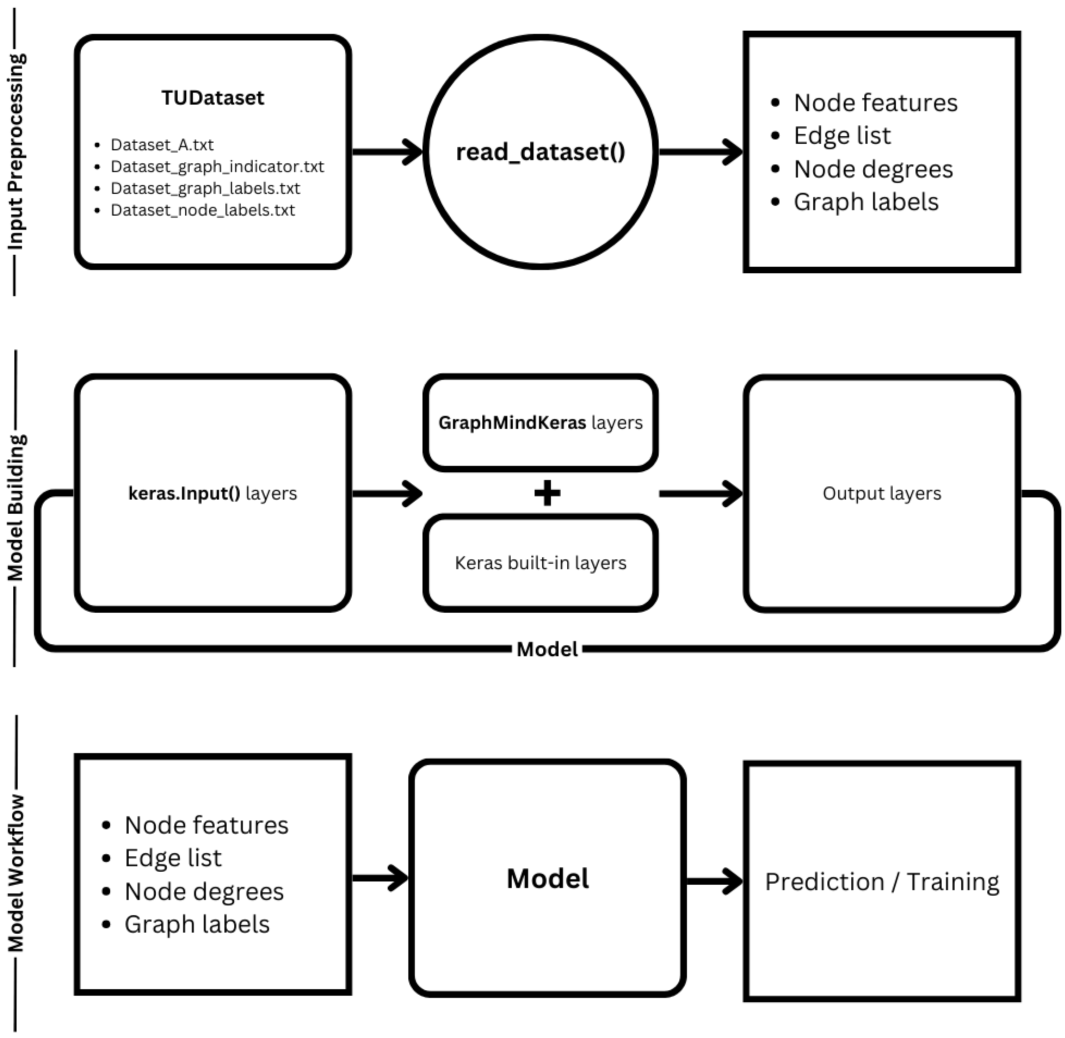


Figure 4.2 A diagram of the workflow of GraphMindKeras. *TUDataset* depicts a dataset stored in the TUDataset format. *read_dataset()* is the function that reads the dataset from the hard-disk. To create a Keras model, we utilize both built-in layers, as well as GraphMindKeras layers. The model is then compiled and trained on the respective dataset.

Here, we'll also provide a simple example of reading a dataset, building a model, and training it on a dataset of graphs, while using PyTorch as the back-end.

```
import os
os.environ["KERAS_BACKEND"] = "torch"

import keras
import numpy as np
from keras import ops
from keras import optimizers
from model.layers.convolution import *
from model.layers.common_layers import *
from model.data.read_dataset import read_dataset


# Load the dataset
dataset_path = "data/NCI109"

(
    node_features_ds,
    edge_list_ds,
    degrees_ds,
    graph_features_ds,
) = read_dataset(
    dataset_path,
    include_node_attributes=False,
    include_node_labels=True,
)

# define the model via the functional API
# first, define the input layers
edge_list_inputs = keras.Input(
    shape=edge_list_ds[0].shape,
    name="adjacency_inputs",
    dtype="int32",
)
node_features_inputs = keras.Input(
    shape=node_features_ds[0].shape,
    name="node_inputs",
)
degrees_inputs = keras.Input(
    shape=degrees_ds[0].shape, name="degrees"
```

```
)

# convolution1 takes as input the node features,
# the edge list, and the degrees
convolution1 = ApplyOverBatch(
    SingleGraphConvolution(
        256,
        activation=ops.relu,
        name="convolution1",
    )
)(
    [
        node_features_inputs,
        edge_list_inputs,
        degrees_inputs,
    ]
)

# convolution2 takes as input the output of
# convolution1, the edge list, and the degrees
convolution2 = ApplyOverBatch(
    SingleGraphConvolution(
        256,
        activation=ops.relu,
        name="convolution2",
    )
)(
    [
        convolution1,
        edge_list_inputs,
        degrees_inputs,
    ]
)

# create a residual connection
residual = keras.layers.Add()(
    [convolution1, convolution2]
)

# reduce the node features to a single graph feature
graph_features = ReduceNodeSum()(residual)

# a few fully connected layers
graph_features = keras.layers.Dense(
    256, activation="relu", name="dense"
)(graph_features)

graph_features = keras.layers.Dense(
    256, activation="relu", name="dense2"
)(graph_features)

# the output layer
outputs = keras.layers.Dense(
    graph_features_ds.shape[1],
    activation="sigmoid",
    name="output",
)(graph_features)

# build the model by specifying the inputs and outputs
model = keras.Model(
    inputs=[
        edge_list_inputs,
        node_features_inputs,
        degrees_inputs,
    ],
    outputs=outputs,
)

# compile the model
model.compile(
    optimizer=optimizers.Adam(learning_rate=1e 4),
    loss="binary_crossentropy",
    metrics=["accuracy"],
)

# Train the model
model.fit(
    [edge_list_ds, node_features_ds, degrees_ds],
    graph_features_ds,
    batch_size=32,
    epochs=10,
)
```

