ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

EDGE-AWARE GRAPH ATTENTION NETWORKS: JOINT REASONING ON TEXT AND KNOWLEDGE GRAPHS FOR BIOMEDICAL QUESTION ANSWERING

Elaborato in
Text Mining

Relatore
Prof. Gianluca Moro

Co-relatore
Dott. Giacomo Frisoni

Presentata da
Francesco Boschi

Appello straordinario riservato
Anno Accademico 2020 – 2021

PAROLE CHIAVE

Graph Neural Networks

Natural Language Processing

Machine Learning

Deep Neural Networks

Python

Even if it's just for a moment
I'm gonna burn so bright and so red,
I'd dazzle everyone.
And all that'll be left is pure white ash.
- Ashita no Joe

Sommario

Buona parte dei dati provenienti dal mondo reale, tra i quali quelli testuali, possono essere rappresentati mediante delle strutture a grafo. L'utilizzo di grafi per rappresentare dati testuali porta con se numerosi vantaggi, principalmente legati alla possibilità di mantenere una maggiore quantità di informazioni, come le relazioni tra le parole e la loro tipologia.
Negli utlimi anni sono state proposte numerose architetture di reti neurali per affrontare task su grafi. Molte di queste tengono in considerazione solamente le caratteristiche dei nodi, ignorando o non dando la giusta rilevanza a quelle delle relazioni tra essi, nonostante queste ultime in numerosi task di node classification giochino un ruolo fondamentale.

Questa tesi si pone come obiettivo quello di analizzare le principali GNN, valutarne vantaggi e svantaggi, proporre una soluzione innovativa, considerabile come un'estensione di GAT, e applicarle ad un caso di studio in ambito biomedico.

Si propongono quindi le reti di riferimento, implementate con metodologie in seguito analizzate, e poi applicate ad un sistema di question answering in ambito biomedico in sostituzione alla GNN pre-esistente, nel tentativo di ottenere risultati migliori grazie all'utilizzo di modelli in grado di accettare in input sia le feature dei nodi che quelle degli archi.
Come sarà mostrato in seguito, i modelli da noi proposti sono in grado di battere la soluzione originale e definire il nuovo stato dell'arte per il task in analisi.

Abstract

Much of the real-world dataset, including textual data, can be represented using graph structures. The use of graphs to represent textual data has many advantages, mainly related to maintaining a more significant amount of information, such as the relationships between words and their types. In recent years, many neural network architectures have been proposed to deal with tasks on graphs. Many of them consider only node features, ignoring or not giving the proper relevance to relationships between them. However, in many node classification tasks, they play a fundamental role.

This thesis aims to analyze the main GNNs, evaluate their advantages and disadvantages, propose an innovative solution considered as an extension of GAT, and apply them to a case study in the biomedical field.

We propose the reference GNNs, implemented with methodologies later analyzed, and then applied to a question answering system in the biomedical field as a replacement for the pre-existing GNN. We attempt to obtain better results by using models that can accept as input both node and edge features. As shown later, our proposed models can beat the original solution and define the state-of-the-art for the task under analysis.

Introduzione

Contesto

Buona parte dei dati provenienti dal mondo reale possono essere rappresentati mediante delle strutture a grafo. Pensiamo per esempio ai social network, nei quali gli utenti possono essere rappresentati come nodi e gli archi tra essi ne rappresentano le relazioni, o più in generale al World Wide Web.

Anche il testo, spesso presentato sotto forma non strutturata e di conseguenza più difficile da analizzare, può essere espresso mediante un grafo. Tale grafo può essere un semplice albero sintattico, nel quale emergono le categorie lessicali di ciascuna parola dipendentemente dal contesto in cui compare (figura 1), piuttosto che una struttura più complessa che permette di individuare le entità, ovvero i concetti presenti nel testo, e le relazioni che le legano, come gli Abstract Meaning Representation [1]. Gli AMR sono dei grafi etichettati, diretti e aciclici che contengono intere frasi e hanno come obiettivo quello di astrarre la rappresentazione sintattica, nel senso che due frasi con lo stesso significato ma non identiche sono assegnate allo stesso AMR (figura 2).

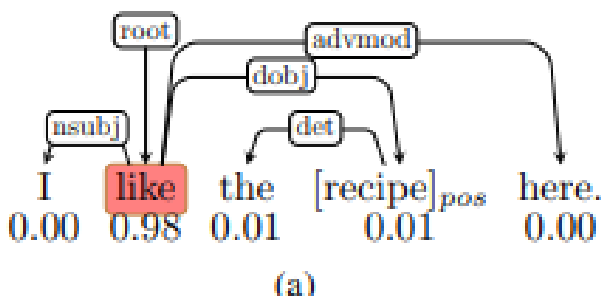

Figure 1: Example of syntax tree.
From: *Relational Graph Attention Network for Aspect-based Sentiment Analysis*, 2020 [2]

L'utilizzo di grafi per rappresentare dati testuali offre numerosi vantaggi: queste struttture sono infatti intuitive, flessibili e basate su regole matematiche

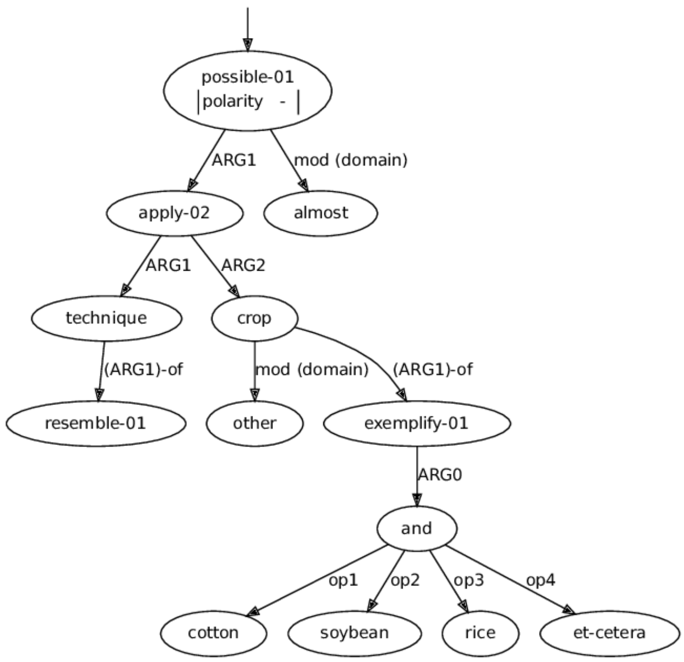

Figure 2: Example of AMR.
From: *MRP 2020: The Second Shared Task on Cross-Framework and Cross-Lingual Meaning Representation Parsing*, 2020 [3]

che forniscono un elevato grado di dimostrabilità. Inoltre mantengono, rispetto a del semplice testo, numerose informazioni aggiuntive, come relazioni e tipo di relazioni tra parole, tipologia delle entità, cardinalità ecc. che arricchiscono e disambiguano il testo rendendolo più facilmente utilizzabile per i task più disparati.

Tendenzialmente i nodi, gli archi e le caratteristiche di un grafo vengono trasformate e portate in uno spazio vettoriale di dimensione ridotta, cercando di preservare le proprietà e la struttura del grafo stesso: si parla di graph embedding.
Questa famiglia di algoritmi può essere suddivisa in varie sottocategorie che si differenziano per l'output ottenuto, per l'input richiesto e per la logica che viene applicata per il raggiungimento del risultato. Esistono infatti tecniche di graph embedding che mirano ad ottenere una rappresentazione per ciascun nodo, altre procedure che generano un solo embedding per l'intero grafo, caso più raro in quanto difficilmente scalabile e con maggior perdita di informazione. Una seconda considerazione riguarda il grafo richiesto in input; buona parte degli approcci accettano grafi omogenei, nei quali le relazioni non hanno direzione, tipologia e tutti i nodi svolgono lo stesso ruolo.

