UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 3131

GRAPH CLASSIFICATION USING MACHINE LEARNING TECHNIQUES

Ana Čačić

Zagreb, June 2023

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 10 March 2023

MASTER THESIS ASSIGNMENT No. 3131

Student:	**Ana Čačić (0036507969)**
Study:	Computing
Profile:	Computer Science
Mentor:	asst. prof. Lana Horvat Dmitrović

Title: **Graph classification using machine learning techniques**

Description:

Over the past decade, there has been significant growth in the field of machine learning that focuses on graph-structured data. This growth has led to the development of more advanced methods for analyzing and solving problems related to graph data, such as the classification of nodes, edges, and entire graphs. The objective of this thesis is to describe the problem of graph classification and explore machine learning methods that solve it. As part of the work, software solutions for some researched methods should be implemented and applied to a chosen data set. Finally, the obtained results should be evaluated and compared.

Submission date: 23 June 2023

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 10. ožujka 2023.

DIPLOMSKI ZADATAK br. 3131

Pristupnica:	**Ana Čačić (0036507969)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentorica:	doc. dr. sc. Lana Horvat Dmitrović

Zadatak: **Klasifikacija grafova korištenjem metoda strojnog učenja**

Opis zadatka:

Tijekom prošlog desetljeća došlo je do značajnog rasta u području strojnog učenja koje je usredotočeno na podatke strukturirane grafovima. Taj rast je doveo do razvoja naprednijih metoda za analizu i rješavanje problema vezanih uz podatke strukturirane grafovima, kao što su klasifikacija čvorova, bridova i grafova. Cilj ovog diplomskog rada je opisati problem klasifikacije grafova te istražiti metode strojnog učenja koje ga rješavaju. U sklopu rada treba implementirati programska rješenja za neke od istraženih metoda te ih primijeniti na odabrani skup podataka. Na kraju se dobiveni rezultati trebaju evaluirati i usporediti.

Rok za predaju rada: 23. lipnja 2023.

Completing this journey would not have been possible without all the wonderful people in my life. From the depths of my heart, I extend my gratitude to:

My mentor, Lana, for her professional guidance, immense patience, genuine understanding, and infinite kindness.

My family, particularly my mother Jasna, sister Marija, and grandmother Ana, for their endless love and constant encouragement.

My friends, who always believed in me and with whom I've shared countless moments of joy.

And to the love of my life, my fiancé Roberto, for his unconditional love, for always standing by my side, and for all the wonderful moments we've shared over the past seven years.

CONTENTS

1. Introduction

1.1. Motivation

Graphs have become an essential tool for representing relationships or interactions between objects, thereby impacting multiple research domains, ranging from social to natural sciences. Recent decades have witnessed significant growth in both the quality and quantity of graph-structured datasets, which has led to advancements in the fields of traffic prediction [6], drug discovery [19], recommendation systems [23], and many more. Inspired by these developments and intrigued by this domain, this thesis aims to explore machine learning on graphs and make a small contribution to this field.

1.2. Related Work

1.2.1. A Review of Existing Datasets

For any meaningful research, regardless of the domain, it is essential to have access to adequate data.

One of the earliest graph dataset collections is the Stanford Network Analysis Project[1] (**SNAP**). Established in 2004 and actively developed since, SNAP includes hundreds of datasets spanning over twenty categories, like social networks, communication networks, web graphs, and many more [15].

In 2020, two large graph dataset collections were introduced. The first one is the **TU-Dataset**[2], a collection spanning 120 datasets that can be used for graph classification and regression [16].

The second one is the Open Graph Benchmark[3] (**OGB**), a set of real-life challeng-

[1]`http://snap.stanford.edu/`

[2]`https://chrsmrrs.github.io/datasets/`

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

ing graph datasets. These datasets are large-scale and encompass multiple important machine-learning tasks for graphs. OGB provides an automated end-to-end graph machine-learning pipeline that simplifies and standardizes the process of graph data loading and model evaluation [9].

1.2.2. Analysis of Existing Methods

Graph classification is one aspect of analyzing graph data. In the pre-neural era, researchers analyzed statistical and structural graph properties. The main focus was on finding unique features that would differentiate graphs.

With the evolution of the field, new methods that combined graph data with traditional machine learning algorithms emerged. Amongst these methods, embedding-based techniques stood out.

Further advancements in machine learning shifted the focus from its traditional methods to methods from the realm of deep learning - neural networks.

Nowadays, the go-to methods for graph classification or similar tasks are graph neural networks (GNNs). Due to their expressiveness and adaptability, they perform better than traditional methods. This claim is supported by the fact that GNNs are in the top best-performing models on benchmark datasets, like the previously mentioned OGB[4].

A more detailed description of the above mentioned methods can be found in Chapter 2.

1.3. Problem Definition

Road networks are an example of real-world graph models that are an essential part of human life. They play a crucial role in everyday commutes, as well as in long-distance travel. However, the existing dataset collections are sparse regarding traffic networks and lack a dataset focused on graph classification.

In the interest of contributing to this field, the specific goals of this thesis are:

1. Creating a road network dataset for graph classification.
2. Experimenting on the new dataset with different GNNs.
3. Comparing the obtained results.

[4]`https://ogb.stanford.edu/docs/leader_graphprop/`

1.4. Thesis Structure

The second chapter introduces the basic terminology needed to understand this thesis and gives a brief historical overview of the evolution of machine learning on graphs. The third chapter provides insight into the dataset's creation and analysis. The fourth chapter dives into the theoretical overview of selected models and offers implementational details. The fifth chapter presents the results of this thesis, while the sixth chapter comments on these results and suggests possible further improvements. The seventh chapter offers concluding remarks. The last few pages contain references used in this thesis and its abstract.

2. Terminology

2.1. What is a Graph?

A **graph** is a discrete data structure that is the focus of a mathematical branch called **graph theory**. This data structure models connections between pairs of objects.

The field of graph theory dates its origin to the 18th century, and mathematicians have thoroughly studied it. However, there is no need for an exhaustive glossary overview. Instead, only a small subset of terms has to be defined to fully understand the vocabulary used in the rest of the thesis.

2.1.1. Basic Definitions

A **simple graph** $\mathcal{G}$ consists of a non-empty finite set $\mathcal{V}$, whose elements are called **vertices**, and a finite set $\mathcal{E}$, whose elements are called **edges**. Set $\mathcal{E}$ is a 2-element subset of $\mathcal{V}$ that has distinct vertices. A graph $\mathcal{G}$ can be denoted as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ [5].

Figure 2.1 shows an example of a simple graph.

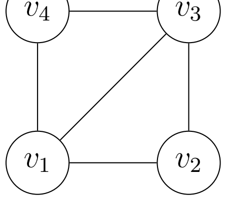

Figure 2.1: Example of a simple graph

[5]In scientific literature, the terms *graph*, *vertex*, and *edge* are frequently used interchangeably with the terms *network*, *node*, and *link*, respectively. The former is typically used to describe a mathematical representation, while the latter often refers to real-life systems. The distinction between these two terminologies is subtle, and consequently, these terminologies are often used as synonyms for one another [1].

Two simple graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ are **isomorphic** if there exists a bijective correspondence (1-1 mapping) between sets $\mathcal{V}_1$ and $\mathcal{V}_2$ such that the number of edges connecting any two chosen vertices in $\mathcal{V}_1$ equals the number of edges connecting the corresponding two vertices in $\mathcal{V}_2$.

Figure 2.2 depicts an example of two simple graphs that are isomorphic.

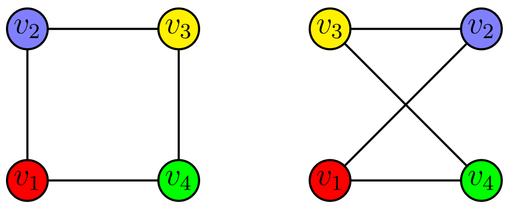

Figure 2.2: Example of two isomorphic graphs

A simple graph does not permit the existence of an edge $e = \{u, u\}$, that is, a **loop** or **self-loop**.

If a graph has multiple edges between two vertices, then it is called a **multigraph**.

Figure 2.3 gives an example of a multigraph that has a self-loop.[6].

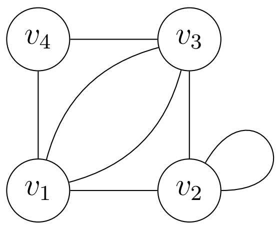

Figure 2.3: Example of multigraph with a loop

If there exists an edge $e = \{u, v\}$, i.e., edge e **connects** vertices u and v, then u and v are **neighbors**. Vertices u and v are **incident** to edge e. All neighbors of u form a **neighborhood**, which can be denoted as $\mathcal{N}(u)$.

Figure 2.4 shows an example of a simple graph and highlights one node's neighborhood.

[6]It is not necessary for a multigraph to have a self-loop but it can.

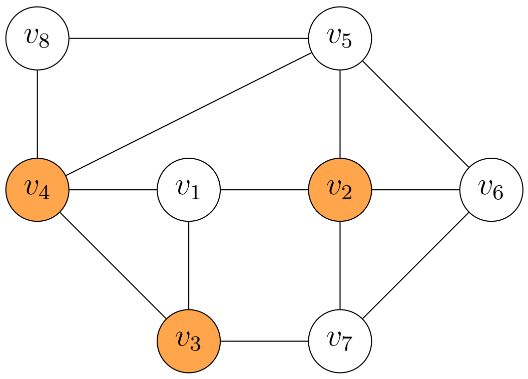

Figure 2.4: v_1's neighborhood $\mathcal{N}(v_1)$

If an edge e can be denoted as either $e = \{u, v\}$ or $e = \{v, u\}$ then this edge is **undirected**. If the order of the vertices matters, i.e., the edge has a direction, then it is **directed**.

Figure 2.5 gives an example of a directed graph.

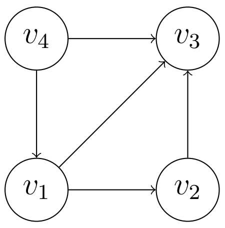

Figure 2.5: Example in a directed graph

The **degree** of a vertex is the number of incident edges. In a directed graph, the number of incoming edges is the **in-degree**, and the number of outgoing edges is the **out-degree**.

Edges can have assigned *weights*, in which case the graph is **weighted**. On the contrary, a graph with no weights is **unweighted**.

Figure 2.6 shows an example of a weighted graph.

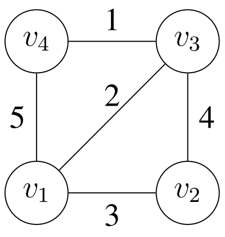

Figure 2.6: Example of a weighted graph

There are a few ways to represent a graph, and one is by using an **adjacency matrix**. An adjacency matrix $\mathbf{A}$ is a square matrix that indicates whether or not two vertices are connected. Formally, $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$.

In the case of a simple graph, the adjacency matrix is a $(0, 1)$-matrix. The number 0 indicated the absence of an edge. Any number higher than 0 (e.g., number 1) indicates the presence of an edge. If a graph is undirected, its adjacency matrix is symmetric. A directed graph does not necessarily have a symmetric adjacency matrix. A weighted graph can have numbers other than 1 to indicate the presence and weight of an edge.

A **walk** is a sequence of edges connecting a series of vertices, which can be either finite or infinite. A **trail** is a type of walk where every edge is unique. A **path** is a trail where each vertex is encountered only once.

2.1.2. Graph Traversal

Graph **traversal** is the process of visiting every vertex in a graph. Two commonly used graph traversal algorithms are **BFS** and **DFS**.

Breadth-first search (BFS) explores all adjacent vertices before moving on to the next set of vertices. The algorithm starts at some initial vertex and checks its immediate neighbors, then the neighbors of the neighbors, and so forth.

Depth-first search (DFS) explores the graph as deep as possible before backtracking. Backtracking occurs when there are no more unvisited adjacent vertices. This process repeats until all vertices are visited.

Figure 2.7 visualizes BFS and DFS.

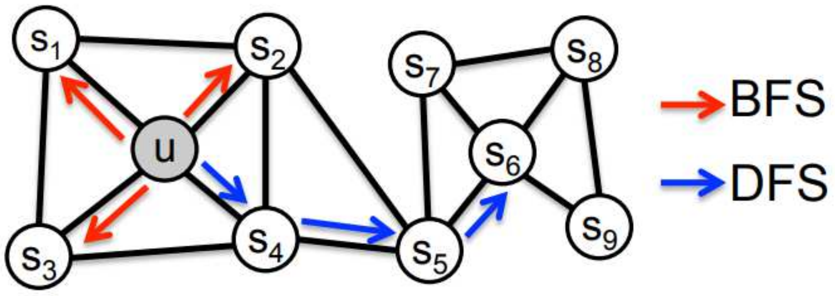

Figure 2.7: A visual representation of BFS and DFS [7]

2.2. Machine Learning Fundamentals

This section aims to explain the fundamental concepts of machine learning relevant to this thesis.