Testing

To ensure the correctness of the implementations, we have generated a bunch of random graphs, and compared the outputs of our layers against the expected results, obtained by manually computing the operations in pure Python + NumPy [45] (A.1).

Keras also provides tools for hyperparameter optimization, such as the **keras_tuner** package. For testing the performance of a model on a dataset, the method **.evaluate()** can be used. For performing statistical model validation tests, traditionally the **sklearn.model_selection** [46] package is used, e.g., **train_test_split**, **cross_val_score**, etc.

Future work

The toolkit could be improved in several ways: adding more GNN architectures in the form of Keras layers, adding support for more dataset formats, etc.

4.2 L-KHC implementation

Introduction

The same package, GraphMindKeras, hosts the implementation of the L-KHC algorithm, which we have developed to be able to build learnable representations of planar graphs, as described in Chapter 3.

In order to develop the extended L-KHC algorithm, we first needed an implementation of the original KHC algorithm [31]. However, to our big surprise, we have found no implementation of the KHC algorithm on the web, regardless of the language, and thus we had to implement it ourselves, based on the paper.

Both the KHC and L-KHC algorithms can be accessed at (A.1). They are implemented in Python 3, with the help of the Sage library [47] for efficient graph operations, as well as an implementation of the SPQR-tree [35].

We have tested the correctness of our implementations on a few datasets of planar molecular graphs from the TUDataset collection (e.g., NCI109, FRANKENSTEIN, etc.). It has correctly labeled each pair of graphs as isomorphic if and only if they were in fact so.

The file **khc.py** contains the implementation of the KHC algorithm. The function **find_planar_code(G)** takes as input a Sage graph object, and returns a list of integers, which represents the code of the graph. For better readability, we also provide the function **code_to_string(code)**, which converts the output of **find_planar_code(G)** to a string, by replacing the ids of the special symbols from the code with their corresponding names. Below, we also provide a simple Python script that demonstrates the invocation of our implementation.

```python
import sage.all as sageall
from khc import find_planar_code, code_to_string

G = sageall.Graph()
G.add_vertices(range(1, 5))
G.add_edges([(1, 2), (1, 3), (2, 3), (3, 4)])

code = find_planar_code(G)

print(code_to_string(code))

# Output:
# (C (A (B (P 2 0 * )P )B (B (S 6 1 * )S )B )A )C
```

The file **l_khc.py**, on the other hand, contains the implementation of the L-KHC algorithm. However, as this algorithm makes sense only when applied on a whole dataset of graphs, rather than a single instance, it provides the function **l_khc(dataset, node_labels, node_attributes, regression)**. The function takes as input the path to a dataset in the TUDataset format, and has two outputs: the list of the codes of the graphs in the dataset (ready to be used as input to a GNN), and the target outputs of the graphs. The argument **node_labels** is a boolean flag indicating whether the dataset contains node labels, and **node_attributes** is a boolean flag indicating whether the dataset contains node attributes. The argument **regression** is a boolean flag indicating whether the task is a regression task or a classification task.

The example below shows how to use the L-KHC algorithm on a dataset of graphs from the TUDataset collection, and later store the results in a file via the **pickle** module.

```python
from khc.l_khc import l_khc
import pickle

dataset_path = "./data/NCI109"

codes, labels = l_khc(dataset_path, True, False, False)


with open("processed_dataset_NCI109.pkl", "wb") as f:
    pickle.dump((codes, labels), f)
```

5 Supporting Experiments

In order to evaluate the performance of the L-KHC algorithm and the Planar GNN we have proposed, we have conducted a series of experiments. We have chosen three different graph datasets, each representing a set of molecules (and thus planar graphs). The datasets were arbitrarily selected from the TUDataset collection [44]. We have focused on datasets that focus on the binary classification of molecules, as it is one of the most common tasks in chemoinformatics. We have preprocessed the datasets with the L-KHC algorithm to generate codes for the graphs. The neural network models were then trained on the codes to predict the properties of the molecules.

The architecture we have used for the Planar GNN is the same across all experiments, but we have used different hyperparameters for each dataset. The best hyperparameters were found by performing a grid search on the validation set. The architecture of the Planar GNN is as follows:

- 2 fully connected layers with ReLU activation function. We have limited ourselves to 2 layers because they are universal approximators [48]. The number of hidden units in each layer is a hyperparameter. The goal of these layers is to generate a rich representation of the input.
- a 1D convolutional layer with ReLU activation function, kernel size 6 and stride 3. The number of filters is a hyperparameter. The goal of this layer is to compress the code in order to speed up the execution of the recurrent layers, which is inherently sequential and thus slow.
- a layer of bidirectional GRU units. The number of units is a hyperparameter. During experimentations, we found that there was no significant difference between using GRU and LSTM units, and thus, we chose GRU units for their simplicity.
- 2 fully connected layers with ReLU activation function, in order to generate the output.
- a layer of 1 hidden unit with sigmoid activation function, for classification.
- a few dropout layers across the network, to prevent overfitting.