Negli ultimi anni questo ambito ha attirato l'attenzione di numerosi studiosi, che hanno inizialmente sviluppato reti neurali su grafi [4]. Kipf et al. [5] hanno proposto una rete convoluzionale sui grafi detta GCN, poi migliorata per tenere in considerazione le relazioni tra i nodi [6] ed infine evoluta grazie al lavoro di Velickovic et al. [7] "Graph Attention Networks" (GAT).

GAT unisce i vantaggi del meccanismo di Attention [8] e delle classiche GCN [5]: se prima durante l'aggregazione i nodi connessi venivano tutti considerati in egual maniera, grazie a GAT ciascun nodo sarà in automatico in grado di assegnare una rilevanza differente a ciascun nodo del vicinato.

Problema

Uno dei problemi principali delle reti esistenti risiede nel fatto che ignorano le caratteristiche delle relazioni, rappresentate mediante gli archi del grafo, nonostante spesso giochino un ruolo fondamentale nel comprendere appieno gli aspetti semantici del testo. La figura 3, per esempio, mostra come, mediante la tipologia delle relazioni, sia possibile classificare i nodi vicini a "David Beckham" in tre categorie: "locazione", "lavoro", "famiglia" e assegnare a ciascuno di essi una rilevanza differente dipendentemente dal contensto in analisi.

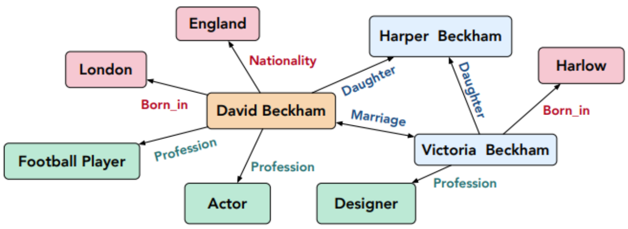

Figure 3: Example of multi-relational graph.
From: *r-GAT: Relational Graph Attention Network for Multi-Relational Graphs*, 2021 [2]

Sebbene ultimamente stiano nascendo numerose reti in grado di prendere in cosiderazione anche la tipologia delle relazioni [9][10][11][12][13][2], spesso quese soluzioni non sono general purpose, non prendono in considerazione tutte le caratteristiche tipiche di un grafo, non mettono a disposizione il codice dell'implementazione o non sfruttano il meccanismo di self-attention, risultando quindi inadatte alle nostre esigenze.

Una delle implementazioni più interessanti è sicuramente EGAT [14], in quanto, tramite un meccanismo di self-attention di ispirazione a GAT [7], genera una nuova rappresentazione per ciascun nodo e relazione tenendo in considerazione non solo i nodi del vicinato, ma anche la relazione che li unisce, sfruttando semplici trasformazioni sul grafo stesso.

Contributo

La tesi si colloca in questo contesto. Considerando come punto di partenza l'architettura di EGAT [14], della quale il codice non è reso pubblico, ci si pone come obiettivo quello reimplementare tale soluzione, proporne una variante e applicarle ad un secondo lavoro, QA-GNN [15], andando a sostituire la GNN utilizzata in modo da verificare se, tenendo in considerazione anche le feature delle relazioni, sia possibile ottenere un aumento di performance.

Organizzazione della tesi

L'elaborato è suddiviso nei seguenti capitoli:

- **Capitolo 1** - presenta un quadro generale sulle GNN [4] e sulle modalità con la quale vengono scambiati i dati tra i nodi, analizzando le principali implementazioni e valutandone vantaggi e svantaggi.
- **Capitolo 2** - illustra i lavori di riferimento, nello specifico la soluzione reimplementata (EGAT) e quella alla quale sarà applicata (QA-GNN).
- **Capitolo 3** - si descrive nel dettaglio il design proposto e l'applicazione ad un caso di studio in ambito biomedico. Viene analizzata la fase di sperimentazione e i risultati ottenuti.
- **Capitolo 4** - vengono mostrati i passaggi più interessanti dell'implmenetazione proposta e i comandi necessari all'esecuzione.
- **Capitolo 5** - nell'ultimo capitolo si traggono le conclusioni e si descrivono i possibili sviluppi futuri.

Introduction

Context

Most of the real-world datasets come together with some form of graph structure: in social networks, users are represented as nodes and edges representing their relationships, or more generally, the World Wide Web.

Text, which is often presented in an unstructured form and thus more challenging to analyze, can be expressed using a graph. Such a graph can be a simple syntactic tree (figure 1), in which the lexical categories of each word emerge depending on its context, rather than a more complex structure that allows identifying entities present in the text and relationships among them, such as Abstract Meaning Representation (AMR) [1]. AMRs are labeled, directed, acyclic graphs that contain entire sentences and aim to abstract the syntactic representation of the sentence: this means that two sentences with the same meaning but not identical must be assigned to the same AMR (figure 2).

Using graphs to represent textual data offers many advantages: these structures are intuitive, flexible, and based on mathematical rules that provide a high degree of demonstrability. Moreover, compared to simple text, they maintain much more information, such as relationship and type of relationship between words, entity types, and cardinality that enrich and disambiguate the text, making it more suitable for many tasks.

Graph embedding is a technique used to transform nodes, edges, and features into vector space with a lower dimension while preserving graph structure and information properties. This family of algorithms can be divided into several subcategories depending on the output obtained, the input required and the logic applied to achieve the result. Some graph embedding procedures aim to obtain a representation for each node rather than generate a single embedding for the entire graph, which is less common as it does not scale well and has a more significant information loss. A second consideration concern the graph required as input: most algorithms accept homogeneous graphs, which are not directed, relationships have no type, and each node plays the same role.

Recently, this field of study has attracted the attention of many researchers, who initially developed Graph Neural Networks [4]. Kopf et al. [5] proposed a Graph Convolutional Network, then improved to take into account relationship between nodes [6] and finally enhanced through Velickovic et al. work "Graph Attention Networks" [7].

GAT combines the advantages of the Attention mechanism [8] and of the classical GCN [5]: before this architecture, during the aggregation, connected nodes were equally weighted; now, thanks to GAT, each node will automatically be able to assign a different relevance to each node in the neighborhood.

Problem

One of the main problems of existing networks is that they ignore relationship features, which often play a fundamental role in thoroughly understanding every semantic aspect of the text. The figure 3, for example, shows how through relation types, it is possible to classify nodes connected to "David Beckham" in three categories: "location", "work", and "family" and assign them a different relevance depending on the content under analysis.

Although recently many networks can take into account the type of relations [9][10] [11][12][13][2], often these solutions are not general-purpose, they do not take in consideration all the typical characteristics of a graph, they do not exploit the self-attention mechanism, and usually the implementation code is not available, resulting therefore unsuitable for our requirements.

One of the most exciting implementations is certainly EGAT [14], since, through a self-attention mechanism inspired by GAT [7], it generates a new representation for each node and relation, taking into account nodes of the neighborhood and relations between them, using simple transformations on the graph itself.

Contribution

This thesis is set in this context. Considering as a starting point the architecture of EGAT [14], which code has not been published yet, the goal is to reimplement and propose a variant of this solution, apply them to a second work, QA-GNN [15], replacing the GNN used, and verify whether taking into account edge features leads to a performance boost.

Thesis Organization

The thesis is divided in the following sections:

- **Chapter 1** - presents an overview of GNNs, how data is exchanged between nodes, analyzes the main implementations, and evaluates advantages and disadvantages.
- **Chapter 2** - illustrates the reference work, precisely the reimplemented solution (EGAT) and the one to which it will be applied (QA-GNN).
- **Chapter 3** - describes the proposed design and the application to a case study in the biomedical field. The experimental phase and the results obtained are analyzed.
- **Chapter 4** - shows the most exciting steps of the proposed implementation and the commands needed to execute them.
- **Chapter 5** - in the last chapter, conclusions are drawn and possible future developments are described.