2.2.1. Machine Learning

First, it is necessary to define the term **machine learning** (ML). Machine learning is a subcategory of *artificial intelligence* (AI). It refers to a set of algorithms and statistical models that computers use for solving complex tasks without being explicitly programmed for those specific tasks. Models *learn* to solve a task by identifying patterns in the input data and optimizing a performance criterion.

Input data for training a machine learning model comes in various forms, such as images, text, audio, etc. This data is converted into **features** - measurable properties or characteristics of the provided data. The goal is to represent that data in a structured way (e.g., as vectors) suitable for the machine learning algorithm.

The main types of tasks machine learning aims to solve fall into two categories: **classification** and **regression**. Classification predicts a discrete label while regression predicts a continuous value.

The classification task aims to predict the correct **label** (or **target**) for the given input data. Classification tasks are divided into either **binary** or **multiclass** classification. Binary classification distinguishes between two classes, typically denoted as the *positive* class and *negative* class, while multiclass classification involves multiple classes (categories).

The process that encompasses all the steps from gathering input data to obtaining a trained model is called a **machine-learning pipeline**. Although numerous variations of a machine-learning pipeline exist, each of these variations will contain the following steps:

1. **Data collection** - gathering raw data.
2. **Data preparation** - converting raw data into a structured format used as input to the machine learning algorithm.
3. **Model training** - consists of iterative processes that adjust the machine learning algorithm's internal parameters to minimize the error and best adapt to the underlying patterns in the input data.

4. **Model evaluation** - assesses the overall performance of the trained algorithm by using a relevant evaluation metric.

The first part of the pipeline deals with collecting, cleaning, and preparing data. The second part includes model selection, training a model, and evaluating its performance.

Choosing the appropriate model depends on multiple factors, such as the type of problem, the data working with, and the complexity of the problem.

Sometimes, problems can be too complex for traditional machine-learning models. In that case, it is necessary to approach the task with *deeper* models.

2.2.2. Deep Learning

A subcategory of machine learning is **deep learning**. Compared to traditional machine learning, deep learning uses **neural networks** with multiple *layers*.

Neural networks are computational models inspired by how biological brains function. They consist of interconnected **neurons**.

Neurons are the basic computational units of a neural network, simulating the functions of biological neurons. Each connection between two neurons has an associated **weight** that determines how important one neuron's output is to the other neuron's *activation*.

Every neuron has an **activation function** that determines the output of that neuron. This function adds non-linearity to the model, which gives the neural network the capacity to learn complex patterns.

The following are some of the commonly used activation functions and their formulas. Figures 2.8 to 2.12 visualize these functions:

Sigmoid:

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

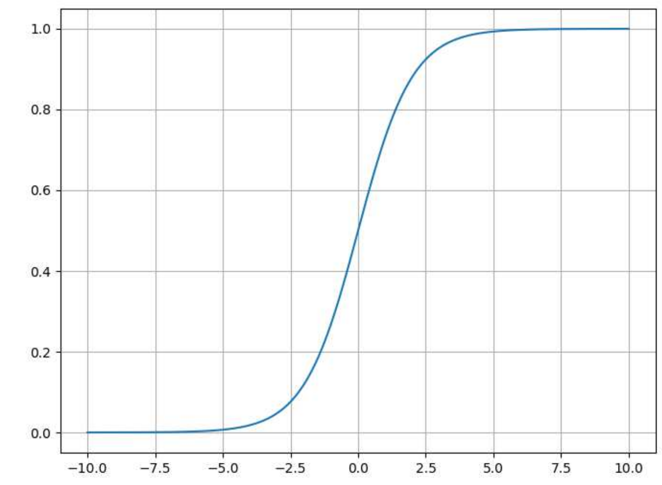

Figure 2.8: Sigmoid function

Tanh:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

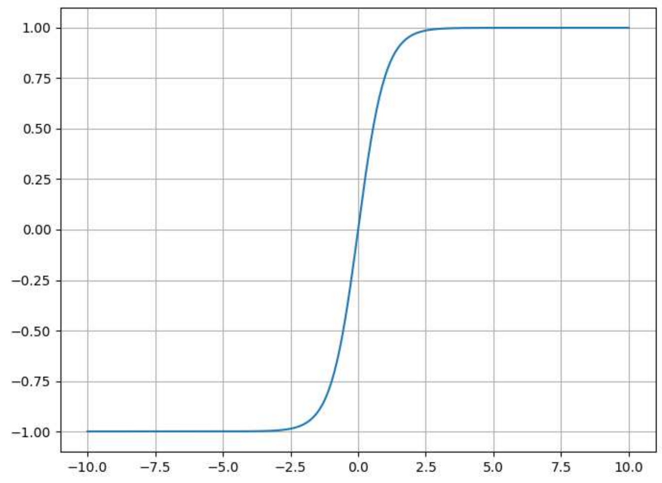

Figure 2.9: Tanh function

Rectified Linear Unit (ReLU):

$$\text{ReLU}(x) = \max(0, x)$$

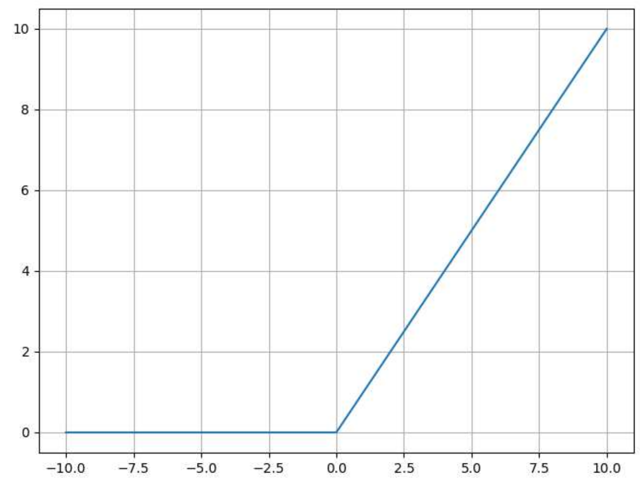

Figure 2.10: ReLU function

Leaky Rectified Linear Unit (Leaky ReLU):

$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{if} x > 0 \\ \alpha x & \text{otherwise} \end{cases}$$

where α is a small constant.

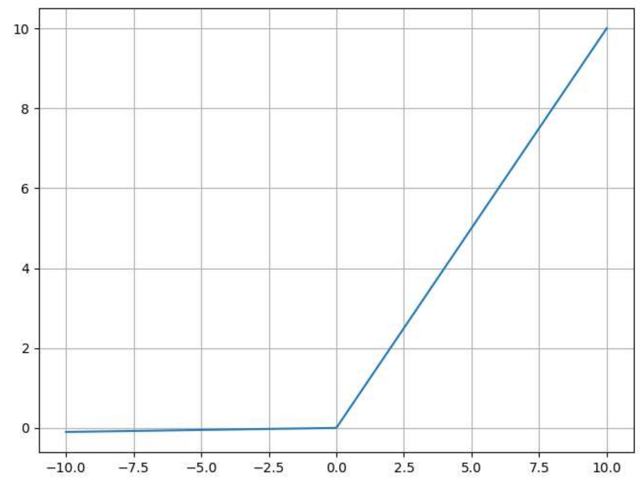

Figure 2.11: Leaky ReLU function

Exponential Linear Unit (ELU):

$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}$$

where α is a positive constant.

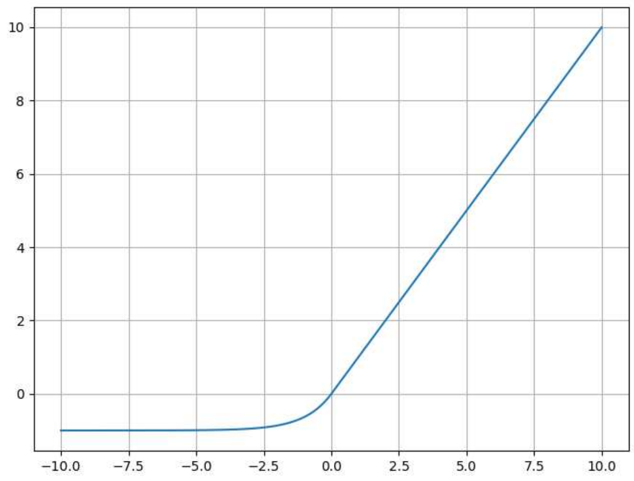

Figure 2.12: ELU function

Alongside weights, neurons typically have **biases** associated with them. This bias shifts a neuron's output, which can benefit the network's performance.

One of the simplest examples of a neural network is the **multilayer perceptron** (MLP). Its key components are the input, hidden, and output layers. Each of these layers consists of a certain number of neurons. Data is processed sequentially from the input layer, through hidden layers, and finally to the output layer.

Figure 2.13 shows a simple multilayer perceptron.

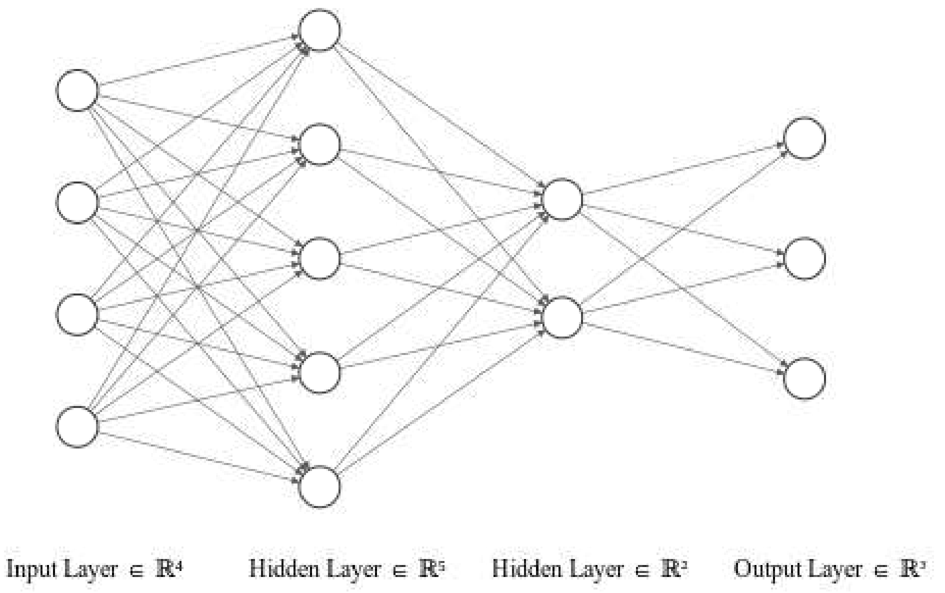

Figure 2.13: Example of a multilayer perceptron

The MLP is a special type of **Feedforward Neural Network** (FFNN). FFNN is a broader term used to describe any neural network where the data flows in only one direction: from input to output. MLP strictly needs to have more than one hidden layer.

The learning process of a simple neural network consists of two main parts: **forward pass** and a **backward pass**.

The forward pass consists of:

1. **Input layer** - receives the input data and forwards it to the next layer.
2. **Propagation through hidden layers**:
   - 2.1. **Weighted sum** - calculates the weighted sum for each neuron by multiplying the incoming values from the previous layer with the current weights of the connections leading to that layer.
   - 2.2. **Activation function** - introduces non-linearity into the network by taking the weighted sum as input to any non-linear function (e.g., sigmoid, tanh, ReLU, etc.) and producing the output for each neuron.
3. **Output layer** - produces the prediction of the network.

The backward pass or **backpropagation** corrects the network's weights to achieve better performance. It consists of the following steps:

1. **Loss calculation** - calculates how much the network's prediction differs from the true output by using a loss function. A commonly used loss function is the **cross-entropy loss** whose formula for multiclass classification is the following:

   $$L(\mathbf{y}, \mathbf{p}) = -\sum_{i=1}^{C} y_i \log(p_i)$$

   where vector $\mathbf{y}$ is a one-hot encoded vector in which y_i is 1 if the class label i is the correct class, and 0 otherwise. Vector $\mathbf{p}$ is the probability vector in which p_i is the predicted probability of a sample belonging to the class i.

2. **Output layer gradient** - computes the gradient of the loss for each neuron in the output layer.

3. **Propagation of gradients through hidden layers**:

   3.1. **Weight gradient**- calculate the gradient of the loss concerning the weights.

   3.2. **Previous output gradient** - calculate the gradient of the loss concerning the outputs of the previous layer.

4. **Weight update** - after computing all gradients, weights (and biases) are updated using an optimization algorithm, like Adam [12].

The forward pass and backpropagation are repeated for multiple epochs until the performance starts to converge or until the model starts to lose its generalization ability (*overfitting*).

Convolutional Neural Networks

A more complicated neural network than the FFNN is the **Convolutional Neural Network** (CNN). CNNs are specifically designed to work with grid-like data, like images, and have been responsible for many breakthroughs in the field of computer vision.

Typically, a CNN consists of the following layers:

1. **Convolutional Layer** - the core of every CNN. It uses *filters* (also known as *kernels*) that slide over input data (usually images) to produce *feature maps*. The objective is to learn spatial features from the input.