In all cases, we have used the Adam optimizer [49], binary cross-entropy loss, and a batch size of 32. As it is usually accepted, we have split the dataset into training and test sets, with a ratio of 80% and 20%, respectively. We have used TensorFlow [42] to implement the models and Tensorboard to monitor the training process.

5.1 NCI109 Dataset

The NCI109 dataset [50] is a dataset of graphs representing organic molecules, provided by the National Cancer Institute (NCI, hence the name). The dataset contains two classes of compounds: active or non-active against human ovarian cancer. Therefore, advances on this dataset may have a direct impact on the development of new anti-cancer drugs. The classes are evenly distributed in the dataset. The dataset contains 4127 graphs, with an average of 29.6 vertices and 32.1 edges per graph. The only additional information provided is the class of each vertex(atom), and there are 38 different classes of vertices in the dataset. We have encoded these node features as one-hot vectors.

After preprocessing the dataset with the L-KHC algorithm, we have obtained a tensor of shape $(4127, 1526, 52)$, where the first dimension represents the number of graphs in the dataset, the second - the length of the largest code (the shorter codes are padded with zeros to match this length), and the third - the size of an element of the code (38 classes + 14 special symbols from the L-KHC algorithm).

The best hyperparameters we have found for the Planar GNN are forming the following network:

1. A fully connected layer with 256 hidden units and ReLU activation function
2. A dropout layer with a dropout rate of 0.5
3. A fully connected layer with 64 hidden units and ReLU activation function
4. A dropout layer with a dropout rate of 0.5
5. A 1D convolutional layer with 256 filters, kernel size 6 and stride 3
6. A bidirectional GRU layer with 256 units
7. A dropout layer with a dropout rate of 0.5
8. A fully connected layer with 512 hidden units and ReLU activation function
9. A dropout layer with a dropout rate of 0.5
10. A fully connected layer with 64 hidden units and ReLU activation function
11. A fully connected, output layer, with 1 hidden unit and sigmoid activation function

The model was trained for 50 epochs, with a learning rate of $8.3e-4$, and exponential decay with a rate of 0.96. On a computer with a Nvidia P100 GPU, the training took around 10 minutes. Figure 5.1 shows the training and test accuracy of the model, while Figure 5.2 shows the training and test loss. The model achieved a test accuracy of $74.92 \pm 1.53\%$ (the higher, the better) when performing a 5-fold cross-validation.

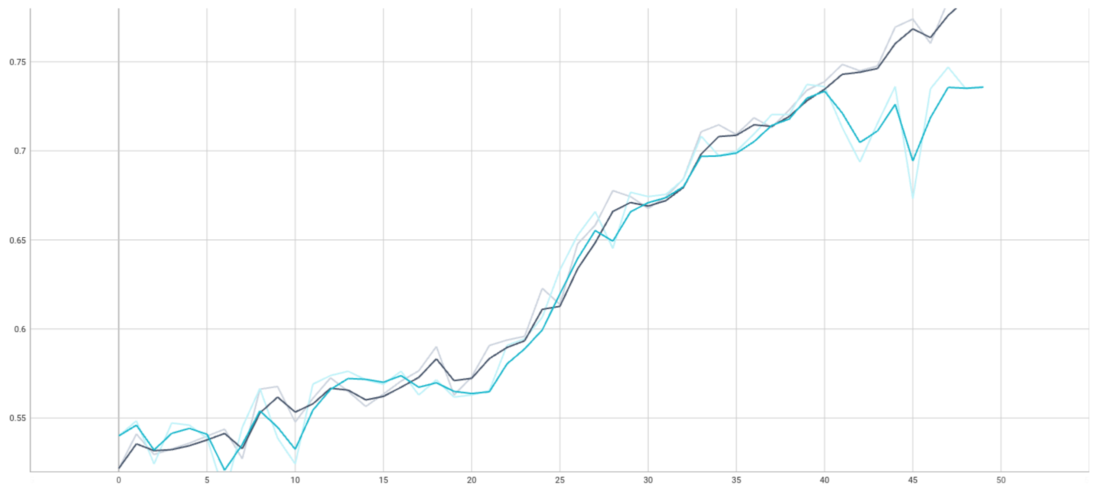


Figure 5.1 Training (black) and test (blue) accuracy of the Planar GNN on the NCI109 dataset

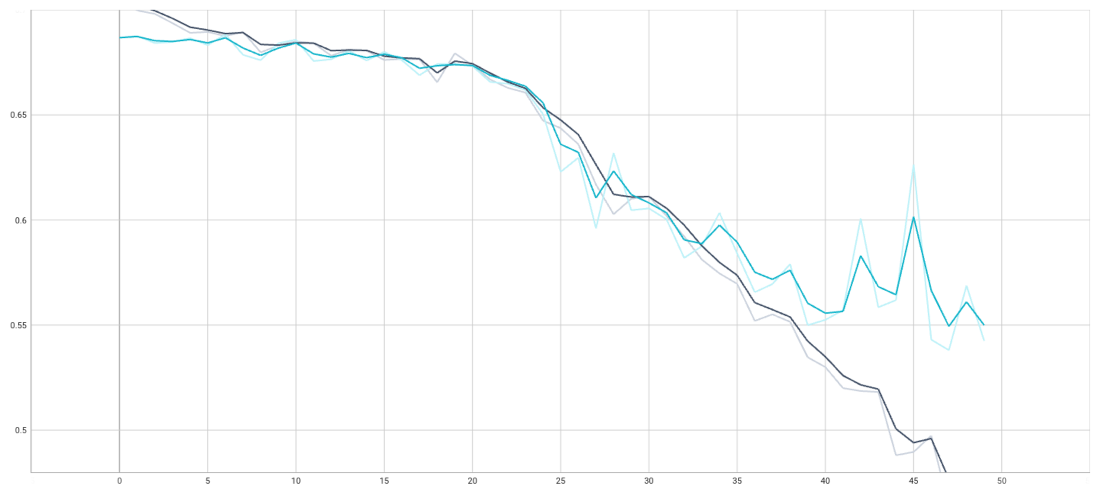


Figure 5.2 Training (black) and test (blue) loss of the Planar GNN on the NCI109 dataset