Indice

Elenco delle figure

Capitolo 1

Background on Graph Neural Networks

This chapter describes the main GNNs developed over the years, their evolutions, and analyzes their limitations and strengths.

1.1 Overview

1.1.1 Motivation

Graphs are data structures that can model entities (nodes) and relationships between them (edges). Recently they have attracted a lot of attention because they have a great expressive power and are suitable to represent many real world datasets (knowledge graphs [16], social networks [17][18], protein-protein interaction [19], text [10][15], web [20] etc.).
Graphs are mainly used for tasks such as node classification, link prediction, and clustering.
Graph neural networks (GNNs) are neural models that operate on graph domain and capture graphs' structure via a message-passing system between nodes.

1.1.2 History

The first GNNs date back to the end of the 90s, when simple recursive neural networks began to be used on direct acyclic graphs to learn transductions from an input structured space to an output structured space [21].

Although the results of these early recursive networks were satisfactory at the time, many datasets are suitable to be represented by graphs with cycles, making the earlier architectures inappropriate in many real-world applications. To address this constraint, new variants of recursive neural networks [22] and feed-forward neural networks [23] were introduced respectively to map a graph and one of its nodes into an m-dimensional Euclidean space and realize adaptive contextual transductions for both classification and regression tasks. These networks were already able to process most common graphs, e.g., acyclic, cyclic, directed, and undirected. Despite their initial success, the basic idea behind these networks is to build state transition systems on graphs that can iterate until convergence is reached: this severely limits these models' extensibility and representation capability.

CNNs

The turning point in GNNs, as well as in Deep Learning in general, is represented by Convolutional Neural Networks (CNNs) [24], which opened the new era of deep learning [25]. CNNs are a class of neural networks that, since the first astounding results obtained in the object recognition task [26] [27], have sparked the interest of experts and have proven to be dominant in many areas in which Euclidean data (e.g., images and text) must be processed.

CNNs take inspiration from the human visual cortex; this means that the first layers are intended to extract simple and essential features from the input (such as lines and edges), which are then aggregated to extract higher-level information. CNNs' architecture is based on:

- **Local connections**: neurons are only locally connected to neurons of the previous level.

- **Shared Weights**: different neurons of the same level share weights: this means that they perform the same operation on different portions of the input.

These two features support spatial invariance, which aims to process images translated in any direction in the same way.
Three main types of layers are used to build a CNN:

- **Convolutional layers**: counting a set of learnable filters whose size is smaller than the input volume. These filters are scrolled through the input and used to apply the convolution operation necessary to extract features.

- **Pooling layers**: it performs a down-sampling of the volume by aggregating the information received in input according to a function defined a priori and not trainable.

- **Fully-connected layers**: a CNN includes a portion that consists of a simple fully-connected neural network which, after receiving in input the result of the convolutional block, performs the task of classification.

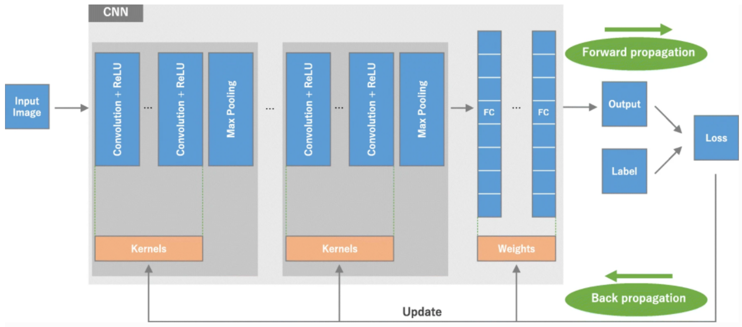

Figure 1.1: An overview of a CNN.
From: *Convolutional neural networks: an overview and application in radiology*, 2018 [24]

However, CNNs can only operate on regular Euclidean data like images and texts. These data structures can be regarded as instances of graphs. Therefore, it is straightforward to generalize CNNs on graphs.

Graph representation learning

The second impetus in the development of GNNs comes from graph representation learning, which aims to map nodes, edges, and subgraphs in the form of low-dimensional vectors [28]. Many approaches in this field rely on hand-engineered features and are limited by their inflexibility and cost. Over the years, many graph representation learning techniques have been developed and, depending on their characteristics, can be subdivided into many categories. We will follow the taxonomy proposed by Frisoni et al. [29] (figure 1.2) and only focus on node-level and graph-level embedding techniques:

- **Node-level embedding**: these approaches aim to produce an embedding for each node of the graph.
  - **Matrix factorization**: matrix factorization models are the historically oldest and most studied theoretically. They represent graphs as a large matrix and try to approximate it with different low-rank matrix factorizations (e.g., SVG [30]).
    Although these models are mathematically interpretable and transparent, they generally only take adjacent nodes into account when

generating the embedding and have high costs in terms of time and memory.

– **Deep Learning with Random Walks**: these techniques are particularly suitable to perform node-embedding tasks on large graphs, as they avoid processing the entire structure, finding the right compromise between accuracy and efficiency.
The firstborn model belonging to this category is DeepWalk [31], based on the idea of generalizing NLP skip-gram models to graphs, extracting from them paths interpreted as sentences in which each node corresponds to a word. After generating different paths, DeepWalk trains a neural network trying to maximize the probability of predicting the node's context (the structure of neighboring nodes), taking into account the embedding of the node itself and the co-occurrences in the neighborhood. This approach is similar to Word Embedding generation.
Numerous studies take inspiration from this first model (e.g., LINE [32], node2vec [33], HARP [34]). Although they are unsupervised models that can take into account second-order neighbors and capture long-distance relationships, they are limited because they do not evaluate global graph information.

– **Deep Learning without Random Walks**: the idea behind these techniques is to apply Deep Leaning models to entire graphs.
A typical approach is to use autoencoders that, through an encoder, try to encode the input generating a compressed representation by aggregating local information and, through a decoder, try to reconstruct the original input. The quality of the model increases as it increases the similarity between the original input and the one generated by the decoder.
An example of a model belonging to this family is VGAE [35].
GNNs are also within this category of techniques. They typically require the graph's adjacency matrix as input, and, although they often perform worse than autoencoder-based models, they do not have to sequence the graph (turn it into a sequence of tokens which is the standard input of encoders). therefore they can retain spatial information. GNNs do not aim to generate node embeddings, but their goal is to optimize graph-related tasks (nodes or graph classification, link prediction between nodes). For this reason, in general, they are highly supervised and need labeled input data. The embedded representation is built internally to achieve the objective imposed by the task. For this reason, it is strictly dependent on the task, on

the data provided in input and consequently not general-purpose. Such models compute node embeddings through a message exchange process which allows data propagation taking adjacent nodes into account.
Two of the most famous works are *Graph Convolutional Networks* (GCN) [5], and *GAT* [7]. The first one replaces message-passing with graph convolution. At the same time, GAT exploits a self-attention mechanism [8] that is automatically able to assign a different relevance to each node in the neighborhood.

- **Graph-level embedding**: these approaches aim to produce a single embedding for the whole graph.