2. **Pooling Layer** - this layer typically follows the convolutional layer. It reduces the dimensions of feature maps, preserving important information. Two types of pooling operators are common:

  - **Max pooling** - takes the maximum value from a group of values in the feature map.
  - **Average pooling** - takes the average of a group of values in the feature map.

3. **Fully Connected Layer**(FC) - after several convolutional layers followed by pooling layers, the fully connected layer produces a raw output of the neural network. These raw outputs are often followed by a **softmax** activation that transforms them into a vector of probabilities, producing the classification prediction. The FC layer is sometimes called a *dense* or *linear* layer.

Figure 2.14 shows an example of a CNN.

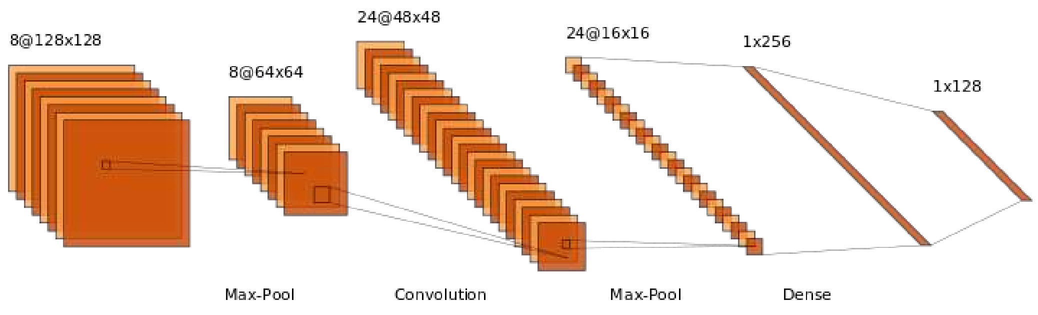

Figure 2.14: Example of a CNN architecture

2.3. Towards Machine Learning on Graphs

Machine learning algorithms can be applied to various graph data in order to solve a variety of tasks. These tasks can be categorized by multiple criteria.

When classified by the components being analyzed, there are these types of tasks:

1. **Node-level** - tasks focused on individual nodes, such as node classification.
2. **Link-level** - tasks concerning links, like link prediction.
3. **Graph-level** - tasks that target the entire graph, for example, graph classification.

In terms of the learning paradigm, tasks can be divided into:

1. **Transductive learning** - training and prediction occur on the same data (typically nodes)

2. **Inductive learning** - a subset of data is used during training, and predictions are made on unseen data

In relation to the amount of labeled data, tasks can be categorized into:

1. **Supervised learning** - all training data is labeled.
2. **Semi-supervised learning** - a subset of training data is labeled, and both labeled and unlabeled data are used during training.
3. **Unsupervised learning** - none of the training data is labeled.

Methods used for solving these types of tasks have rapidly evolved during the last few decades.

This section aims to further explain the terminology used in the rest of this thesis. It also provides a brief, but not complete, overview of how machine learning on graphs has evolved from traditional machine learning methods to neural networks.

2.3.1. Feature Engineering

Solving graph-related tasks in the earlier stages relied on experts finding the appropriate *features* to use as input to a standard machine-learning pipeline. In this context, a **feature** or an **attribute** is any additional information associated with nodes, edges, or graphs.

Such features were designed by calculating various statistical and topological graph properties (e.g., node degree, and path lengths). These hand-engineered features captured the underlying structure and relationships within graphs and allowed traditional machine-learning methods to make more sophisticated predictions.

Figure 2.15 shows the traditional machine learning on graphs workflow.

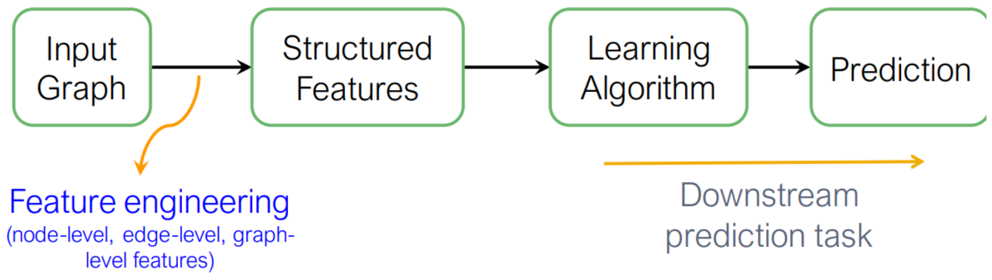

Figure 2.15: Traditional machine learning for graphs [14]

2.3.2. Node Embeddings

Since handcrafting features is time-consuming, the natural question would be whether this part of the machine-learning pipeline can be automated. This question yields a positive answer, and the focus shifts from manual feature engineering to **representation learning**. Representation learning is a process in which models recognize patterns in input data and automatically learn features or representations suitable for a specific task. In the context of graph-based representation learning, one common approach is generating **node embeddings**.

Figure 2.16 depicts how graph representation learning replaces the manual feature selection.

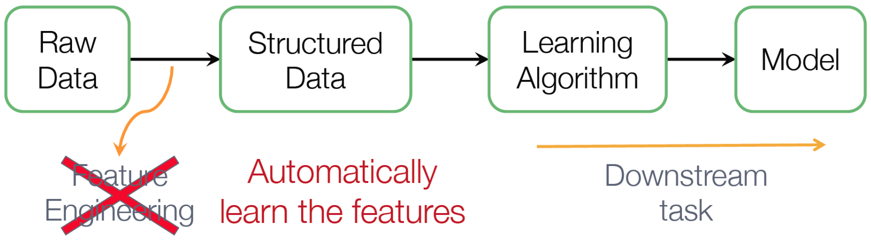

Figure 2.16: Graph representation learning [14]

Node embeddings are node representations in a continuous vector space. The goal is to perform a mapping from a typically high-dimensional, non-Euclidean input space (graph) to a low-dimensional, Euclidean latent space. This mapping has to preserve graph topology and relationships of nodes. As a result, the distance between vectors in the latent space should reflect the relative distance of nodes in the original graph.

Figure 2.17 visualizes the node embedding process.

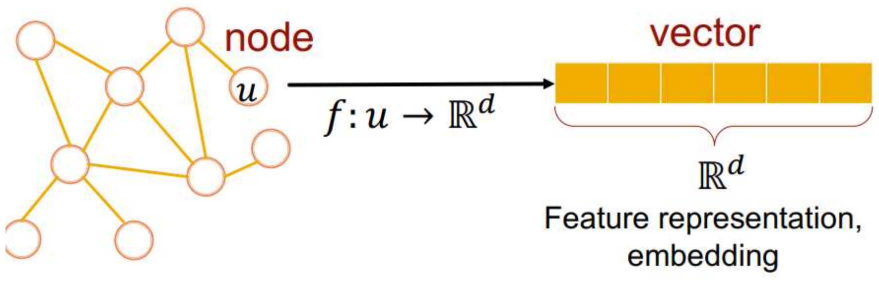

Figure 2.17: Vector representation of a node [14]

Encoder-Decoder Framework

The encoder-decoder framework structure is often used in machine learning tasks, especially for sequence-to-sequence (seq2seq) problems, like machine translation.

The **encoder** takes an input sequence, processes it in an adequate way, and outputs a fixed-sized vector.

The **decoder** takes the fixed-sized vector from the encoder and uses it to produce an output sequence (e.g., a translated sentence in a machine translation task).

In the context of graphs, the encoder (ENC) maps nodes to low-dimensional embeddings. The goal is to encode nodes u and v in a way that their similarity[7] is approximate to the dot product of their embedding vectors:

$$\text{similarity}(u, v) \approx \mathbf{z}_u^\top \cdot \mathbf{z}_v$$

The decoder (DEC) uses these embeddings and tries to make predictions about the original input graph. One such example would be trying to predict the neighbors of a node u, given its embedding $\mathbf{z}_u$.

Figure 2.18 depicts the encoding process.

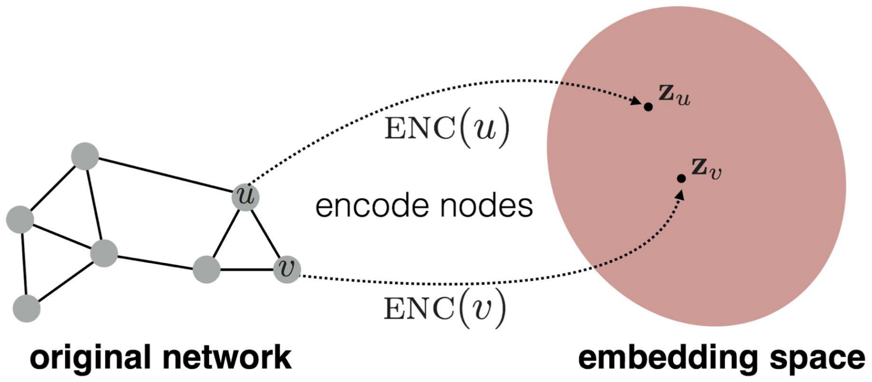

Figure 2.18: Encoder mapping nodes to low-dimensional space [14]

Random Walk Methods

Two well-known node embedding methods based on random walks are **DeepWalk** [17] and **node2vec** [7]. While both approaches employ random walks, node2vec introduces a biased random walk that interpolates between a BFS-like and a DFS-like walk based on hyperparameters.

[7]A brief overview of node similarity measures can be found in [18]

2.3.3. Graph Neural Networks

Node embedding methods like DeepWalk and node2vec are an important step in automating feature design. However, these methods produce *shallow embeddings*. Shallow embeddings have limitations that prevent a deeper insight into the graph's structure. Some of these limitations include:

- Absence of an iterative update mechanism for node embeddings.
- Inability to incorporate node features.
- Inherent transductive nature.

These challenges can be overcome by using more advanced models. These models lie in the realm of **deep learning**, and they are known as **graph neural networks** (GNNs).

Preliminaries

Nodes often contain additional information that, for every node $u \in \mathcal{V}$ in a graph $\mathcal{G}$, is represented as a **feature vector**, $\mathbf{x}_u \in \mathbb{R}^d$. Stacking the feature vectors of all nodes together results in a **node feature matrix**, $\mathbf{X} \in \mathbb{R}^{|\mathcal{V}| \times d}$. This can be also written as $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{|\mathcal{V}|}]^\top$.

The goal is to combine this feature matrix with the adjacency matrix and be able to generate node representations that, in fact, rely on the graph's structure and feature information. However, the adjacency matrix enforces a node ordering, which is problematic for the representation learning process since nodes (and edges) do not have an order.

It is crucial to ensure that changing the order of nodes and edges does not change the outputs. To respect this property, GNNs must satisfy either **permutation invariance** or **permutation equivariance**.

The invariance property ensures that regardless of how shuffled or rearranged the nodes of a graph are, the output remains the same.

The equivariance property assures that if the order of the input nodes changes, the order of the output nodes changes accordingly. Even if the specific representation of each node changes its position due to permutation, it will still accurately represent that node's characteristics and role in the graph.

The following equations illustrate both of these demands, respectively:

$$f(\mathbf{PX}, \mathbf{PAP}^\top) = f(\mathbf{X}, \mathbf{A}) \tag{2.1}$$

$$f(\mathbf{PX}, \mathbf{PAP}^{\top}) = \mathbf{P}f(\mathbf{X}, \mathbf{A}) \tag{2.2}$$

where $\mathbf{P}$ represents any permutation matrix.

Neural Message Passing

The defining feature of GNNs is their employment of the **neural message passing** mechanism. In this context, the term **message** typically refers to aggregated structural information or features from each node's neighbor. This information is then used to update the node's own features. Neural message passing is the process where nodes exchange messages and update them. During each message passing iteration, for each node $u \in \mathcal{V}$, a **hidden embedding** $\mathbf{h}_u^{(k)}$ is generated and updated according to aggregated information from u's neighborhood $\mathcal{N}_u$.

The message-passing framework can be formalized using the following formula presented in [8]:

$$\mathbf{h}_u^{(k+1)} = \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right) \tag{2.3}$$

where UPDATE and AGGREGATE are arbitrary differentiable functions (i.e., neural networks), and the superscript denotes the iteration.

The message $\mathbf{m}_{\mathcal{N}(u)}$ that aggregates neighborhood information can be defined as:

$$\mathbf{m}_{\mathcal{N}(u)} = \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\}) \tag{2.4}$$

This reduces the formula to:

$$\mathbf{h}_u^{(k+1)} = \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}\right) \tag{2.5}$$

After K iterations of passing messages, the final output is u's embedding:

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}, \forall u \in \mathcal{V} \tag{2.6}$$

Figure 2.19 visualizes the message-passing framework.