Comparison to State-of-the-Art

This result is comparable to the state-of-the-art results on the dataset, as can be seen in the table 5.3. Our Planar GNN typically outperforms message-passing neural networks, such as graph convolution variants, e.g., GCN [20], EigenGCN-3 [51], and combinations of graph convolutional networks with graph attention, e.g., SAGPool_g [52] and GraphSage [53]. Nevertheless, it is still outperformed by more complex models, such as WKPI-kC [54].

Method	Accuracy (%)
Planar GNN	74.92 ± 1.53
SAGPool_g [52]	74.06 ± 0.78
SAGPool_h [52]	67.86 ± 1.41
GCN [20], [51]	70.7
GraphSage [53], [51]	70.3
EigenGCN-3 [51]	74.9
WKPI-kC [54]	87.4 ± 0.3

Figure 5.3 Comparison of the Planar GNN to state-of-the-art models on the NCI109 dataset (the higher, the better).

Overfitting

If we let the same model train for longer, i.e., 100 epochs, we can observe some curious behavior. The training accuracy continues to increase, reaching close to 100%, while the test accuracy stagnates at around 75%, but it does not decrease. However, as the loss on the training set continues to decrease, the loss on the test set starts to increase. This behavior can be seen in Figures 5.4 and 5.5. We believe this indicates that we have overachieved our initial goal: to help GNNs better differenciate between graphs. In other words, the missclassified graphs, or more specifically their representation, differs significantly from the graphs in the training set, which leads to the model overfitting on the training set.

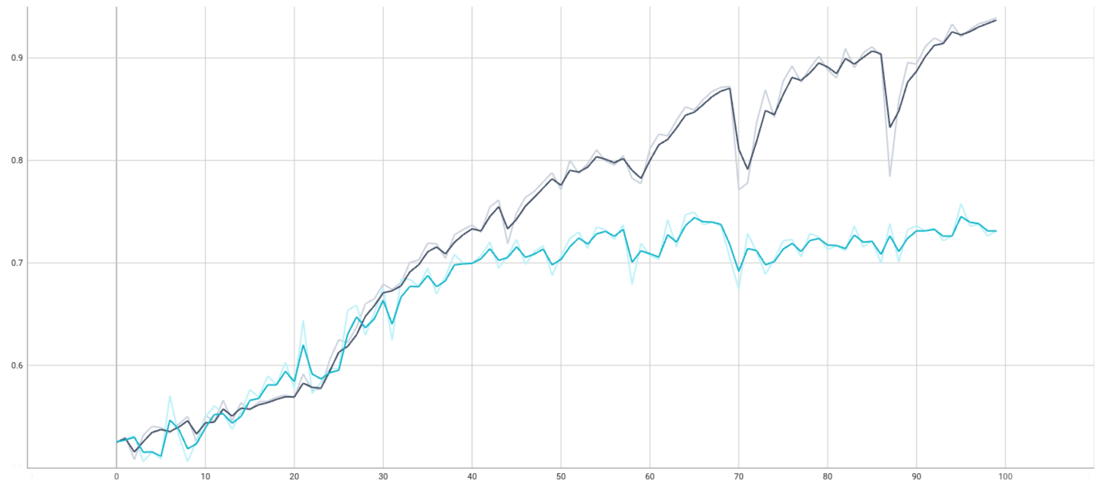


Figure 5.4 Training (black) and test (blue) accuracy of the Planar GNN on the NCI109 dataset, trained for 100 epochs

5.2 ZINC Dataset

The second dataset we have tested our Planar GNN on is the ZINC dataset [55], which is an incredibly popular benchmark for graph neural networks. Similarly to the NCI109 dataset, the ZINC dataset contains graphs representing molecules; however, here, the task is to regress the solubility of the molecules, which is a real scalar. The dataset contains 250k graphs, but we have only used a subset of

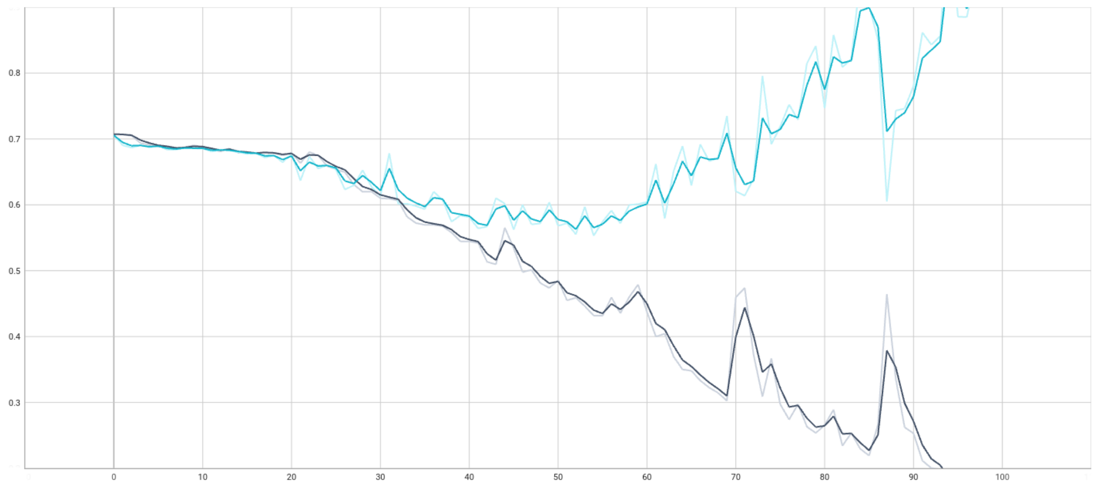


Figure 5.5 Training (black) and test (blue) loss of the Planar GNN on the NCI109 dataset, trained for 100 epochs

12k graphs, as the dataset is too large to process on a single GPU. The average number of vertices in the graphs is 23.15, and the average number of edges is 24.90. The dataset contains 20 different classes of vertices, which we have encoded as one-hot vectors. The dataset also has classes for the edges, but we have not used them in our experiments.