  - **Supervised learning**: the network needs to learn how to perform aggregation, and a parameterized pooling layer is usually used for this purpose (e.g., DiffPool [36], SortPool [37], TopKPool [38]. Other more basic techniques perform a simple sum or average of all nodes in an additional final layer [39].
  - **Unsupervised learning**: although there are numerous works belonging to this family, a pooling operation is usually performed, similar to an MLP. Just as in an MLP, representations of individual tokens (words or portions thereof) are learned and then joined; similarly, these models generate and merge embeddings of individual nodes. To mirror the classical MLP technique of adding a CLS token that aggregates the embeddings of all the words in the sentence, a *supernode* connected to every other node can also be added in a graph-based approach [40].
    There are also more straightforward techniques that perform flat pooling (e.g., average, maximum, sum) or hierarchical pooling, which recursively applies a chain of flat pooling techniques after partitioning the graph into subgraphs until embedding individual nodes.
  - **Statistical representations**: this last category of graph-level embedding techniques represents graphs with hand-engineered feature vectors, after defining a priori the number of dimensions [41] [42]. They can consider local characteristics and statistical or topological properties (e.g., number of nodes, edges, nodes average, nodes max, etc.).

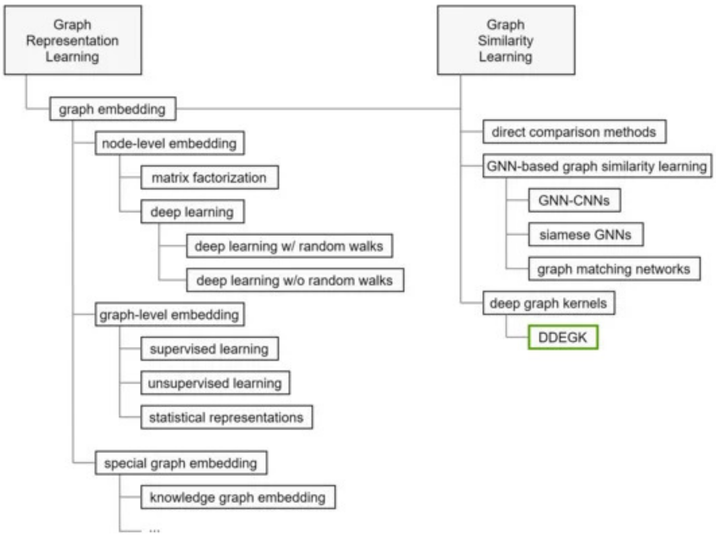

Figure 1.2: Graph Representation Learning taxonomy.
From: *Unsupervised Event Graph Representation and Similarity Learning on Biomedical Literature*, 2022 [29]

1.2 Main models

1.2.1 GCN

In section 1.1.2 CNNs were analyzed, in which the operation of convolution is applied, and neurons of a given layer are multiplied with a set of weights called filters or kernels. These filters act as a sliding window across the image to allow each pixel to aggregate information from its neighbors. In addition, by sliding the same filter across the entire image, different portions are processed in the same way (shared weights).

The convolution operation performed in GCNs is the same but on graphs instead of images. The model learns node features, considering the neighborhood, making the network a generalization of CNNs suitable not only on regular Euclidean structured data.
The original idea behind GCN was inspired by wave propagation, implemented by leveraging the Eigen-decomposition of graph Laplacian matrix, necessary to understand the graph structure. This process is similar to the idea behind PCA [43] and LDA [44], in which such decomposition is used to reduce dimensionality.
GNNs take into account the adjacency matrix (A), as well as a matrix containing the features of the graph nodes (input features).
A is a matrix expressing connections between nodes: cell $A[i, j]$ has a value of

1 if nodes i and j are connected, 0 otherwise. The typical equation used during the forward step is the following:

$$H^{[i+1]} = \sigma(W^{[i]}H^{[i]} + b^{[i]}) \tag{1.1}$$

To the standard equation is then added the adjacency matrix A:

$$H^{[i+1]} = \sigma(W^{[i]}H^{[i]}A^* + b^{[i]}) \tag{1.2}$$

$W^{[i]}$ contains the weights of the previous layer (i), $H^{[i]}$ contains the feature representation of the previous layer (i) and $b^{[i]}$ is the bias.

Thomas N. Kipf and Max Welling [45] refer to matrix A^* as renormalization trick which is obtained by the following procedure:

1. A self loop is added to each node, which means setting each value on the main diagonal of A to 1.
   The dot product between A and the feature matrix X represent the sum of neighbors, self loops are added to take into account the feature of the node itself.

2. The dot product between the matrix A just obtained and the feature matrix X is computed to produce AX.

3. A symmetric normalization us applied to AX:

$$A^* = D^{-1/2}AXD^{-1/2} \tag{1.3}$$

D, named degree matrix, contains information about the degree of each node, which is the number of edges attached to each vertex.
This normalization prevents numerical instabilities and vanishing/exploding gradients and simplifies model convergence. Also, the lower a node degree, the stronger it will belong to a cluster and so taken into account.

Once this flexible model for propagating information within graphs is defined, it can be used within arbitrarily complex networks, as shown in image 1.3.
Through this message-passing system, each layer will update node features which will then be forwarded to the next layer after applying an activation function (e.g., ReLU).
An exciting feature of GCNs is that they can learn features representation even before the training process.

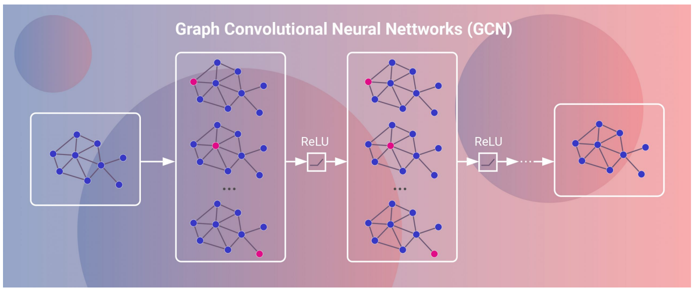

Figure 1.3: A simplified representation of a GCN architecture.
From: *Gentle Introduction to Graph Neural Networks and Graph Convolutional Networks* [46]

1.2.2 RGCN

RGC [6] was one of the first models to show how a classical GCN could be applied to relational data, specifically for link prediction and entity classification tasks. The authors also introduced parameter sharing techniques to ensure a greater degree of scalability for the model to be used even on large graphs.

Given a directed and labeled graph $G = (V, \mathcal{E}, R)$ with nodes $v_i \epsilon V$ and labeled edges $(v_i, r, v_j) \epsilon \mathcal{E}$, where $r \epsilon R$ is a relation type, the typical message-passing framework, used to perform the convolution, can be expressed as:

$$h_i^{l+1} = \sigma(\sum_{m \epsilon M_i} g_m(h_i^l, h_j^l)) \tag{1.4}$$

In this equation, h_i^l is the hidden state of node v_i in the l-th layer of the network and d^l is the dimensionality of the representation.
Messages of the form $g_m(.,.)$ are accumulated and passed through an element-wise activation function σ, for example a ReLU. M_i is the set of incoming messages for node v_i and often corresponds to the set of incoming edges. $g_m(.,.)$ can be a message-specific neural network or simply a linear transformation. Schlichtkrull et al. [6] defined the following propagation model to update the node embeddings in a relational and directed graph:

$$h_i^{l+1} = \sigma(\sum_{r \epsilon R} \sum_{j \epsilon N_i^r} \frac{1}{c_{i,r}} W_r^l h_j^l + W_0^l h_i^l) \tag{1.5}$$

N_i^r denotes the set of neighbor indices of node i under relation $r \epsilon R$, $c_{i,r}$ is a problem–specific normalization constant, that can be learned or chosen in advance.

Equation 1.5 aggregates embeddings of the neighboring nodes through a normalized sum. Different from regular GCNs, a relation-specific transformation

is introduced (i.e. depending on the type and the direction of an edge).
In addition, a self-loop with a special relation type is added to each node. In this way, during the aggregation of each node, the node itself is also taken into account.
A neural network layer update consists of evaluating (1.5) in parallel for every node in the graph, as shown in figure 1.4.
The overall RGCN model has the following form: L RGCN are stacked and the output of the previous layer becomes the input of the next layer. The input of the first layer can be chosen as a unique one-hot vector for each node in the graph.