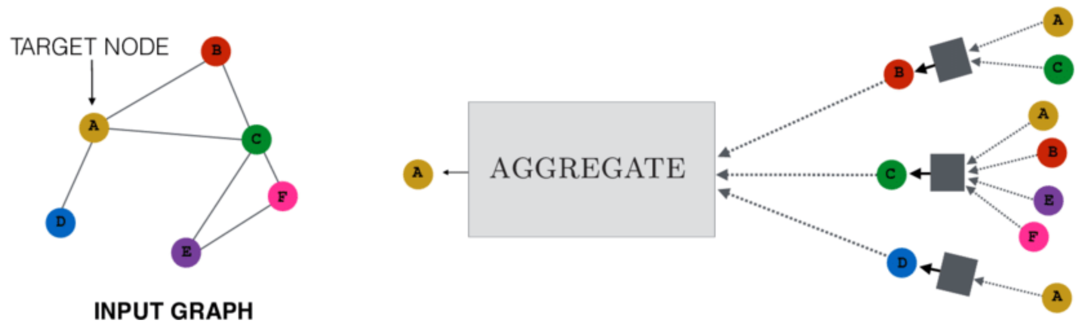

Figure 2.19: Visualization of the message passing framework [13]

After obtaining the node embeddings, it is possible to produce a single graph-level representation needed for tasks like graph classification. This can be achieved by using a READOUT function. The READOUT function aggregates information from all node embeddings and is formulated as:

$$\mathbf{z}_G = \text{READOUT}(\mathbf{h}_u^{(K)}, \forall u \in \mathcal{V}) \tag{2.7}$$

where $\mathbf{z}_G$ represents the graph's embedding.

This neural message-passing procedure is very similar to the convolution operation in CNNs. Namely, the goal of the convolution operation is to collect local features from a structured grid. Neural message passing also aims to aggregate local features but from an unstructured graph.

Basic GNN Layer

By turning the abstract terms UPDATE and AGGREGATE into practical functions, the result is the basic GNN framework:

$$\mathbf{h}_u^{(k+1)} = \phi\left(\mathbf{W}_{\text{self}}^{(k+1)}\mathbf{h}_u^{(k)} + \mathbf{W}_{\text{neigh}}^{(k+1)} \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k)} + \mathbf{b}^{(k+1)}\right) \tag{2.8}$$

where ϕ denotes any elementwise non-linear function (e.g., ReLU), $\mathbf{W}_{\text{self}}^{(k)}, \mathbf{W}_{\text{neigh}}^{(k)} \in \mathbb{R}^{d^{(k)} \times d^{(k-1)}}$ are trainable parameter matrices, and $\mathbf{b}^{(k)} \in \mathbb{R}^{d^{(k)}}$ is the bias vector.

The READOUT function can either be a straightforward operation, such as a summation or averaging of node embeddings or leverage more complex techniques, like the hierarchical pooling presented in [24].

GNN Categories

According to [4], most GNNs fall into one of these categories:

1. **Convolutional**
2. **Attentional**
3. **Message-passing**

Figure 2.20 visually showcases these categories.

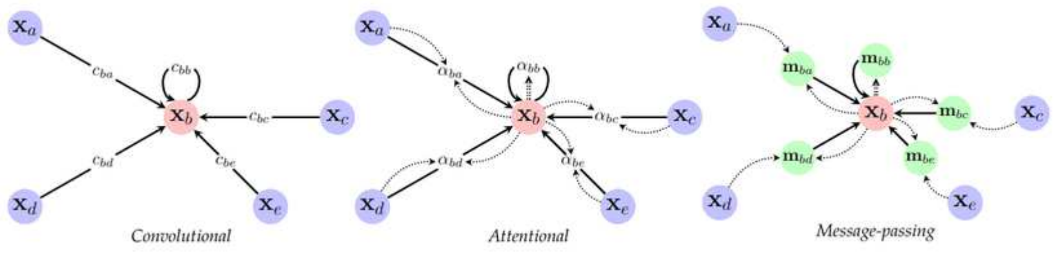

Figure 2.20: Types of GNN layers presented in [4]

The following equations represent each of these types of GNNs, respectively:

$$\mathbf{h}_u = \phi\Big(\mathbf{x}_u, \bigoplus_{v \in \mathcal{N}(u)} c_{vu}\psi(\mathbf{x}_v)\Big) \tag{2.9}$$

$$\mathbf{h}_u = \phi\Big(\mathbf{x}_u, \bigoplus_{v \in \mathcal{N}(u)} a(\mathbf{x}_u, \mathbf{x}_v)\psi(\mathbf{x}_v)\Big) \tag{2.10}$$

$$\mathbf{h}_u = \phi\Big(\mathbf{x}_u, \bigoplus_{v \in \mathcal{N}(u)} \psi(\mathbf{x}_u, \mathbf{x}_v)\Big) \tag{2.11}$$

ϕ and ψ are usually affine transformations with learnable parameters and activation functions, and $\bigoplus$ is any aggregator invariant to permutation (e.g., maximum or average).

In the convolutional layer, the transformed nodes are multiplied with fixed coefficients c_{vu}. Similarly, the attentional layer also has coefficients, but they are learnable attention scores represented by $a(\mathbf{x}_u, \mathbf{x}_v)$. The message-passing layer utilizes a learnable message function that enables nodes to exchange aggregated information.

An important observation is that the expressive power of these approaches can be ordered as *convolution* $\subseteq$ *attention* $\subseteq$ *message-passing*.

3. Dataset

3.1. Dataset Description

One of the goals of this thesis was to create a dataset that contains road networks of various European cities. Some countries do not use the word *city* to describe their urban areas. Those countries who do use the term *city* do not necessarily define it the same way as other countries. For the sake of simplicity, the word *city* will be used interchangeably with *urban area*.

The emphasis was on more developed urban areas with the criterion of requiring at least 50,000 inhabitants. However, due to a lack of data and human error, not all cities meeting this criterion are in the dataset. An additional requirement was for countries to have at least eight cities meeting the first criterion. This requirement ensures that each class has at least one representative after various dataset splits.

The assembled dataset contains 1,715 graphs (cities) belonging to one out of 30 countries. Road networks are represented as directed multigraphs. The edges represent roads suited for cars. Nodes represent either junctions or dead-ends, which is a simplified version of the original graph. More about how these graphs are simplified can be found in [2].

The nodes and edges of these graphs contain additional information about the cities they represent. Table 3.1 outlines these attributes.

Feature	Attributed to	Description
degree	nodes	Number of incoming and outgoing edges of a node.
length	edges	Length of a road segment, measured in meters.
highway	edges	Type of road.

Table 3.1: Node and edge features

Table 3.2 lists and describes road types present in the dataset. This table is an exhaustive overview of all road types appearing in the dataset, but not all possible road types.

Road Type	Description	Priority
`motorway`	Major highways intended for fast motor traffic.	1
`trunk`	Important roads that aren't motorways.	2
`primary`	Roads linking large cities.	3
`secondary`	Roads linking medium-sized cities.	4
`tertiary`	Roads linking smaller cities and villages.	5
`residential`	Roads accessing or within a residential area.	6
`motorway_link`	Roads linking to/from a motorway.	7
`trunk_link`	Roads linking to/from a trunk road.	8
`primary_link`	Roads linking to/from a primary road.	9
`secondary_link`	Link roads for secondary roads.	10
`tertiary_link`	Link roads for tertiary roads.	11
`living_street`	Roads in residential areas with pedestrian priority.	12
`escape`	Lanes for emergency stopping.	13
`road`	Generic road (when the exact type isn't known).	14
`busway`	Roads designated for buses only.	15
`bus_stop`	Designated place where buses pick up or drop off passengers.	16
`crossing`	Points where pedestrians can cross.	17
`emergency_bay`	Safe stopping place alongside fast roads.	18
`mini_roundabout`	Small roundabouts suitable for all vehicles.	19
`passing_place`	Small sections of road for vehicles to pass each other.	20
`rest_area`	Places beside the road for drivers to rest.	21

Table 3.2: OpenStreetMap Road Types

Road types not present in Table 3.2 fall into the category `unclassified`, and these edges have the value 0 assigned to their `highway` attribute.

Figures 3.1 and 3.2 are a visual representation of two cities (graphs) from the dataset.

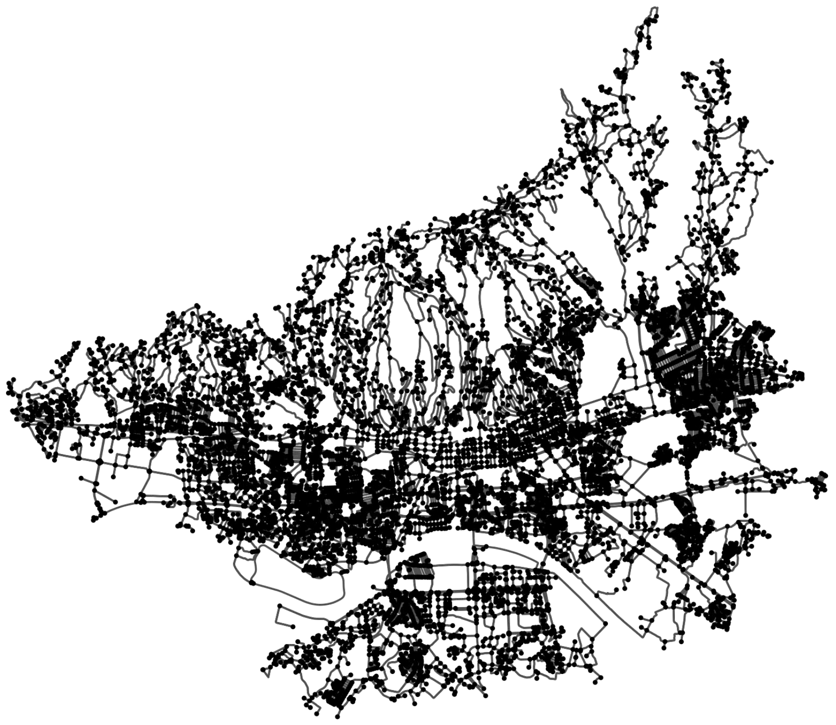

Figure 3.1: Zagreb, Croatia

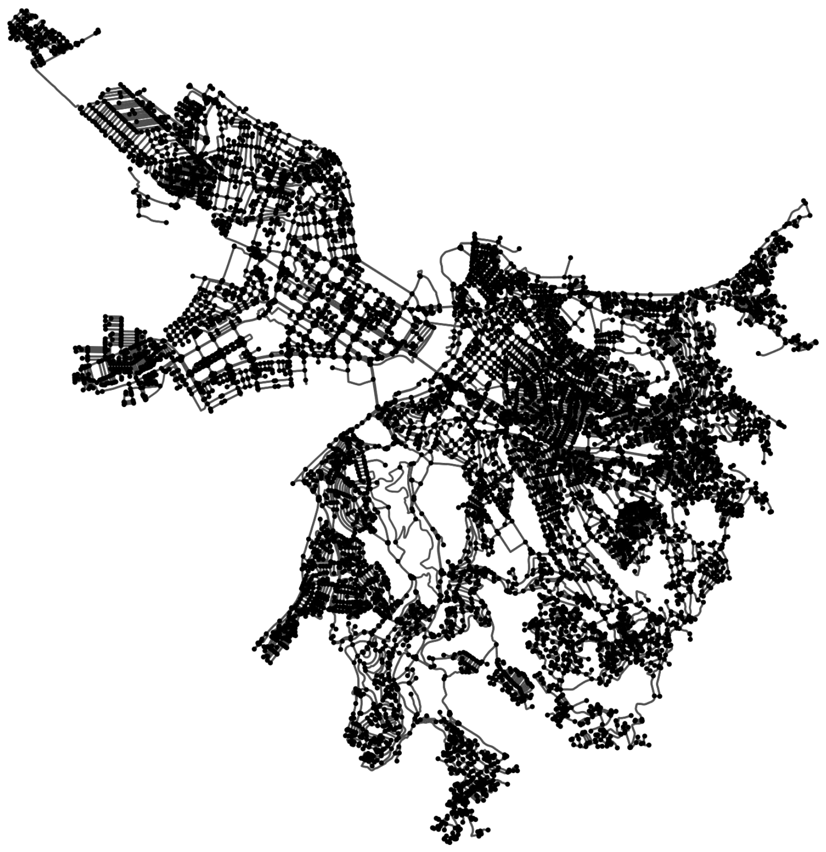

Figure 3.2: Belgrade, Serbia

3.2. Dataset Generation

Creating this road network dataset consisted of two major parts:

1. Obtaining data from OpenStreetMap[8] (OSM).
2. Transforming this data into a format compatible with PyTorch Geometric[9] (PyG).

3.2.1. OSM to GraphML

OpenStreetMap is a community-based map service that creates a free, editable world map. OSM was created as an alternative to proprietary mapping services. Volunteers around the globe use surveys, GPS data, aerial images, and other open sources to gather and maintain data. To ensure efficient data download, OSM offers Overpass API[10], a read-only API that returns custom-selected parts of the map.

The Overpass API is crucial for OSMnx[11], a Python package built on top of NetworkX[12]. OSMnx allows researchers and practitioners to easily download, construct, analyze, and visualize complex street networks from anywhere in the world [2].