After preprocessing the dataset with the L-KHC algorithm, we have obtained a tensor of shape $(12000, 478, 34)$, where the first dimension represents the number of graphs in the dataset, the second - the length of the largest code, and the third - the size of an element of the code (20 classes + 14 special symbols from the L-KHC algorithm).

The best hyperparameters we have found for the Planar GNN architecture on the ZINC dataset are:

1. A fully connected layer with 256 hidden units and ReLU activation function
2. A dropout layer with a dropout rate of 0.5
3. A fully connected layer with 256 hidden units and ReLU activation function
4. A dropout layer with a dropout rate of 0.5
5. A convolutional layer with 256 filters, kernel size 6 and stride 3, and ReLU activation function
6. A bidirectional GRU layer with 256 units
7. A dropout layer with a dropout rate of 0.5
8. A fully connected layer with 512 hidden units and ReLU activation function
9. A dropout layer with a dropout rate of 0.5
10. A fully connected layer with 256 hidden units and ReLU activation function
11. A fully connected, output layer, with 1 hidden unit and no activation function (regression)

The model was trained for 50 epochs, using a MAE (mean absolute error) loss, with a started learning rate of $1e-3$, with a reduce factor of 0.5 every 20 epochs. The training took around 10 minutes on a Nvidia P100 GPU. After performing a 5-fold cross-validation, the model has achieved a MAE of 0.3355 ± 0.009 (the lower, the better). This is a decent result, outperforming most message-passing neural networks. Table 5.6 provides a detailed comparison of the Planar GNN to state-of-the-art models on the ZINC dataset. The results in the table (other than the Planar GNN) are the result of a study conducted by Vijay et al. [56].

Method	Accuracy (%)
Planar GNN	0.335 ± 0.009
GCN [20]	0.278 ± 0.003
GAT [21]	0.384 ± 0.007
GraphSage [53]	0.398 ± 0.002
GatedGCN [57]	0.435 ± 0.011
GIN [29]	0.526 ± 0.051

Figure 5.6 Comparison of the Planar GNN to state-of-the-art models on the ZINC dataset (the lower, the better).

Discussion

The Planar GNN architecture has achieved promising, competitive results on both the NCI109 and ZINC datasets, outperforming most message-passing neural networks. This proves that the L-KHC algorithm is able to generate codes that are useful for distinguishing between different graphs, and that the Planar GNN is able to effectively process these codes.

Conclusion

Summary

In this thesis, we have studied the field of graph neural networks, with a focus on their application to molecular data. Later on, we presented a novel approach to graph neural networks based on the concept of graph isomorphism testing. Its goals were to:

- Address the limitation of the current state-of-the-art graph neural networks in the form of message-passing networks, which is the inability to distinguish certain graph structures.
- Present a model capable of overcoming this limitation by producing a unique representation of a graph based on the KHC algorithm for planar graph isomorphism testing.
- Produce a graph representation that can be fed into a traditional deep learning model, thus benefiting from the advancements in the field of deep learning while still being able to process graph data.
- Construct a bridge between the fields of graph neural networks and modern deep learning techniques.
- Develop a state-of-the-art graph neural network architecture capable of outperforming existing models on the task of graph classification and regression.

We have achieved this by modifying the KHC algorithm for planar graph isomorphism testing and introducing the Planar GNN architecture, which is based on the former. The Planar GNN architecture has shown promising results on the tasks of graph classification and regression, outperforming most existing message-passing graph neural network architectures on the evaluated datasets. Breakthroughs in the field of graph neural networks, and consequently chemoinformatics and bioinformatics, since graph-structured data is frequent in these domains, could have a significant impact on the healthcare, materials industry, and other fields. We hope the Planar GNN architecture will be a step in this direction.

Moreover, we have developed a memory-efficient and portable toolkit for building and training traditional graph neural networks.

Further Research

The Planar GNN architecture is a novel approach to graph neural networks, and as such, there are many directions in which it can be further developed, for example:

- Extension of the L-KHC algorithm to be able to process edge features.
- Exploration of other structures of the network capable of learning sequential data, such as transformers.

- Extension of the architecture to a generative model capable of generating graphs. This can be achieved by training the model to generate a L-KHC representation of a graph based on some desired properties and then reconstructing the molecule from this representation.
- Exploration of the architecture in other areas where planar graphs are common, such as computer vision.