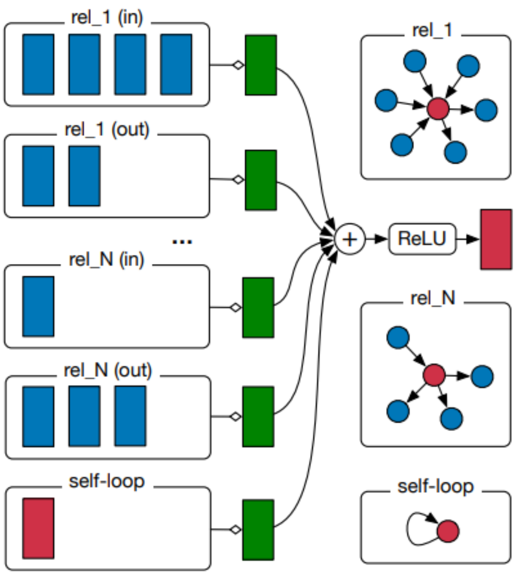

Figure 1.4: 2: Diagram for computing the update of a single graph node/entity (red) in the R-GCN model. Embeddings from neighboring nodes (dark blue) are gathered and then transformed for each relation type individually. The resulting representation (green) is aggregated via a normalized sum and passed through a ReLU activation function.
From: *Modeling Relational Data with Graph Convolutional Networks*, 2017 [6]

Regularization

The main problem of this message-passing framework is its application to highly and multi-relation data, because off the rapid growth in number of parameters depending on the number of relation types in the graph. This can also cause overfitting on rare relations.

To solve these issues, authors introduced two separate methods for weight regularization: *basis* and *block-diagonal* decomposition.
With the basis decomposition, each W_r^l is defined as:

$$W_r^l = \sum_{b=1}^{B} a_{rb}^l V_b^l \tag{1.6}$$

Equation 1.6 can be seen as a linear combination of basis transformation $V_b^l \epsilon \mathbb{R}^{d^{l+1} x d^l}$ with coefficients a_{rb}^l such that only the coefficients depend on the relation type r.

In the block diagonal decomposition, each W_r^l is defined through the direct sum over a set of low-dimensional matrices:

$$W_r^l = \bigoplus_{b=1}^{B} Q_{br}^l \tag{1.7}$$

W_r^l are block diagonal matrices: $diag(Q_{1r}^l, ..., Q_{br}^l)$ with $Q_{br}^l \epsilon \mathbb{R}^{(d^{l+1}/B)x(d^l/B)}$.

The basis function decomposition 1.6 is a sort of weight sharing between different relation types, while block decomposition 1.7 can be seen as a sparsity constraint on the weight matrices for each relation type.
Both decompositions reduce the number of parameters needed on high and multi-relational data and can also reduce the overfitting problem on rare relations.

1.2.3 GAT

Graph Attention Networks (GATs) are architectures that operate on graphs, exploiting masked self-attentional layers [8] succeeding in improving and solving the main limitations of the previous methods based on graph convolution.

The main idea is the same that led to the birth of GCNs, namely the desire to aggregate information between neighboring nodes. In addition to that, it would be good if graph convolution layers guaranteed the following properties:

- Computational and storage efficiency (max $\mathcal{O}(V + E)$)
- Number of parameters independent of the input graph
- Taking into account the neighborhood of a node
- Ability to assign different importance to each neighbor

Consider a graph G with n nodes, defined by a set of node features $(\overrightarrow{h_1}, \overrightarrow{h_2}, ..., \overrightarrow{h_n})$ and its adjacency matrix A, which expresses connections between nodes: $A_{i,j} = 1$ if nodes i and j are connected, 0 otherwise.

Typical graph convolutional layer

A typical graph convolutional layer computes a set of new node features $(\overrightarrow{h_1}', \overrightarrow{h_2}', ..., \overrightarrow{h_n}')$ taking into account node features and graph structure.

The first operation performed in each graph convolutional layer is a shared transformation between all nodes, performed using a shared weight matrix W.

$$\overrightarrow{g_i} = W\overrightarrow{h_i} \tag{1.8}$$

This transformation is necessary to obtain a higher-level representation and the new representations are then used to perform the convolutional operation.

The convolutional operator can be defined as a weighted sum of all the neighbors of the node.
Given N_i the set of neighbors of node i, including i itself, the new features can be defined as:

$$\overrightarrow{h_i} = \sigma(\sum_{j \epsilon N_i} \alpha_{ij}\overrightarrow{g_j}) \tag{1.9}$$

σ is an activation function, g_j is the new representation of the node features, and a_{ij} is a weight factor that allows giving different relevance to each neighbor of the node i.
All previous GNN-based approaches defined this parameter explicitly (e.g., by a trainable weight matrix or by considering the graph's structure) thus preventing satisfying one of the desired properties defined above.

Velickovic et al. [7] idea is to let the coefficient a_{ij} be defined implicitly by using a self-attention [8] mechanism over the node features. The reason behind this decision was that the self-attention mechanism had already proven to be critical achieving state-of-the-art performance in numerous machine translation tasks.

To calculate the coefficient a_{ij}, first the non-normalized coefficients e_{ij} are computed for each pair of nodes i and j taking into account their features.

$$e_{ij} = a(\overrightarrow{h_i}, \overrightarrow{h_j}) \tag{1.10}$$

a is the attentional mechanism and can be implemented in many different ways. In the original paper, Velickovic et al. decided to use a straightforward single-layer neural network, whose parameters are trained with the rest of the network.
The graph structure is injected by exploiting the set of neighboring nodes N_i,

forcing the node to pay attention only to the nodes to which it is directly connected.
In addition, coefficients are normalized using a softmax function to ensure more excellent stability and a more straightforward comparison between coefficients belonging to different nodes.

$$a_{ij} = \frac{exp(e_{ij})}{\sum_{k\epsilon N_i} exp(e_{ik})} \tag{1.11}$$

To further regularize the learning process, Velickovic et al. decided to use multi-head-attention, introduced by Vaswani et al. [8].
Namely, the layer just described is repeated k times. Each replica is independent and has its parameters.
The individual layers' outputs are then aggregated, usually by an average or concatenation operation.

$$\overrightarrow{h_i}' = \|_{k=1}^{K} \sigma(\sum_{j\epsilon N_i} a_{ij}^k W^k \overrightarrow{h_j}) \tag{1.12}$$

a_{ij}^k are the attention coefficients computed by the k-th head and W^k is the weight matrix used for the initial linear transformation by the k-th replica.

Finally, it has been demonstrated how, by applying the dropout technique [47] during the computation of a_{ij} coefficients, it is possible to regularize the training process, especially for small training sets. In this way, during the training, the neighborhood of each node is sampled stochastically.