OSMnx can save the downloaded data to disk in three different formats: GraphML, SVG, and shapefiles. GraphML is used in this thesis.

3.2.2. GraphML to PyG

GraphML[13] is an XML-based file format that describes the structural properties of a graph. The collected data contains attributes related to the graph itself, its nodes, and its edges.

After selecting the convenient node and edge attributes, each graph was represented as a `torch_geometric.data.Data` object. These transformed graphs are then stored in their respective folders (country name) as `.pt`[14] files. All this data is later loaded into a custom `Dataset` class and used throughout the training and validation phases.

[8] `https://www.openstreetmap.org/`
[9] `https://pyg.org/`
[10] `https://wiki.openstreetmap.org/wiki/Overpass_API`
[11] `https://osmnx.readthedocs.io/en/stable/`
[12] `https://networkx.org/`
[13] `http://graphml.graphdrawing.org/`
[14] `https://fileinfo.com/extension/pt`

3.3. Dataset Analysis

Table 3.3 displays basic information about the dataset and its two subsets.

country	#graphs			#nodes			#edges		
	all	train	test	all	train	test	all	train	test
Albania	8	7	1	19 967	12 576	7391	45 692	29 764	15 928
Austria	9	8	1	31 874	29 365	2509	74 084	68 074	6010
Belarus	24	22	2	32 360	29 664	2696	83 605	75 922	7683
Belgium	27	24	3	27 364	25 180	2184	57 911	53 356	4555
BiH	20	18	2	22 396	20 435	1961	50 453	45 988	4465
Bulgaria	21	19	2	38 060	35 040	3020	94 409	86 288	8121
Croatia	9	8	1	21 751	20 438	1313	49 650	46 765	2885
Czechia	18	16	2	41 567	38 253	3314	95 578	87 386	8192
Denmark	29	26	3	51 859	47 920	3939	116 842	107 898	8944
Finland	21	19	2	58 225	55 554	2671	129 905	123 512	6393
France	114	103	11	228 911	206 970	21 941	492 756	445 886	46 870
Germany	185	166	19	372 127	346 908	25 219	890 538	829 932	60 606
Greece	43	39	4	92 298	85 193	7105	215 011	197 921	17 090
Hungary	20	18	2	53 507	51 578	1929	137 582	132 857	4725
Italy	144	129	15	278 709	258 196	20 513	586 735	541 560	45 175
Kosovo	13	12	1	12 559	12 063	496	27 757	26 674	1083
Macedonia	10	9	1	17 690	15 854	1836	41 973	37 666	4307
Netherlands	59	53	6	182 552	172 123	10 429	434 194	409 039	25 155
Norway	20	18	2	43 685	41 391	2294	98 370	92 908	5462
Poland	92	83	9	192 580	171 412	21 168	439 126	391 839	47 287
Portugal	56	49	7	53 095	48 225	4870	110 163	99 853	10 310
Romania	46	42	4	78 121	69 186	8935	184 888	164 072	20 816
Russia	177	159	18	240 189	188 407	51 782	603 625	486 515	117 110
Serbia	36	32	4	49 442	46 122	3320	118 624	110 597	8027
Slovakia	11	10	1	15 839	14 341	1498	34 878	31 461	3417
Spain	144	130	14	281 250	255 279	25 971	553 666	500 960	52 706
Sweden	45	41	4	92 323	87 215	5108	211 746	200 091	11 655
Switzerland	10	9	1	13 975	11 912	2063	32 627	27 321	5306
UK	221	199	22	970 365	883 760	86 605	2 160 587	1 968 157	192 430
Ukraine	83	75	8	151 567	131 575	19 992	409 638	358 068	51 570

Table 3.3: Basic dataset information

The primary dataset is shuffled and divided into two subsets (90% to 10% ratio). Employing a stratified split ensures that each class is proportionally represented in both subsets. The larger subset serves the purpose of model training and evaluation. Meanwhile, the smaller subset is supposed to mimic unseen data (i.e., test data) to assess the accuracy of the models' evaluation.

3.4. Imbalanced Dataset

This assembled dataset is, as expected, highly unbalanced. Some countries (classes) have significantly more cities (samples) than others. The reason for this imbalance is the fact that some countries are more developed, have a larger land area, and have a larger population size.

Typically, the imbalanced dataset problem could be addressed by **oversampling** or **undersampling** the dataset. Oversampling refers to duplicating samples belonging to the minority class. On the other hand, undersampling implies removing data from the majority classes.

For most machine learning tasks, new incoming data is anticipated (e.g., text, images, numbers, etc.). Such data is, in most cases, easy to generate and frequent.

On the contrary, cities are very static and stable real-life entities. It is highly improbable to expect sudden changes in either area size or population. Therefore, no duplication or reduction of samples was made.

This imbalanced dataset problem can be addressed by giving underrepresented classes more weight. A model should be more penalized during the training phase if it misclassifies a sample from an underrepresented class.

One such technique utilizes the volume of samples per class, also called the **effective number of samples** (ENS) [5].

The effective number of samples can be calculated by using this formula:

$$\mathrm{ENS}_c = \frac{1-\beta^{n_c}}{1-\beta} \tag{3.1}$$

In this equation, n_c is the number of samples per class c, while β is a hyperparameter, typically $\beta \in [0.9, 0.99]$.

By multiplying the originally calculated loss and the effective number of samples per class, the underrepresented classes will contribute more to the total loss.

4. Selected Models

4.1. Attention-Based Models

4.1.1. Attention is All You Need

In 2017, the paper [20] revolutionized the field of deep learning, especially sequence-to-sequence learning.

The **attention mechanism** was developed as a solution for the limitations traditional sequence-to-sequence models faced. Attention allows a model to *focus*, or, in other words, to *attend* to specific parts of the input data when generating the output.

Self-attention is a mechanism that lets an input sequence create dependencies with any other part of itself. Self-attention does not only consider local context but rather determines the importance of various positions in the input data, enabling the model to capture different types of relationships in the input sequence.

Multi-head attention implies running multiple attention mechanisms in parallel, allowing the model to focus on different types of information from various input parts. The term *head* refers to multiple sets of attention weights. Outputs from all multi-head attentions are concatenated and linearly transformed to produce the final output.

4.1.2. Graph Attention Network

Introduced in [21], the **Graph Attention Network** (GAT) is an attention-based architecture designed primarily for node classification. This model leverages the *self-attention* mechanism, which makes it highly effective and directly applicable to inductive learning problems. GAT achieves state-of-the-art results and is one of the most used GNNs.

A single GAT layer includes several steps:

Attention Coefficients Computation. The model initially computes **attention coefficients** which determine how important a node's (intermediate) neighbor is. This can be expressed as follows:

$$e_{ij} = a(\mathbf{W}\mathbf{h}_i, \mathbf{W}\mathbf{h}_j) \tag{4.1}$$

where e_{ij} is the attention score (coefficient) that denotes how important node j's features are to node i. Vectors $\mathbf{h}_i$ and $\mathbf{h}_j$ are the feature vectors of nodes i and j, respectively. Matrix $\mathbf{W}$ is a shared, learnable matrix used to linearly transform feature vectors and a is the attention mechanism which is typically implemented as a single-layer feed-forward neural network.

Coefficient Normalization. To be able to compare attention coefficients, they have to be normalized:

$$\alpha_{ij} = \text{softmax}(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik})} \tag{4.2}$$

where α_{ij} is the normalized attention coefficient.

The attention mechanism a used in the original paper is a single-layer feed-forward neural network, parameterized by a weight vector $\mathbf{a}^\top$, and including LeakyReLU as the activation function:

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^\top[\mathbf{W}\mathbf{h}_i || \mathbf{W}\mathbf{h}_j]))}{\sum_{k \in \mathcal{N}(i)} \exp(\text{LeakyReLU}(\mathbf{a}^\top[\mathbf{W}\mathbf{h}_i || \mathbf{W}\mathbf{h}_k]))} \tag{4.3}$$

where $||$ is the concatenation operator, which essentially merges two (in this context) vectors together.

Figure 4.1 visualizes the attention mechanism used in the original paper.

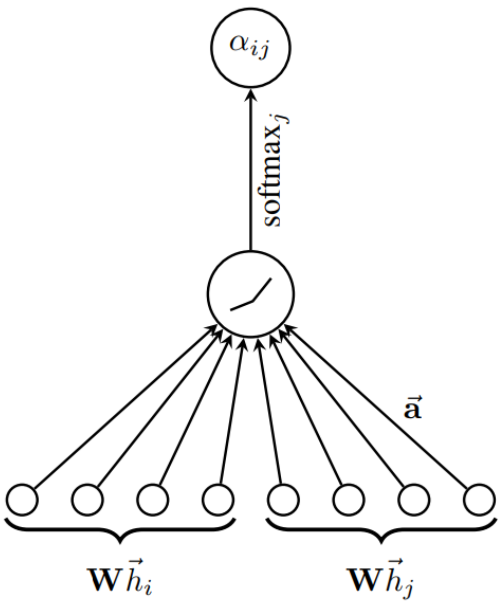

Figure 4.1: Illustration of the attention mechanism used in [21]

Edge Attributes. This formula can be extended to include edge features as follows:

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^\top[\mathbf{W}\mathbf{h}_i||\mathbf{W}\mathbf{h}_j||\mathbf{W}_e\mathbf{e}_{ij}]))}{\sum_{k\in\mathcal{N}(i)}\exp(\text{LeakyReLU}(\mathbf{a}^\top[\mathbf{W}\mathbf{h}_i||\mathbf{W}\mathbf{h}_k||\mathbf{W}_e\mathbf{e}_{ik}]))} \tag{4.4}$$

Output Features. After calculating the attention scores between nodes, GAT generates a feature vector for every node using this formula:

$$\mathbf{h}_i' = \phi\left(\sum_{j\in\mathcal{N}(i)}\alpha_{ij}\mathbf{W}\mathbf{h}_j\right) \tag{4.5}$$

where $\mathbf{h}_i'$ is the new feature vector of node i and ϕ is any non-linear activation function.

Multi-Head Attention.

Applying multi-head attention can be beneficial to capturing different types of patterns and relationships between nodes. This is expressed with the following formula:

$$\mathbf{h}_i' = \Big\Vert_{k=1}^{K}\phi\left(\sum_{j\in\mathcal{N}(i)}\alpha_{ij}^{k}\mathbf{W}^{\mathbf{k}}\mathbf{h}_j\right) \tag{4.6}$$

where K represents the number of heads, and $||$ denotes the concatenation all K vectors.

The final output layer does not concatenate the results but averages them instead:

$$\mathbf{h}'_i = \phi\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}(i)} \alpha_{ij}^k \mathbf{W}^k \mathbf{h}_j\right) \tag{4.7}$$

Figure 4.2 illustrates multi-head attention with three heads.

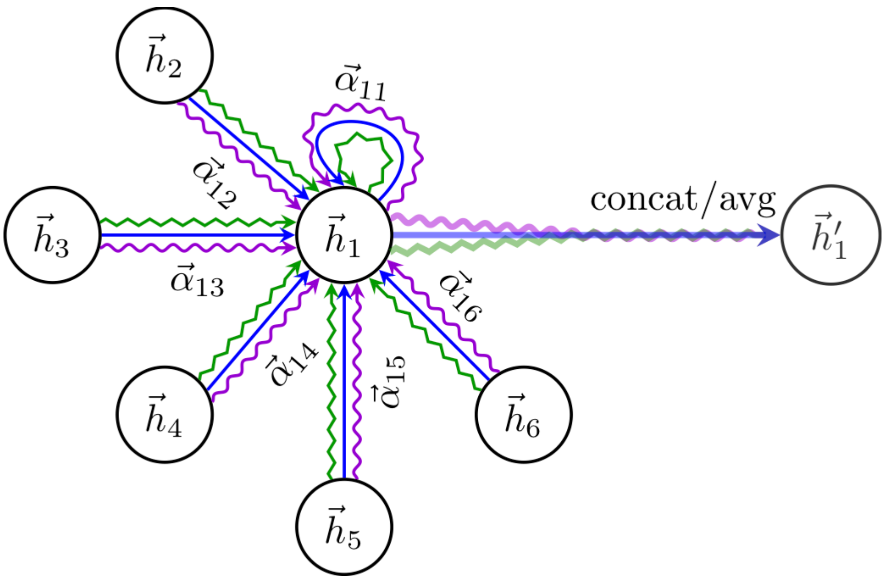

Figure 4.2: Illustration of multi-head attention from [21]

4.1.3. Graph Attention Network v2

In [3], the authors analyzed the GAT architecture and concluded that there is a way to improve it and make it more expressive. They call this graph neural network *GATv2*.

The authors defined the terms **static** and **dynamic** attention. Static attention mechanisms compute attention weights once, remaining unchanged throughout a particular operation or process. They do not adapt based on the context. Dynamic attention mechanisms recalculate the attention weights at every step and adapt based on the context.