Bibliography

1. Wu, Yongji; Lian, Defu; Xu, Yiheng; Wu, Le; Chen, Enhong. Graph Convolutional Networks with Markov Random Field Reasoning for Social Spammer Detection. *Proceedings of the AAAI Conference on Artificial Intelligence*. 2020, vol. 34, no. 01, pp. 1054–1061. Available from DOI: 10.1609/aaai.v34i01.5455.
2. Fout, Alex; Byrd, Jonathon; Shariat, Basir; Ben-Hur, Asa. Protein Interface Prediction using Graph Convolutional Networks. In: Guyon, I.; Luxburg, U. Von; Bengio, S.; Wallach, H.; Fergus, R.; Vishwanathan, S.; Garnett, R. (eds.). *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017, vol. 30. Available also from: https://proceedings.neurips.cc/paper_files/paper/2017/file/f507783927f2ec2737ba40afbd17efb5-Paper.pdf.
3. Khalil, Elias; Dai, Hanjun; Zhang, Yuyu; Dilkina, Bistra; Song, Le. Learning Combinatorial Optimization Algorithms over Graphs. In: Guyon, I.; Luxburg, U. Von; Bengio, S.; Wallach, H.; Fergus, R.; Vishwanathan, S.; Garnett, R. (eds.). *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017, vol. 30. Available also from: https://proceedings.neurips.cc/paper_files/paper/2017/file/d9896106ca98d3d05b8cbdf4fd8b13a1-Paper.pdf.
4. Datta, Samir; Limaye, Nutan; Nimbhorkar, Prajakta; Thierauf, Thomas; Wagner, Fabian. A Log-space Algorithm for Canonization of Planar Graphs. *CoRR*. 2008, vol. abs/0809.2319. Available from arXiv: 0809.2319.
5. Rücker, Christoph; Meringer, Markus. How many organic compounds are graph-theoretically nonplanar? *Match-communications in Mathematical and in Computer Chemistry*. 2002, vol. 45. Available also from: https://api.semanticscholar.org/CorpusID:115878158.
6. Kuratowski, Casimir. Sur le problème des courbes gauches en Topologie. *Fundamenta Mathematicae*. 1930, vol. 15, no. 1, pp. 271–283. Available also from: http://eudml.org/doc/212352.
7. Rosenblatt, Frank. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*. 1958, vol. 65 6, pp. 386–408. Available also from: https://api.semanticscholar.org/CorpusID:12781225.
8. Minsky, M.; Papert, S. *Perceptrons*. Cambridge, MA: MIT Press, 1969.
9. Bishop, Christopher M.; Bishop, Hugh. Deep Neural Networks. In: *Deep Learning: Foundations and Concepts*. Cham: Springer International Publishing, 2024, pp. 171–207. ISBN 978-3-031-45468-4. Available from DOI: 10.1007/978-3-031-45468-4_6.
10. Bishop, Christopher M.; Bishop, Hugh. Gradient Descent. In: *Deep Learning: Foundations and Concepts*. Cham: Springer International Publishing, 2024, pp. 209–232. ISBN 978-3-031-45468-4. Available from DOI: 10.1007/978-3-031-45468-4_7.

11. Bishop, Christopher M.; Bishop, Hugh. Convolutional Networks. In: *Deep Learning: Foundations and Concepts*. Cham: Springer International Publishing, 2024, pp. 287–324. isbn 978-3-031-45468-4. Available from doi: 10.1007/978-3-031-45468-4_10.

12. Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

13. Aggarwal, Charu. Recurrent Neural Networks. In: *Neural Networks and Deep Learning: A Textbook*. Cham: Springer International Publishing, 2023, pp. 265–304. isbn 978-3-031-29642-0. Available from doi: 10.1007/978-3-031-29642-0_8.

14. Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

15. Hochreiter, Sepp; Schmidhuber, Jürgen. Long Short-Term Memory. *Neural Comput.* 1997, vol. 9, no. 8, pp. 1735–1780. issn 0899-7667. Available from doi: 10.1162/neco.1997.9.8.1735.

16. Cho, Kyunghyun; Merrienboer, Bart van; Gulcehre, Caglar; Bahdanau, Dzmitry; Bougares, Fethi; Schwenk, Holger; Bengio, Yoshua. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. 2014. Available from arXiv: 1406.1078 [cs.CL].

17. Scarselli, Franco; Gori, Marco; Tsoi, Ah Chung; Hagenbuchner, Markus; Monfardini, Gabriele. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*. 2009, vol. 20, no. 1, pp. 61–80. Available from doi: 10.1109/TNN.2008.2005605.

18. Gilmer, Justin; Schoenholz, Samuel S.; Riley, Patrick F.; Vinyals, Oriol; Dahl, George E. Neural Message Passing for Quantum Chemistry. *CoRR*. 2017, vol. abs/1704.01212. Available from arXiv: 1704.01212.

19. Hamilton, William L. Graph Representation Learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*. [N.d.], vol. 14, no. 3, pp. 1–159.

20. Kipf, Thomas N.; Welling, Max. *Semi-Supervised Classification with Graph Convolutional Networks*. 2017. Available from arXiv: 1609.02907 [cs.LG].

21. Veličković, Petar; Cucurull, Guillem; Casanova, Arantxa; Romero, Adriana; Liò, Pietro; Bengio, Yoshua. *Graph Attention Networks*. 2018. Available from arXiv: 1710.10903 [stat.ML].

22. Bahdanau, Dzmitry; Cho, Kyunghyun; Bengio, Yoshua. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. Available from arXiv: 1409.0473 [cs.CL].

23. Vaswani, Ashish; Shazeer, Noam; Parmar, Niki; Uszkoreit, Jakob; Jones, Llion; Gomez, Aidan N.; Kaiser, Lukasz; Polosukhin, Illia. *Attention Is All You Need*. 2023. Available from arXiv: 1706.03762 [cs.CL].

24. Kingma, Diederik P; Welling, Max. *Auto-Encoding Variational Bayes*. 2022. Available from arXiv: 1312.6114 [stat.ML].

25. KIPF, Thomas N.; WELLING, Max. *Variational Graph Auto-Encoders.* 2016. Available from arXiv: `1611.07308 [stat.ML]`.

26. GROHE, Martin; NEUEN, Daniel. *Recent Advances on the Graph Isomorphism Problem.* 2021. Available from arXiv: `2011.01366 [cs.DS]`.

27. WEISFEILER, Boris; LEMAN, Andrey. *The reduction of a graph to canonical form and the algebra which appears therein.* Nauchno-Technicheskaya Informatsia, vol. 2(9):12-16, 1968. Available also from: `https://www.iti.zcu.cz/wl2018/pdf/wl_paper_translation.pdf`.