The layer defined satisfies all of the desired properties defined before:

- The computation of attentional coefficients and their aggregation can be parallelized, making the layer computationally efficient.
- The layer can be implemented using a sparse matrix, requiring no more than $\mathcal{O}(V + E))$ entries.
- The number of parameter is fixed and independent of the input graph.
- Node's neighbors are taken into account, and to each one is assigned a different relevance

1.2.4 GATv2

Shaked Brody et al. with their work "How Attentive are Graph Attention Networks?" [49] showed that GAT computes a limited kind of attention. Namely, the ranking of the attention scores is unconditioned on the query node. They define this kind of attention as *static* which does not allow GAT to handle

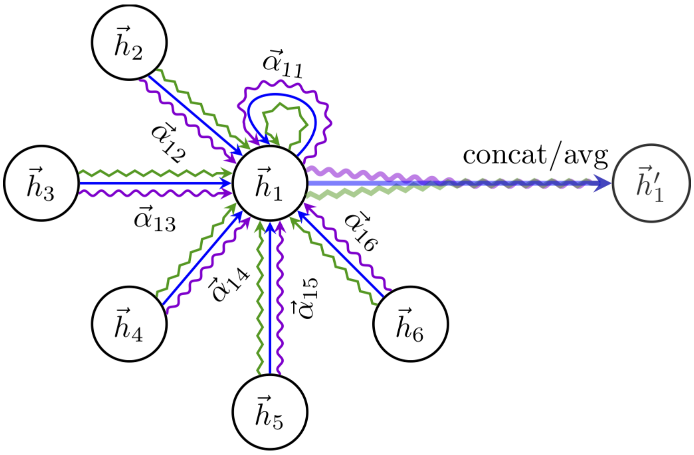

Figure 1.5: A multi-head GAT layer. Every neighbor of the central node sends it's own vector of attentional coefficients, one per each head. These are used to compute k different new node features which are then aggregated, in the example by a concatenation or average operation.
From: *https://petar-v.com/GAT/* [48]

some graph problems.
Another way to think about attention is the ability to attend to the most relevant inputs, given a specific query. This is possible only by decaying other inputs by giving them lower scores than others. If one key, as with GAT, is always given an attention score greater or equal than other keys, queries can not ignore this key and apply the score decay.

Static attention: a family of scoring functions $\mathcal{F}$ computes static scoring for a given set of key vectors $K = \{k_1, ..., k_n\}$ and query vectors $Q = \{q_1, ..., q_m\}$ if, for every function $f \epsilon \mathcal{F}$, there is a "highest scoring" key j_f such that for every query i: $f(q_i, k_{jf}) \geq f(q_i, k_j)$.
This attention is minimal because every function $f \epsilon \mathcal{F}$ a key is always selected regardless of the query, as shown in figure 1.6. These solutions can not work correctly when keys have different relevance depending on the query.

Dynamic attention: a family of scoring functions $\mathcal{F}$ computes dynamic scoring for a given set of key vectors $K = \{k_1, ..., k_n\}$ and query vectors $Q = \{q_1, ..., q_m\}$ if, for any mapping $\varphi : [m] \to [n]$, exists a function $f \epsilon \mathcal{F}$ such that for any query i and and any key k $j_{\neq \varphi(i)}$: $f(q_i, k_{\varphi(i)}) \geq f(q_i, k_j)$.
In other words, the dynamic attention can chose every key $\varphi(i)$ given the query i, by making $f(q_i, k_{\varphi(i)})$ the maximal in $\{f(q_i, k_j) | j \epsilon [n]\}$.

Although the equation for calculating the non-normalized attention coefficients e_{ij} (1.10) can be implemented in different ways, the one proposed by Velickovic et al. has become the de facto standard and is now used in many ap-

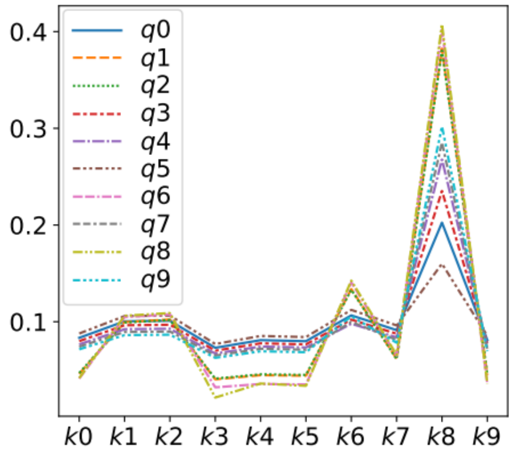

Figure 1.6: Static attention computed on a complete bipartite graph. The ranking of attention scores is the same for all nodes in the graph and is independent on the query node. In the example, all queries (q_0 to q_9) attend mostly on the key k_8.

From: *How Attentive are Graph Attention Networks?*, 2021 [49]

plications belonging to different domains, especially in all GAT implementation of the main libraries.

Given the attention mechanism a, a weight matrix W, the previous equations can be rewritten as:

$$e(h_i, h_j) = LeakyReLU(a^T[Wh_i \parallel Wh_j]) \tag{1.13}$$

$$a_{i,j} = softmax_j(e(h_i, h_j)) = \frac{exp(e(h_i, h_j))}{\sum_{j' \epsilon N_i} exp(e(h_i, h_j'))} \tag{1.14}$$

$$h_i' = \sigma(\sum_{j \epsilon N_i} a_{ij} W h_j) \tag{1.15}$$

The learned parameter a can be rewritten as a concatenation between a_1 and a_2, so equation 1.13 becomes:

$$e(h_i, h_j) = LeakyReLU(a_1^T W h_i + a_2^T W h_j) \tag{1.16}$$

In the original GAT, exists a node j_{max} such that $a_2^T W h_{j_{max}}$ is maximal along all nodes j. Due to the monotonicity of LeakyReLU and the softmax function, for every query node i, the node j_{max} also leads to the maximal value of its attention. For this reason, the classical GAT computes static attention.

The main problem in GAT scoring function 1.13 is that W and a are applied consecutively and can be collapsed in a single layer.

GATv2 solves this problem and obtains a much higher expressive power just by modifying the order of internal operations:

$$GAT: \qquad e(h_i, h_j) = LeakyReLU(a^T[Wh_i \parallel Wh_j]) \tag{1.17}$$

$$GATv2: \qquad e(h_i, h_j) = a^T LeakyReLU(W[h_i \parallel h_j]) \tag{1.18}$$

GATv2 has the same time complexity as GAT, but by merging linear layers, GAT can be computed faster.

The authors demonstrated the weakness of GAT on a simple task on a bipartite graph 1.6. They also showed that GATv2 is more robust to edge noise because dynamic attention allows decaying noisy edges.

Finally, they compared GAT and GATv2 on 12 benchmarks, and they found that GAT is inferior to GATv2 across all of them.

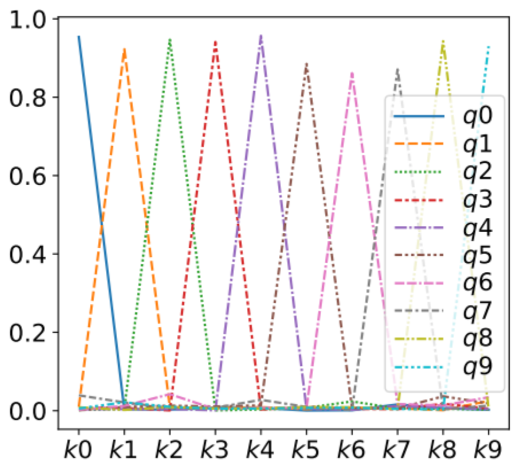

Figure 1.7: Dynamic attention computed on the same complete bipartite graph 1.6. The ranking of attention scores now is not the same for all nodes in the graph and depends on the query node.

From: *How Attentive are Graph Attention Networks?*, 2021 [49]

Capitolo 2

Reference Architectures

This chapter describes the two main reference architectures for this thesis, namely EGAT [14], the implemented GNN, and QA-GNN [15], used in our case study.

2.1 Edge-Featured Graph Attention Network (EGAT)

This section describes the EGAT [14] solution, which is the starting point for our implementation.

Our decision fell on EGAT because it was the one with the view of GAT applied to labeled graphs most similar to ours.
The idea is to create a version of GAT in which the aggregation operation considers node and edge features. Each of these elements can be taken into account and weighted differently through the attention mechanism.

2.1.1 Introduction

Wang et al. [14] noticed that most existing architectures do not consider edges features, which in many real-world node classification tasks play a fundamental role. As the authors themselves point out, for example, in a trading network, the node labels may be relevant to the transactions. In such a case, the information contained in edges may have a more significant contribution to the classification accuracy than node features.