The attention mechanism in the original GAT can be represented as:

$$e_{ij}(\mathbf{h}_i, \mathbf{h}_j) = \text{LeakyReLU}(\mathbf{a}^\top[\mathbf{W}\mathbf{h}_i || \mathbf{W}\mathbf{h}_j]) \tag{4.8}$$

The original GAT utilizes a static attention mechanism. However, by applying a small change, the attention mechanism gains the ability to dynamically adapt to the context.

The GATv2 evolves the attention mechanism to:

$$e_{ij}(\mathbf{h}_i, \mathbf{h}_j) = \mathbf{a}^\top \text{LeakyReLU}([\mathbf{W}\mathbf{h}_i || \mathbf{W}\mathbf{h}_j]) \tag{4.9}$$

The second formula enhances the expressiveness of the GAT model and achieves slightly better results on benchmark datasets.

4.2. Message-Passing GNNs

4.2.1. The Weisfeiler-Lehman Test

Determining whether or not two finite graphs are isomorphic is a complex problem. The isomorphism question is considered to be in the computational complexity class *NP-intermediate*.

In 1968, B. Weisfeiler and A. Lehman proposed an efficient heuristic that can distinguish two non-isomorphic graphs - the **Weisfeiler-Lehman test**.

The Weisfeiler-Lehman (WL) test is an iterative algorithm based on graph labeling. Initially, each node in both graphs is assigned a label based on a simple property (e.g., degree). These labels are updated in each iteration. The new labels are generated by a hash function that takes as input a multiset of labels from neighboring nodes, as well as the node's current label. This updating procedure continues until all nodes converge, i.e., until the labels of the previous step are the same as the labels of the current step. If, at any point, the two graphs have different labels, they are deemed non-isomorphic. However, if the algorithm completes and the two graphs have the same labels, they *might be* isomorphic.

Algorithm 1 provides a code-like, high-level depiction of the WL test[15].

The WL test is prone to producing false positives, declaring two graphs to be isomorphic when they are not. There are a few known graphs that the WL test cannot distinguish, called *WL-counterexamples*.

A perfect learning algorithm would produce identical node embeddings for two graphs if and only if they are isomorphic. However, there is no evidence of the existence of such an algorithm, and to this day, the WL test is considered to be the most powerful graph isomorphism test.

[15]This pseudocode is mostly derived from the procedure's description in [8].

Algorithm 1 The Weisfeiler-Lehman Test

```
procedure WL($\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1), \mathcal{G}_2 = (\mathcal{V}_2, \mathcal{E}_2)$)
    $L_{G_1} \leftarrow \varnothing$
    $L_{G_2} \leftarrow \varnothing$
    for all $v \in \mathcal{V}_i, i \in \{1, 2\}$ do
        if $v$ has discrete feature(s) $\mathbf{x}_v$ then
            $l^{(0)}_{\mathcal{G}_i}(v) \leftarrow \mathbf{x}_v$
        else
            $l^{(0)}_{\mathcal{G}_i}(v) \leftarrow deg(v)$
        end if
    end for
    repeat
        for all $v \in \mathcal{V}_i, i \in \{1, 2\}$ do
            $l^{(j)}_{\mathcal{G}_i}(v) \leftarrow \text{HASH}\left(l^{(j-1)}_{\mathcal{G}_i}(v), \{\{l^{(j-1)}_{\mathcal{G}_i}(u), \forall u \in \mathcal{N}(v)\}\}\right)$
        end for
    until $K$                                        ▷ state where both graphs converge
    for all $v \in \mathcal{V}_i, i \in \{1, 2\}$ do
        for $j = 0$ to $K - 1$ do
            Add $l^{(j)}_{\mathcal{G}_i}(v)$ to $L_{G_i}$
        end for
    end for
    if $L_{G_1} = L_{G_2}$ then
        return True
    else
        return False
    end if
end procedure
```

4.2.2. Graph Isomorphism Network

Some analogies between the WL test and GNNs can be observed - each iteration aggregates information from the local neighborhood to update the representation of each node. The WL test uses a *hash function* to aggregate and update discrete labels. On the other hand, GNNs use *neural networks* to aggregate and update continuous node embeddings.

In [22], the authors talk about how *powerful* GNNs are, i.e., how accurate they can distinguish between different graph structures. They concluded that GNNs are *no more powerful than the WL test*.

GNNs can be *as powerful as* the WL test under these conditions:

1. The GNN aggregates and updates node features iteratively with:

$$\mathbf{h}_v^{(k+1)} = \phi\Big(\mathbf{h}_v^{(k)}, f\Big(\Big\{\mathbf{h}_u^{(k)} : u \in \mathcal{N}(v)\Big\}\Big)\Big)$$

   where the functions f and ϕ are injective.

2. The GNN applies a graph-level readout function that operates on a multiset of feature nodes and is injective.

Here, f is any *aggregation* function and ϕ is the *update* function that takes two arguments.

After establishing conditions for a maximally powerful GNN, authors in [22] proposed a GNN architecture that satisfies these conditions - the **Graph Isomorphism Network** (GIN).

In GIN, node representations are updated using the following formula:

$$\mathbf{h}_v^{(k+1)} = \text{MLP}^{(k+1)}\big((1 + \epsilon^{(k+1)}) \cdot \mathbf{h}_v^{(k)} + \sum_{u \in \mathcal{N}(v)} \mathbf{h}_u^{(k)}\big), \tag{4.10}$$

The possible readout functions mentioned in [22], ranked by expressive power, are **sum** $>$ **mean** $>$ **max**.

From the node's update formula, it is clear that GIN does not utilize the information provided by edges. Authors of [10] have implemented a modified version of GIN that includes edge features - **GINE**[16].

[16]This modification has no formal name. The name *GINE* is used in PyG's implementation of the modified GIN.

This modification can be expressed with the following formula:

$$\mathbf{h}_v^{(k+1)} = \text{MLP}^{(k+1)}\Big((1 + \epsilon^{(k+1)}) \cdot \mathbf{h}_v^{(k)} + \sum_{u \in \mathcal{N}(v)} \text{ReLU}(\mathbf{h}_u^{(k)} + \mathbf{e}_{u,v})\Big) \tag{4.11}$$

4.3. Implemented Architectures

All three of the implemented models used *cross-entropy loss* as the loss function and *Adam* as the optimizer.

They all use the following hyperparameters:

- `batch_size`: 8
- `learning_rate`: 0.01
- `beta`: 0.99

The **learning rate** is a hyperparameter the optimizer uses to determine the step size. A smaller step size may result in a more accurate convergence but a longer training process as a consequence. On the other hand, a larger step size will speed up the training process but may be at risk of overshooting the optimal solution.

A **mini-batch** is a subset of the entire dataset of the size `batch_size`. The model performs a forward pass using this subset, calculates the loss, and only then updates the model's weights.

The `beta` hyperparameter is used in the ENS function mentioned in chapter 3.4.

4.3.1. GAT and GATv2

GAT and GATv2 were implemented using the same architecture and hyperparameters with the distinction being the convolutional layer. This allows for their direct comparison.

Both models have the following hyperparameters:

- `hidden_channels1`: 16
- `hidden_channels2`: 16
- `hidden_channels3`: 16
- `heads1`: 8
- `heads2`: 8
- `heads3`: 4

The following is the layer-wise breakdown of the implemented GAT model:

1. **First GAT Layer**:
   - Input size: `#nodes_per_batch`×1 (`#node_features`)
   - Applys the GAT convolution with 8 attention heads.
   - Each head outputs a node representation of size 16 (`hidden_channels1`).
   - These outputs are concatenated, leading to a node representation of size 128 (`hidden_channels1` · `heads1`).
   - Output size: `#nodes_per_batch`×128
   - Applys ELU activation element-wise.

2. **Second GAT Layer**:
   - Input size: `#nodes_per_batch`×128
   - Applys the GAT convolution with 8 attention heads.
   - Each head outputs a node representation of size 16 (`hidden_channels2`).
   - These outputs are concatenated, leading to a node representation of size 128 (`hidden_channels2` · `heads2`).
   - Output size: `#nodes_per_batch`×128
   - Applys ELU activation element-wise.

3. **Third GAT Layer**:
   - Input size: `#nodes_per_batch`×128
   - Applys the GAT convolution with 4 attention heads.
   - Each head outputs a node representation of size 16 (`hidden_channels3`).
   - These outputs are averaged, leading to a node representation of size 16.
   - Output size: `#nodes_per_batch`×16
   - Applys ELU activation element-wise.

4. **Global Pooling Layers**:
   - Input size (for each pooling type): `#nodes_per_batch`×16
   - Three types of pooling layers are applied to nodes to get a graph representation: mean, max, and sum pooling.
   - Output size (for each pooling type): `batch_size`×16
   - Output size after concatenating these node representations: `batch_size`×48

5. **Linear Layer**:
   - Input size: `batch_size`×48
   - A fully connected layer that transforms the given input into the dimension matching the number of classes.
   - Output size: `batch_size`×30 (`#classes`)

The number `#nodes_per_batch` represents the number of nodes for each mini-batch and thus likely changes in each iteration. Replacing `GATConv` with `GATv2Conv` yields the GATv2 model.

Figure 4.3 depicts a high-level overview of the previously described architecture.

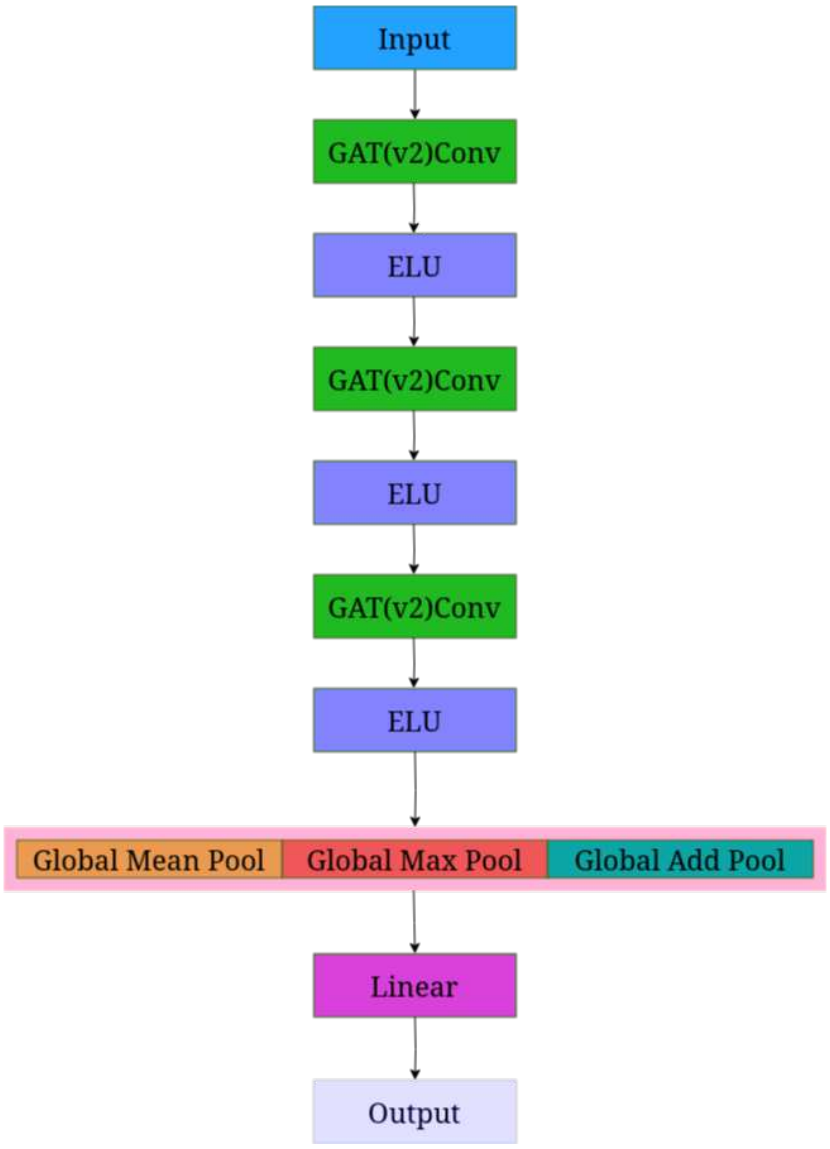

Figure 4.3: Implemented architectures of GAT and GATv2

One iteration contains the following steps:

1. **Forward pass**
   - The model gets an input of `batch_size` graphs, together with all its features.

  - After passing through each described layer, the model produces a raw output vector for each of the input graphs.

2. **Loss computation**
  - These raw output vectors are directly used in the cross-entropy function to calculate the loss[17].
  - Corrects the loss if ENS is used by penalizing the model for misclassifying smaller countries.

3. **Backpropagation**
  - Calculates gradients of the loss concerning the model's parameters (e.g., shareable matrices $\mathbf{W}$).

4. **Weight updates**
  - The optimizer (Adam) uses the calculated gradients to adjust the model's parameters.