28. MORRIS, Christopher; RITZERT, Martin; FEY, Matthias; HAMILTON, William L.; LENSSEN, Jan Eric; RATTAN, Gaurav; GROHE, Martin. *Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks.* 2021. Available from arXiv: `1810.02244 [cs.LG]`.

29. XU, Keyulu; HU, Weihua; LESKOVEC, Jure; JEGELKA, Stefanie. *How Powerful are Graph Neural Networks?* 2019. Available from arXiv: `1810.00826 [cs.LG]`.

30. HOPCROFT, J. E.; TARJAN, R. E. Isomorphism of Planar Graphs (Working Paper). In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department.* Ed. by MILLER, Raymond E.; THATCHER, James W.; BOHLINGER, Jean D. Boston, MA: Springer US, 1972, pp. 131–152. ISBN 978-1-4684-2001-2. Available from DOI: `10.1007/978-1-4684-2001-2_13`.

31. KUKLUK, Jacek P. *Algorithm and experiments in testing planar graphs for isomorphism.* Vol. 8. The University of Texas at Arlington, 2003. No. 2. Available also from: `https://ailab.wsu.edu/subdue/papers/KuklukJGAA05.pdf`.

32. DIMITROV, Radoslav; ZHAO, Zeyang; ABBOUD, Ralph; CEYLAN, İsmail İlkan. *PlanE: Representation Learning over Planar Graphs.* 2023. Available from arXiv: `2307.01180 [cs.LG]`.

33. HARARY, F.; PRINS, G. Publicationes Mathematicae Debrecen. In: 1966, chap. The block-cutpoint-tree of a graph., pp. 103–107.

34. DI BATTISTA, G.; TAMASSIA, R. Incremental planarity testing. In: *30th Annual Symposium on Foundations of Computer Science.* 1989, pp. 436–441. Available from DOI: `10.1109/SFCS.1989.63515`.

35. GUTWENGER, Carsten; MUTZEL, Petra. A Linear Time Implementation of SPQR-Trees. In: *International Symposium Graph Drawing and Network Visualization.* 2000. Available also from: `https://api.semanticscholar.org/CorpusID:14338454`.

36. WEINBERG, L. A Simple and Efficient Algorithm for Determining Isomorphism of Planar Triply Connected Graphs. *IEEE Transactions on Circuit Theory.* 1966, vol. 13, no. 2, pp. 142–148. Available from DOI: `10.1109/TCT.1966.1082573`.

37. WHITNEY, Hassler. Congruent Graphs and the Connectivity of Graphs. *American Journal of Mathematics* [online]. 1932, vol. 54, no. 1, pp. 150–168 [visited on 2024-06-08]. ISSN 00029327, ISSN 10806377. Available from: `http://www.jstor.org/stable/2371086`.

38. KŐNIG, Dénes. Theory of finite and infinite graphs. In: [online]. Boston : Birkhäuser, 1990, chap. Euler trails and hamiltonian cycles, pp. 90–91 [visited on 2024-06-08]. Available from: `https://archive.org/details/theoryoffinitein0000koni/page/422/mode/2up`.

39. VAN ROSSUM, Guido; DRAKE, Fred L. *Python 3 Reference Manual.* Scotts Valley, CA: CreateSpace, 2009. ISBN 1441412697.

40. CHOLLET, Francois et al. *Keras.* GitHub, 2015. Available also from: `https://github.com/fchollet/keras`.

41. PASZKE, Adam; GROSS, Sam; MASSA, Francisco; LERER, Adam; BRADBURY, James; CHANAN, Gregory; KILLEEN, Trevor; LIN, Zeming; GIMELSHEIN, Natalia; ANTIGA, Luca; DESMAISON, Alban; KÖPF, Andreas; YANG, Edward; DEVITO, Zach; RAISON, Martin; TEJANI, Alykhan; CHILAMKURTHY, Sasank; STEINER, Benoit; FANG, Lu; BAI, Junjie; CHINTALA, Soumith. *PyTorch: An Imperative Style, High-Performance Deep Learning Library.* 2019. Available from arXiv: `1912.01703 [cs.LG]`.

42. MARTÍN ABADI; ASHISH AGARWAL; PAUL BARHAM; EUGENE BREVDO; ZHIFENG CHEN; CRAIG CITRO; GREG S. CORRADO; ANDY DAVIS; JEFFREY DEAN; MATTHIEU DEVIN; SANJAY GHEMAWAT; IAN GOODFELLOW; ANDREW HARP; GEOFFREY IRVING; MICHAEL ISARD; JIA, Yangqing; RAFAL JOZEFOWICZ; LUKASZ KAISER; MANJUNATH KUDLUR; JOSH LEVENBERG; DANDELION MANÉ; RAJAT MONGA; SHERRY MOORE; DEREK MURRAY; CHRIS OLAH; MIKE SCHUSTER; JONATHON SHLENS; BENOIT STEINER; ILYA SUTSKEVER; KUNAL TALWAR; PAUL TUCKER; VINCENT VANHOUCKE; VIJAY VASUDEVAN; FERNANDA VIÉGAS; ORIOL VINYALS; PETE WARDEN; MARTIN WATTENBERG; MARTIN WICKE; YUAN YU; XIAOQIANG ZHENG. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* 2015. Available also from: `https://www.tensorflow.org/`. Software available from tensorflow.org.

43. TRAXL, Dominik; BOERS, Niklas; KURTHS, Jürgen. Deep Graphs - A general framework to represent and analyze heterogeneous complex systems across scales. *Chaos.* 2016, vol. 26, no. 6. Available from DOI: `http://dx.doi.org/10.1063/1.4952963`.