Some approaches use a priori defined static aggregation functions to integrate edge features. These solutions perform well on some specific graphs and tasks, but are not general-purpose solutions suitable for any need. Other works manage to process the information contained in the relations, but always with limitations.
For example, R-GCN proposed by Schlichtkrull et al. [50] accepts only discrete

features, not allowing to work with continuous attributes.
Gong et al. [51] have developed a framework that accepts continuous features for edges but uses them as simple weights between pairs of nodes.
On top of that, each graph has different preferences for node and edge features and should be able to learn how to manage them automatically.

EGAT addresses all these challenges and can be considered an extension of GAT. The original attention mechanism is enhanced so that edges information can be exploited during the computation of the attention coefficients.
The authors had to redefine the attention mechanism, data structures, and the entire process used in traditional GAT.
Edges features are also updated like those of the nodes as iterations proceed, allowing the model to maintain consistency between edges and nodes.
EGAT accepts as input graphs with discrete or continuous features for both edges and nodes.

2.1.2 EGAT model

EGAT layer overview

A single EGAT layer is composed of two different blocks: a node attention block and an edge attention block, as shown in figure 2.1. The proposed design is symmetrical: both nodes and edges can update their features in a parallel and equivalent way.

Each EGAT layer accepts as inputs a set of node features $H = \{\overrightarrow{h_1}, \overrightarrow{h_2} ..., \overrightarrow{h_N}\}$, $\overrightarrow{h_i} \epsilon \mathbb{R}^{F_H}$ and a set of edge features $E = \{\overrightarrow{e_1}, \overrightarrow{e_2} ..., \overrightarrow{e_M}\}$, $\overrightarrow{e_i} \epsilon \mathbb{R}^{F_E}$.
N and M represent the number of nodes and edges, respectively, while F_E and F_H represent the number of their respective features.
At the end of the process, the layer will produce high-level outputs, namely a new set of node feature $H^{'} = \{\overrightarrow{h_1^{'}}, \overrightarrow{h_2^{'}} ..., \overrightarrow{h_N^{'}}\}$, $\overrightarrow{h_i^{'}} \epsilon \mathbb{R}^{F_H^{'}}$ and a new set of edge features $E^{'} = \{\overrightarrow{e_1^{'}}, \overrightarrow{e_2^{'}} ..., \overrightarrow{e_N^{'}}\}$, $\overrightarrow{e_i^{'}} \epsilon \mathbb{R}^{F_E^{'}}$.

Moreover, two extended adjacency matrices of nodes (M_H) and edges (M_E) are injected into the two blocks. Compared to a classic adjacency matrix, they expand a third dimension to indicate which edge connects every pair of nodes. M_H has a size of $[NxNxM]$, while M_E has a size of $[MxMxN]$. Each cell $M_{E_{x,y,z}}$ is set to 1 if nodes x and y are connected via edge z, 0 otherwise, while $M_{H_{x,y,z}}$ is set to 1 if node z is placed between edges x and y, 0 otherwise.
The original implementation also requires an extra matrix for each attention block, called mapping matrices for nodes and edges, used to transform both adjacency matrices and make it more intuitive to find relationships between nodes and edges. In the implemented solution, it has been decided to omit

them because we do not use sparse matrices, making this step useless and even harmful in computational terms.

The cardinality of F and $F^{'}$ can be different (for both F_H and F_E since an independent linear transformation is applied on both edges and nodes. For this transformation, we use two learnable matrices, $W_H \epsilon \mathbb{R}^{F_H x F^{'}_H}$ and $W_E \epsilon \mathbb{R}^{F_E x F^{'}_E}$.

For each node i and edge p, their transformed features are computed as:

$$\overrightarrow{h_i^*} = W_H \overrightarrow{h_i} \tag{2.1}$$

$$\overrightarrow{e_p^*} = W_E \overrightarrow{e_p} \tag{2.2}$$

After this initial transformation, the results are fed into both node and edge attention blocks to produce the new set of edge and node features.

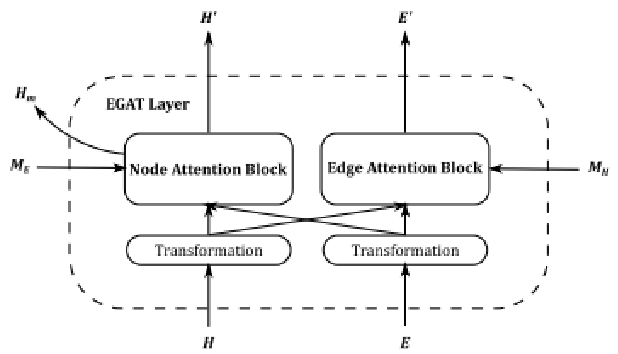

Figure 2.1: Structure of EGAT layer. It accepts node features H and edge features E as inputs and produces two sets of new features, $H^{'}$ and $E^{'}$. Each attention block is also fed with its extended adjacency matrix M_H and M_E. Unlike the original solution, edge mapping and node matrices are not provided as input to each attention block.

From: *Edge-Featured Graph Attention Network*, 2021 [14]

Node attention block

The node attention block accepts a set of node features H, a set of edge features E, the adjacency matrix M_H and produces a new set of node features $H^{'}$ passed to the next EGAT layer, and a set of node features H_m used in the last level merge layer to achieve a multi-scale concatenation. The adjacency matrix is unique for each graph and thus can be computed in a pre-processing step before the training process.

From now on, we will refer to the node features after the linear transformation as h_i.

The model can easily find the edge connecting two nodes thanks to the adjacency matrix. Based on that, an enhanced attention mechanism that takes into account not only node features but also the features of the edge connecting them can be performed on each node to produce the attention coefficients.
For each node i, the weight w_{ij} is computed for every node $j \epsilon N_i$, where N_i is the set of the first-order neighbors of the node i, including i itself.
The logic behind this process is pretty much similar to the one used in normal GAT:

- Features of node i (h_i) and the neighbor j (h_j) are concatenated (2.3).
- To the previous result, features of the edge connecting node i and j ((e_{ij}))are also concatenated (2.4).
- Concatenated features are then parametrized by a weight vector ($\overrightarrow{a}^T$) (2.5).
- A LeakyReLU function is applied as the activation function (2.6).
- At the end, the result in normalized across node i neighborhood, by using a softmax function. The whole process can be formulated as shown inf equation 2.7.

$$\overrightarrow{h}_{ij} = [\overrightarrow{h}_i \parallel \overrightarrow{h}_j] \tag{2.3}$$

$$\overrightarrow{eh}_{ij} = [\overrightarrow{h}_{ij} \parallel \overrightarrow{e}_{ij}] \tag{2.4}$$

$$\overrightarrow{a}_{ij} = \overrightarrow{a}^T \overrightarrow{eh}_{ij} \tag{2.5}$$

$$a_{ij} = LeakyReLU(\overrightarrow{aeh}_{ij}) \tag{2.6}$$

$$a_{ij} = \frac{exp(LeakyReLU(\overrightarrow{a}^T[\overrightarrow{h}_i \parallel \overrightarrow{h}_j \parallel \overrightarrow{e}_{ij}]))}{\sum_{k \epsilon N_i} exp(LeakyReLU(\overrightarrow{a}^T[\overrightarrow{h}_i \parallel \overrightarrow{h}_k \parallel \overrightarrow{e}_{ik}]))} \tag{2.7}$$

As shown in equation 2.7, the aggregated features should also include the features of the node itself. Without edge features, this can be done by adding an identity matrix to the adjacency one. We use a tricky method by adding a virtual self-loop to each node that does not have an edge that connects itself.

Features of this edge are initialized as an average on each dimension of adjacent edges' features.