4.3.2. GIN

The GIN implementation uses the following hyperparameters:

- `hidden_channels1`: 16
- `hidden_channels2`: 32
- `hidden_channels3`: 32
- `epsilon`: 0.26

The hyperparameter `epsilon` is trainable and its above-stated value is only the initial value.

The following is the layer-wise breakdown of the implemented GIN model:

1. **First GINE Layer**:
  - Input size: `#nodes_per_batch`$\times 1$ (`#node_features`)
  - Applys the GINE convolution with an embedded MLP which has two linear layers (both of size 16 (`hidden_channels1`)) with BatchNorm1d and ReLU in between.
  - Output size: `#nodes_per_batch`$\times 16$ (`hidden_channels1`)
  - Applys ELU activation element-wise.

[17]Cross-entropy loss calculates the probabilities of a graph belonging to each class before calculating the loss.

2. **Second GINE Layer**:
   - Input size: `#nodes_per_batch`×16 (`hidden_channels1`)
   - Applys the GINE convolution with an embedded MLP which has two linear layers (both of size 32 `hidden_channels2`) with BatchNorm1d and ReLU in between.
   - Output size: `#nodes_per_batch`×32 (`hidden_channels2`)
   - Applys ELU activation element-wise.
3. **Third GINE Layer**:
   - Input size: `#nodes_per_batch`×32 (`hidden_channels2`)
   - Applys the GINE convolution with an embedded MLP which has two linear layers (both of size 32 (`hidden_channels3`)) with BatchNorm1d and ReLU in between.
   - Output size: `#nodes_per_batch`×32 (`hidden_channels3`)
   - Applys ELU activation element-wise.
4. **Global Pooling Layers**:
   - Input size (for each pooling type): `#nodes_per_batch`×32 (`hidden_channels3`)
   - Three types of pooling layers are applied to nodes to get a graph representation: mean, max, and sum pooling.
   - Output size (for each pooling type): `batch_size`×32 (`hidden_channels3`)
   - Output size after concatenating these node representations: `batch_size`×96
5. **Final Linear Layer**:
   - Input size: `batch_size`×96
   - A fully connected layer that transforms the concatenated graph representations into the dimension matching the number of classes.
   - Output size: `batch_size`×30 (`#classes`)

Figure 4.4 displays the high-level overview of GIN's implemented architecture.

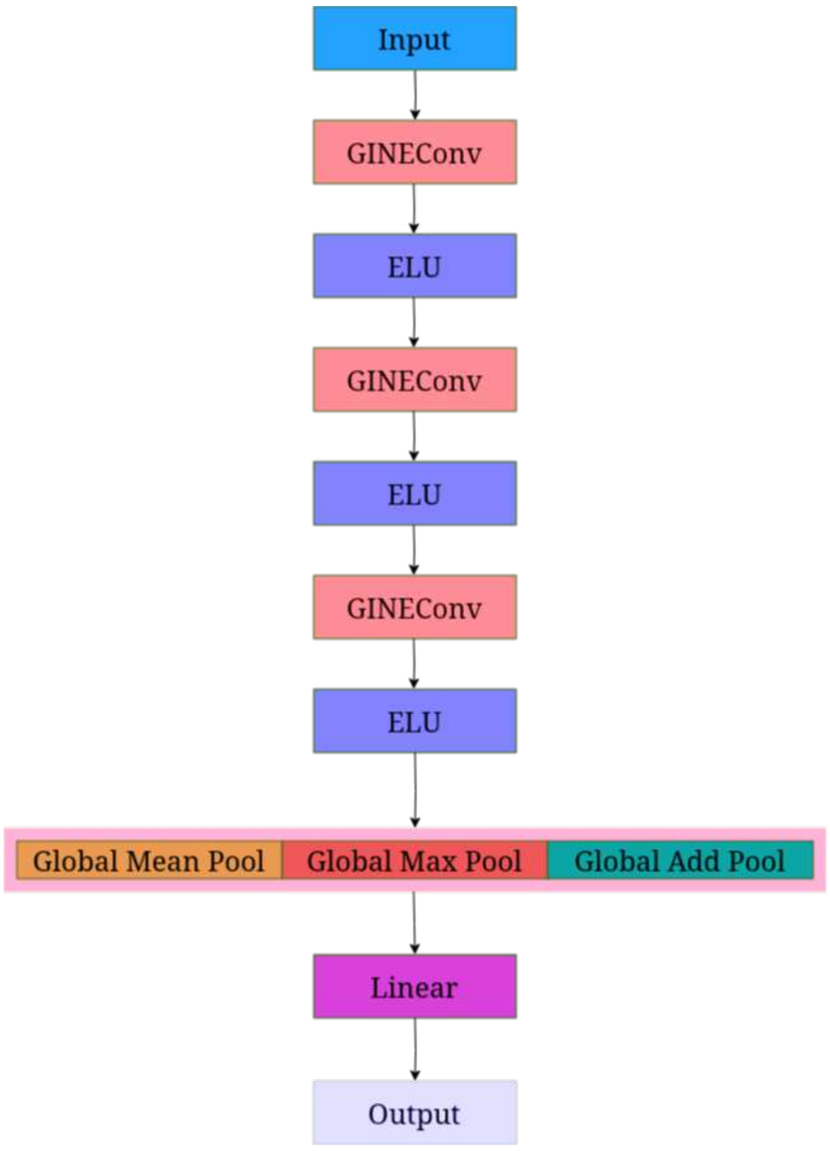

Figure 4.4: Implemented GIN architecture

The GINE convolution layer has an internal function that performs the *aggregation* function. The MLP serves as the *update* function.

Figure 4.5 depicts the architecture of the implemented MLP.

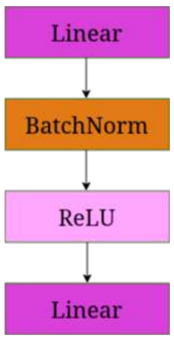

Figure 4.5: Implemented MLP architecture

BatchNorm1d applies batch normalization over a 2D or 3D input. The details of this normalization can be found in [11].

5. Model Evaluation and Results

5.1. Cross-Validation

Cross-validation is a resampling technique used for evaluating machine learning models whose purpose is to get an unbiased estimate of the model's performance. This procedure has a hyperparameter k that denotes the number of folds (groups) the dataset is split into, which is the reason why cross-validation is often referred to as k-fold cross-validation.

Outlined below are the steps of the cross-validation procedure:

1. Shuffle the dataset randomly.
2. Split the dataset into k folds.
3. For each fold:
    - 3.1. Set aside the current fold as the validation dataset.
    - 3.2. Treat the remaining data as the training dataset.
    - 3.3. Train the model on the training dataset and then evaluate it on the validation dataset.
    - 3.4. Store the evaluation score.
4. Average all evaluation scores to obtain a final evaluation score.

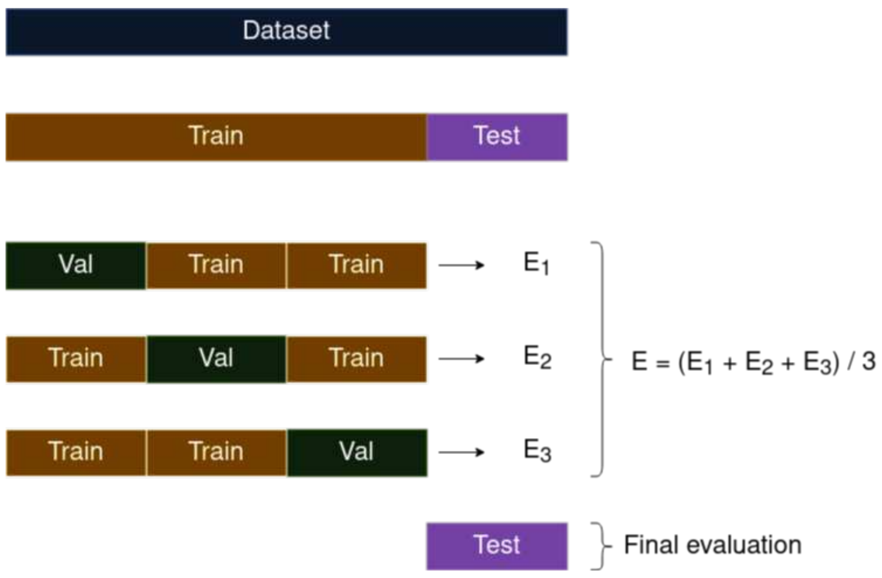

Figure 5.1: 3-fold cross-validation

The resulting score provides insight into how well a model performs on certain metrics. Typically, the next step is to train that observed model on the whole dataset employed during cross-validation. Once trained, this model is either evaluated on a test dataset to validate the performance or deployed and awaiting new incoming data.

When dealing with an imbalanced dataset, as is the case in this thesis, it is crucial for a valid performance evaluation to ensure that each fold in the cross-validation process has a similar proportion of samples from each class as the entire (train) dataset. To achieve this, only a small detail in the cross-validation process has to be changed - instead of randomly sampling the folds, a *stratified* sampling strategy has to be used. This change in the cross-validation process is called **stratified k-fold cross-validation**.

In this thesis, the models are evaluated using a stratified 3-fold cross-validation, as depicted in Figure 5.1.

5.2. Evaluation Metrics

In any machine learning task, choosing the appropriate evaluation metric for performance measurement is a crucial step. The choice depends on multiple factors. A primary consideration is what insight one aims to extract from these evaluations and how interpretable the used metrics are. Another factor to consider is the distribution of data and its possible imbalance.

While there are numerous metrics to choose from for multiclass classification, accuracy, precision, recall, and F1-score are frequently the first ones to be considered.

Precision, recall, and F1-score were originally designed for binary classification problems but can easily be adapted to multiclass classification. In this adaptation, data from one class is treated as *positive* (P), while the remaining data is treated as *negative* (N). This is done for each class individually.

To be able to define the mentioned metrics, the following terms need to be defined:

– **True positive** (TP) - The model correctly identifies data as belonging to the positive class.
– **True negative** (TN) - The model correctly identifies data as belonging to the negative class.
– **False positive** (FP) - The model mistakenly classifies data as positive when it actually belongs to the negative class.
– **False negative** (FN) - The model mistakenly classifies data as negative when it actually belongs to the positive class.

Accuracy is a very straightforward metric. It is calculated as the ratio of correct predictions to the number of total predictions. While accuracy is intuitive and easy to understand, it can sometimes report misleadingly optimistic evaluations, especially when working with highly imbalanced datasets. Accuracy can be expressed with the following formula:

$$\text{accuracy} = \frac{\text{TP+TN}}{\text{TP+FP+TN+FN}}$$

Precision is the model's ability not to label a negative sample as positive. Precision can be seen as a measure of quality. The formula for calculating precision is the following:

$$\text{precision} = \frac{\text{TP}}{\text{TP+FP}}$$

Recall is the model's ability to find all positive samples. Recall can be seen as a measure of quantity. The formula for calculating recall is the following:

$$\text{recall} = \frac{\text{TP}}{\text{TP+FN}}$$

In the context of this thesis' task, precision and recall can be further clarified by looking at a specific country, e.g. Croatia:

– **Precision** answers the question: Out of all the cities the model predicted to be in Croatia, how many were truly Croatian cities?
– **Recall** answers the question: Out of all the Croatian cities in the dataset, how

many did the model correctly identify as being in Croatia?

The $\mathbf{F}_1-$score is the harmonic mean of precision and recall. It is a less intuitive but very useful metric. The following formula is used to calculate the F_1-score:

$$\text{F1-score} = \frac{2}{\frac{1}{\text{precision}}+\frac{1}{\text{recall}}} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision}+\text{recall}} = \frac{\text{TP}}{\text{TP}+\frac{1}{2}(\text{FP}+\text{FN})}$$

These metrics ought to be maximized. A general rule of thumb is that a model that achieves 70% - 90% on these metrics can be considered a *good* model.

In the multiclass setting, all of these binary metrics can be aggregated in two ways:

1. **Macro-average** - computes the metric independently for each class and then takes the average. This treats all classes equally and is therefore a good choice when dealing with imbalanced data.
2. **Micro-average** - the individual metrics for all classes are aggregated and then the average metric is computed. This average can be influenced more by larger classes and is therefore a better choice when the data is somewhat balanced.

In this thesis, macro averaging was used.

5.3. Result Comparison

In order to get an unbiased evaluation, the models were evaluated on the train subset depicted in Table 3.3 using 3-fold cross-validation. Tables 5.1 and 5.2 show the results without and with the usage of ENS, respectively.

Table 5.1: Evaluation results

	Accuracy	Precision	Recall	Macro-F1
GAT	0.1545	0.1627	0.1545	0.1320
GATv2	0.2610	0.2503	0.2610	0.2204
GIN	0.3998	0.4272	0.3998	0.3734

Table 5.2: Evaluation results (with ENS)

	Accuracy	Precision	Recall	Macro-F1
GAT	0.1732	0.1620	0.1732	0.1351
GATv2	0.2046	0.2019	0.2046	0.1568
GIN	0.4155	0.4057	0.4155	0.3783

To test if the evaluation was correct, all the models were trained on the whole train subset and tested on the test subset. The results are shown in Tables 5.3 and 5.4, without and with ENS, respectively.