44. MORRIS, Christopher; KRIEGE, Nils M.; BAUSE, Franka; KERSTING, Kristian; MUTZEL, Petra; NEUMANN, Marion. TUDataset: A collection of benchmark datasets for learning with graphs. In: *ICML 2020 Workshop on Graph Representation Learning and Beyond (GRL+ 2020).* 2020. Available from arXiv: `2007.08663`.

45. HARRIS, Charles R.; MILLMAN, K. Jarrod; WALT, Stéfan J. van der; GOMMERS, Ralf; VIRTANEN, Pauli; COURNAPEAU, David; WIESER, Eric; TAYLOR, Julian; BERG, Sebastian; SMITH, Nathaniel J.; KERN, Robert; PICUS, Matti; HOYER, Stephan; KERKWIJK, Marten H. van; BRETT, Matthew; HALDANE, Allan; RÍO, Jaime Fernández del; WIEBE, Mark; PETERSON,

Pearu; Gérard-Marchant, Pierre; Sheppard, Kevin; Reddy, Tyler; Weckesser, Warren; Abbasi, Hameer; Gohlke, Christoph; Oliphant, Travis E. Array programming with NumPy. *Nature*. 2020, vol. 585, no. 7825, pp. 357–362. Available from DOI: 10.1038/s41586-020-2649-2.

46. Buitinck, Lars; Louppe, Gilles; Blondel, Mathieu; Pedregosa, Fabian; Mueller, Andreas; Grisel, Olivier; Niculae, Vlad; Prettenhofer, Peter; Gramfort, Alexandre; Grobler, Jaques; Layton, Robert; VanderPlas, Jake; Joly, Arnaud; Holt, Brian; Varoquaux, Gaël. API design for machine learning software: experiences from the scikit-learn project. In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122.

47. The Sage Developers. *SageMath, the Sage Mathematics Software System (Version x.y.z)*. YYYY. https://www.sagemath.org.

48. Hornik, Kurt; Stinchcombe, Maxwell; White, Halbert. Multilayer feedforward networks are universal approximators. *Neural Networks*. 1989, vol. 2, no. 5, pp. 359–366. ISSN 0893-6080. Available from DOI: https://doi.org/10.1016/0893-6080(89)90020-8.

49. Kingma, Diederik P.; Ba, Jimmy. *Adam: A Method for Stochastic Optimization*. 2017. Available from arXiv: 1412.6980 [cs.LG].

50. Wale, Nikil; Watson, Ian; Karypis, George. Comparison of Descriptor Spaces for Chemical Compound Retrieval and Classification. *Knowl. Inf. Syst.* 2008, vol. 14, pp. 347–375. Available from DOI: 10.1109/ICDM.2006.39.

51. Ma, Yao; Wang, Suhang; Aggarwal, Charu C.; Tang, Jiliang. *Graph Convolutional Networks with EigenPooling*. 2019. Available from arXiv: 1904.13107 [cs.LG].

52. Lee, Junhyun; Lee, Inyeop; Kang, Jaewoo. *Self-Attention Graph Pooling*. 2019. Available from arXiv: 1904.08082 [cs.LG].

53. Hamilton, William L.; Ying, Rex; Leskovec, Jure. *Inductive Representation Learning on Large Graphs*. 2018. Available from arXiv: 1706.02216 [cs.SI].

54. Zhao, Qi; Wang, Yusu. *Learning metrics for persistence-based summaries and applications for graph classification*. 2019. Available from arXiv: 1904.12189 [cs.CG].

55. Sterling, Teague; Irwin, John J. ZINC 15 – Ligand Discovery for Everyone. *Journal of Chemical Information and Modeling*. 2015, vol. 55, no. 11, pp. 2324–2337. Available from DOI: 10.1021/acs.jcim.5b00559. PMID: 26479676.

56. Dwivedi, Vijay Prakash; Joshi, Chaitanya K.; Luu, Anh Tuan; Laurent, Thomas; Bengio, Yoshua; Bresson, Xavier. Benchmarking Graph Neural Networks. *Journal of Machine Learning Research*. 2023, vol. 24, no. 43, pp. 1–48. Available also from: http://jmlr.org/papers/v24/22-0567.html.

57. Bresson, Xavier; Laurent, Thomas. *Residual Gated Graph ConvNets*. 2018. Available from arXiv: 1711.07553 [cs.LG].

List of Figures

List of Abbreviations

- **GNN** - Graph Neural Network
- **MLP** - Multi-Layer Perceptron
- **CNN** - Convolutional Neural Network
- **RNN** - Recurrent Neural Network
- **GCN** - Graph Convolutional Network
- **GAT** - Graph Attention Network
- **GAE** - Graph Autoencoder
- **VGAE** - Variational Graph Autoencoder
- **WL** - Weisfeiler-Lehman algorithm
- **KHC** - An algorithm for planar graph isomorphism testing presented by Kukluk, Holder, and Cook [31]
- **C-KHC** - A modification of the KHC algorithm to handle disconnected graphs
- **L-KHC** - A modification of the KHC algorithm to produce a learnable graph representation

A Attachments

A.1 Source Code

- Source code of the package “GraphMindKeras”:
  `https://gitlab.mff.cuni.cz/lupascoa/graph-neural-network-model`
- Examples of usage of the package:
  `https://gitlab.mff.cuni.cz/lupascoa/graph-neural-network-model/-/tree/master/model/examples?ref_type=heads`
- Tests of the package:
  `https://gitlab.mff.cuni.cz/lupascoa/graph-neural-network-model/-/tree/master/model/tests?ref_type=heads`
- Source code of the KHC and L-KHC algorithms:
  `https://gitlab.mff.cuni.cz/lupascoa/graph-neural-network-model/-/tree/master/khc?ref_type=heads`

A.2 Datasets

- Homepage of the TUDataset website:
  `https://chrsmrrs.github.io/datasets/docs/datasets/`