Table 5.3: Test results

	Accuracy	Precision	Recall	Macro-F1
GAT	0.2342	0.1699	0.2342	0.1891
GATv2	0.2697	0.2822	0.2697	0.2500
GIN	0.4618	0.4616	0.4618	0.4247

Table 5.4: Test results (with ENS)

	Accuracy	Precision	Recall	Macro-F1
GAT	0.1355	0.1185	0.1355	0.1168
GATv2	0.2683	0.2597	0.2683	0.2421
GIN	0.4985	0.4983	0.4985	0.4483

Tables from 5.5 to 5.10 show results on the test dataset per country. The absence of a country in the tables means that all the metrics were 0.

It is important to remark that if the dataset had been split differently, the results would have (very likely) been different.

Country	**Accuracy**	**Precision**	**Recall**	**F1-score**
BiH	1.0	0.5	1.0	0.6667
France	0.1818	0.25	0.1818	0.2105
Germany	0.5789	0.44	0.5789	0.5
Greece	0.5	0.4	0.5	0.4444
Italy	0.0667	0.2	0.0667	0.1
Netherlands	0.6667	0.4444	0.6667	0.5333
Norway	0.5	0.25	0.5	0.3333
Poland	0.1	0.1111	0.1	0.1053
Portugal	0.6667	0.4	0.6667	0.5
Russia	0.9444	0.4595	0.9444	0.6182
Spain	0.7857	0.44	0.7857	0.5641
Switzerland	0.9091	0.9524	0.9091	0.9302
Ukraine	0.125	0.25	0.125	0.1667

Table 5.5: Metrics per country (GAT)

Country	**Accuracy**	**Precision**	**Recall**	**F1-score**
France	0.1818	0.3333	0.1818	0.2353
Greece	0.25	0.5	0.25	0.3333
Italy	0.0667	0.0455	0.0667	0.0541
Netherlands	0.5	0.4286	0.5	0.4615
Norway	0.5	0.0588	0.5	0.1053
Portugal	0.5	0.5	0.5	0.5
Russia	0.4444	0.2353	0.4444	0.3077
Spain	0.7143	0.4545	0.7143	0.5556
UK	0.9091	1.0	0.9091	0.9524

Table 5.6: Metrics per country (GAT + ENS)

Country	**Accuracy**	**Precision**	**Recall**	**F1-score**
Belarus	0.5000	0.2500	0.5000	0.3333
BiH	0.5000	0.3333	0.5000	0.4000
Bulgaria	0.5000	0.1667	0.5000	0.2500
Denmark	0.3333	1.0000	0.3333	0.5000
Finland	0.5000	0.2000	0.5000	0.2857
France	0.6364	0.4375	0.6364	0.5185
Germany	0.6316	0.6316	0.6316	0.6316
Greece	0.2500	0.3333	0.2500	0.2857
Italy	0.3333	0.2632	0.3333	0.2941
Netherlands	0.8333	0.8333	0.8333	0.8333
Poland	0.1000	0.3333	0.1000	0.1538
Portugal	0.1667	1.0000	0.1667	0.2857
Russia	0.6111	0.6111	0.6111	0.6111
Spain	0.7857	0.7857	0.7857	0.7857
UK	0.9091	0.9524	0.9091	0.9302
Ukraine	0.5000	0.3333	0.5000	0.4000

Table 5.7: Metrics per country (GATv2)

Country	**Accuracy**	**Precision**	**Recall**	**F1-score**
Belgium	0.3333	0.3333	0.3333	0.3333
BiH	0.5000	0.1429	0.5000	0.2222
Bulgaria	0.5000	0.1429	0.5000	0.2222
Denmark	0.3333	0.1667	0.3333	0.2222
Finland	0.5000	0.2000	0.5000	0.2857
France	0.2727	0.5000	0.2727	0.3529
Germany	0.3158	0.5455	0.3158	0.4000
Greece	0.2500	0.2000	0.2500	0.2222
Italy	0.4000	0.3750	0.4000	0.3871
Netherlands	0.5000	0.7500	0.5000	0.6000
Poland	0.2000	0.6667	0.2000	0.3077
Portugal	0.5000	0.7500	0.5000	0.6000
Romania	0.7500	0.3750	0.7500	0.5000
Russia	0.5556	0.5000	0.5556	0.5263
Spain	0.7857	0.7857	0.7857	0.7857
UK	0.7273	1.0000	0.7273	0.8421
Ukraine	0.6250	0.3571	0.6250	0.4545

Table 5.8: Metrics per country (GATv2 + ENS)

Country	**Accuracy**	**Precision**	**Recall**	**F1-score**
Belgium	1.0000	0.3000	1.0000	0.4615
BiH	0.5000	1.0000	0.5000	0.6667
Bulgaria	0.5000	0.1429	0.5000	0.2222
Finland	0.5000	1.0000	0.5000	0.6667
France	0.1818	1.0000	0.1818	0.3077
Germany	1.0000	0.9048	1.0000	0.9500
Greece	1.0000	0.4000	1.0000	0.5714
Italy	0.4667	0.5385	0.4667	0.5000
Netherlands	1.0000	0.8571	1.0000	0.9231
Norway	0.5000	0.5000	0.5000	0.5000
Poland	0.4000	0.8000	0.4000	0.5333
Portugal	0.8333	0.5556	0.8333	0.6667
Romania	0.7500	0.5000	0.7500	0.6000
Russia	0.7778	0.6667	0.7778	0.7179
Serbia	0.5000	1.0000	0.5000	0.6667
Spain	0.7857	0.7857	0.7857	0.7857
Sweden	0.7500	0.5000	0.7500	0.6000
Switzerland	1.0000	1.0000	1.0000	1.0000
UK	0.9091	0.9524	0.9091	0.9302
Ukraine	0.5000	0.4444	0.5000	0.4706

Table 5.9: Metrics per country (GIN)

Country	**Accuracy**	**Precision**	**Recall**	**F1-score**
Belgium	0.3333	0.2000	0.3333	0.2500
Bulgaria	0.5000	0.5000	0.5000	0.5000
Czechia	0.5000	1.0000	0.5000	0.6667
Denmark	0.6667	1.0000	0.6667	0.8000
France	0.4545	0.6250	0.4545	0.5263
Germany	0.8947	0.9444	0.8947	0.9189
Greece	0.7500	0.7500	0.7500	0.7500
Hungary	1.0000	0.2000	1.0000	0.3333
Italy	0.3333	0.5556	0.3333	0.4167
Kosovo	1.0000	0.3333	1.0000	0.5000
Netherlands	0.8333	1.0000	0.8333	0.9091
Norway	0.5000	0.5000	0.5000	0.5000
Poland	0.2000	0.6667	0.2000	0.3077
Portugal	0.3333	1.0000	0.3333	0.5000
Romania	0.7500	1.0000	0.7500	0.8571
Russia	0.4444	1.0000	0.4444	0.6154
Slovakia	1.0000	0.1250	1.0000	0.2222
Spain	0.9286	0.5909	0.9286	0.7222
Sweden	0.7500	0.3750	0.7500	0.5000
Switzerland	1.0000	1.0000	1.0000	1.0000
UK	0.9091	1.0000	0.9091	0.9524
Ukraine	0.8750	0.5833	0.8750	0.7000

Table 5.10: Metrics per country (GIN + ENS)

6. Discussion

Choosing the appropriate evaluation metric is crucial for providing a fair assessment of models. An imbalanced dataset can pose a challenge, and by choosing the wrong metric, the model can report misleadingly optimistic results.

Using ENS to combat the imbalanced dataset slightly improved the overall performance.

GIN achieved the best results, which could be attributed to the fact that the message-passing convolution layers are more expressive than attentional convolution layers. It is possible that GIN was able to get a better grasp of the road networks' structure.

Overall, the results could be significantly better. The following modifications could potentially improve models' performances:

– Performing hyperparameter tuning.
– Expanding the dataset to include smaller cities.
– Adding more node features (e.g., *clustering coefficient*, *centrality measures*, and others).
– Integrating more layers to the existing GNNs.
– Exploring different types of GNNs.
– Implementing pooling layers.
– Experimenting with different optimizers.

It is worth mentioning that there is no guarantee that applying these changes will indeed improve model performance. It is possible that cities within a country do not have enough common characteristics, or that cities of different countries are too similar. However, such a conclusion can only be made after exhausting all available options.

7. Conclusion

The objective of this thesis was to delve into the realm of graph-based machine learning, with a specific focus on the graph classification task. One contribution of this thesis is the introduction of a novel dataset, representing European city road networks and their associated countries, thereby addressing the lack of traffic-related graph datasets. This thesis aimed to research if classifying these road networks into their respective countries is feasible. Given the high imbalance in the dataset, techniques were employed to address this issue. The results reported a slight improvement in the models' performance due to the utilization of these techniques. While the overall results are underwhelming, it's important to note that only simple models were used, leaving space for further improvements. Noticeably, the message-passing graph neural network outperformed the attentional networks by a factor of two, which implies that such models might perform better due to being more expressive. However, whether optimal results can be achieved for this classification problem remains an open question for future research.

REFERENCES

[1] Albert-László Barabási and Márton Pósfai. *Network science*. Cambridge University Press, Cambridge, 2016.

[2] Geoff Boeing. OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, Environment and Urban Systems*, 65:126–139, sep 2017.

[3] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks?, 2022.

[4] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges, 2021.

[5] Yin Cui, Menglin Jia, Tsung-Yi Lin, Yang Song, and Serge Belongie. Class-balanced loss based on effective number of samples, 2019.

[6] Austin Derrow-Pinion, Jennifer She, David Wong, Oliver Lange, Todd Hester, Luis Perez, Marc Nunkesser, Seongjae Lee, Xueying Guo, Brett Wiltshire, Peter W. Battaglia, Vishal Gupta, Ang Li, Zhongwen Xu, Alvaro Sanchez-Gonzalez, Yujia Li, and Petar Velickovic. ETA prediction with graph neural networks in google maps. In *Proceedings of the 30th ACM International Conference on Information &amp Knowledge Management*. ACM, oct 2021.

[7] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks, 2016.

[8] William L. Hamilton. Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3):1–159.

[9] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs, 2021.

[10] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks, 2020.

[11] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[12] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[13] Jure Leskovec. Graph neural networks, 2023. CS224W: Machine Learning with Graphs.

[14] Jure Leskovec. Node embeddings, 2023. CS224W: Machine Learning with Graphs.

[15] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[16] Christopher Morris, Nils M. Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. Tudataset: A collection of benchmark datasets for learning with graphs, 2020.

[17] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, aug 2014.

[18] Ahmad Rawashdeh and Anca Ralescu. Similarity measure for social networks – a brief survey. volume 1353, 04 2015.

[19] Jonathan M. Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M. Donghia, Craig R. MacNair, Shawn French, Lindsey A. Carfrae, Zohar Bloom-Ackermann, Victoria M. Tran, Anush Chiappino-Pepe, Ahmed H. Badran, Ian W. Andrews, Emma J. Chory, George M. Church, Eric D. Brown, Tommi S. Jaakkola, Regina Barzilay, and James J. Collins. A deep learning approach to antibiotic discovery. *Cell*, 180(4):688–702.e13, February 2020.

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[21] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

[22] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2019.

[23] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery &amp Data Mining*. ACM, jul 2018.

[24] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling, 2019.

Graph classification using machine learning techniques

Abstract

The field of graph-based machine learning has recently gained a significant amount of attention due to its potential to solve complex real-world problems that can be modeled as graphs. This thesis delves into the problem of graph classification by creating a unique dataset that represents road networks across European cities and setting an objective to train models to classify these cities into their respective countries. This task was approached by using different graph neural networks (GNNs), such as GAT and GIN. Although this thesis yields no definite conclusion regarding the feasibility of this task, it serves as an initial step in further research of this challenge.

Keywords: graph classification, graph neural networks, GNNs, GAT, GIN

Klasifikacija grafova korištenjem metoda strojnog učenja

Sažetak

Područje strojnog učenja temeljeno na grafovima je u posljednje vrijeme privuklo značajnu pozornost zbog svojeg potencijala rješavanja složenih realnih problema koji se mogu modelirati pomoću strukture grafa. Ovaj rad proučava problem klasifikacije grafova stvarajući novi skup podataka, koji predstavlja cestovne mreže europskih gradova, s ciljem da se modele istrenira klasificirati gradove u njihovu odgovarajuću državu. Problemu se pristupilo koristeći različite neuronske mreže grafova (GNNs) kao što su GAT i GIN. Iako ne donosi finalni zaključak o izvedivosti ovog zadatka, ovaj rad služi kao početni korak u daljnjem istraživanju ovog problema.

Ključne riječi: klasifikacija grafova, graf neuronske mreže, GNNs, GAT, GIN