After producing the normalized attention coefficients, for each neighborhood, new node features are computed by performing a weighted sum for each neighbor node 2.8.

$$\overrightarrow{h}'_i = \sigma(\sum_{j \epsilon N_i} a_{ij} \overrightarrow{h}_j) \tag{2.8}$$

Notice that only node features are taken into account to generate new node features. Edge features are used only during attention coefficient computation. In fact, by merging edge and node features at each iteration, the model would be too complex, and the features could become unnecessarily complicated.
In order to also consider edge features during the aggregation and obtain a multi-scale concatenation in the last-level merge layer, a set of edge-integrated node features H_m is also produced, generated as follows:

$$\overrightarrow{m}_i = \sigma(\sum_{j \epsilon N_i} a_{ij}(\overrightarrow{h}_j) \parallel \overrightarrow{e}_{ij}) \tag{2.9}$$

The process is the same as the standard node features, but features of the edge connecting each node pair are concatenated to the neighbor node features. However, this set of edge-integrated node features is never passed to the following EGAT layer as the inputs, for the reasons mentioned before.

Edge attention block

Node features are updated in the node attention block to acquire high-level features, so it is unreasonable to reuse the original low-level edge features during the weight computation. Besides, by obtaining high-level edge features, we can keep a balance of importance between nodes and edges.
For this reason, each EGAT layer contains also an edge attention block, which accepts a set of node features H, a set of edge features E, the adjacency matrix M_E and produces a new set of edge features $E^{'}$ passed to the next EGAT layer. The adjacency matrix is unique for each graph and thus can be computed in a pre-processing step before the training process.

A natural idea to implement this block is to update edge features considering adjacent edges by applying a classical GAT. Two edges are considered adjacent if they have at least one common vertex.
This aggregation is implemented by adopting a tricky approach that switches the roles of nodes and edges in the graph, an approach similar to the one proposed by Chen et al. [52] for community detection.

A new graph is created based on the original one, whose nodes become edges and edges become nodes. The transformation is shown in figure 2.2.
In the new graph, two new nodes (original edges) are connected by an edge (original node) represented by the vertex in common between the two edges in the original graph.
Consequently, the original node adjacency matrix (M_H) will also be transformed into the edge adjacency matrix (M_E) to reflect the structure of the new graph. $M_{E_{xyz}}$ is set to 1 if, in the new graph, nodes x and y are connected via edge z. This means that in the original graph edges x and y had vertex z in common. An example of the new adjacency matrix of the graph of figure 2.2 is shown if figure 2.3.

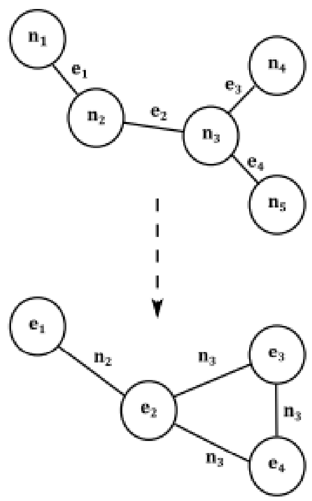

Figure 2.2: An example of graph transformation. Nodes become edges and edges become nodes.
From: *Edge-Featured Graph Attention Network*, 2021 [14]

For each original edge p, the normalized attention weight in relation to edge q can be expressed as:

$$\beta_{pq} = \frac{exp(LeakyReLU(\overrightarrow{b}^T[\overrightarrow{e}_p] \parallel \overrightarrow{e}_q \parallel \overrightarrow{h}_{pq}))}{\sum_{k \epsilon Np} exp(LeakyReLU(\overrightarrow{b}^T[\overrightarrow{e}_p] \parallel \overrightarrow{e}_k \parallel \overrightarrow{h}_{pk}))} \tag{2.10}$$

N_p is the first-order neighbor set of edge p and p itself, $\overrightarrow{b}$ is a weight vector of size $\mathbb{R}^{2F'_E + F'_H}$.
The equation is the same of the one used in the node attention block, with switched roles for nodes and edges.
After producing the normalized attention coefficients, for each neighborhood,

	1	2	3	4
1		2		
2	2		3	3
3		3		3
4		3	3	

Figure 2.3: Edge adjacency matrix of graph 2.2.For simplicity, the one-hot-vector contained in each cell is replaced with the corresponding non-zero index.
From: *Edge-Featured Graph Attention Network*, 2021 [14]

new edge features are computed by performing a weighted sum for each neighbor edge 2.11.

$$\overrightarrow{e}'_p = \sigma(\sum_{q \epsilon N_p} \beta_{pq} \overrightarrow{e}_q) \tag{2.11}$$

Note that there is no middle node between the two edges when the attention weight of an arbitrary edge and the edge itself is computed. As is the case within the node attention block, a dummy node is then created whose features are initialized, in the original solution, with zeros.

EGAT architecture

The overall architecture of an EGAT model is composed by stacking various EGAT layers and joining a final merge layer at the end, as shown in figure 2.4.

As described in section 2.1.2, each EGAT layer not only produces new node features used as input for the next layer but also computes edge-integrated node features (H_m) used in the merge layer.
More specifically, the results of the individual layers are aggregated by concatenating them.
Furthermore, taking inspiration from the GAT [7] architecture, EGAT exploits a multi-head attention mechanism. K different EGAT models (named heads) are instantiated, and each of them produces its edge-integrated node features, then aggregated in the final merge layer.
In the end, the individual results are merged through another concatenation operation. The main difference with GAT is that the final merge is performed on the union of the outputs of all EGAT layers rather than on a single layer.
The final features are computed as follows:

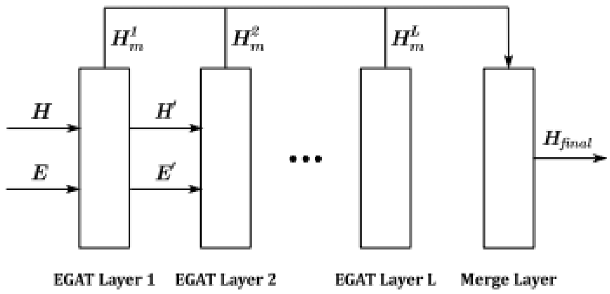

Figure 2.4: Overall EGAt architecture. Many EGAT layers are stacked. Each one receives as input the output of the previous layer. At the end of the architecture, a merge layer merges edge-integrated node features produced by each layer.

From: *Edge-Featured Graph Attention Network*, 2021 [14]

$$\overrightarrow{h}_i^* = \|_{k=1}^{K}(\|_{l=1}^{L} m_i^{l,k}) \tag{2.12}$$

L indicates the number of EGAT layers in each head, $m_i^{l,k}$ represents the edge-integrated node features of node i produced by the l-th layer of the k-th head.

2.1.3 Results obtained by the original model

The original model was tested on five node classification task containing both node-sensitive and edge-sensitive datasets. The former are graphs where node features are highly correlated with node labels, while the latter are graphs where edges are more relevant than nodes.
The model was compared against many approaches, including GAT.

For the node-sensitive tasks, a convolutional operation was performed in the merge layer followed by a softmax function to predict the probability of each class.
Results showed that EGAT is highly competitive against state-of-art solutions, achieving superior performance on one dataset and slightly lower performance on the remaining two, probably due to the dummy edge features introduced and initialized with the number of adjacent edges.

For the edge-sensitive task, to ensure fairness, the authors created three variants of GAT to aggregate edge features into node features by simply performing sum, average or max pooling.
EGAT shows an incredible performance, which is ahead of other approaches on both datasets.

2.2 QA-GNN

Question answering systems need much information in order to carry out the necessary reasoning and achieve the task in the matter. Usually, this knowledge is represented through language models (LMs) or through knowledge graphs (KGs), where entities are represented as nodes and relations between them are the edges that connect them.
Several recent works have obtained excellent results using language models, which generally do not perform well on structured reasoning, such as handling of negations. This situation is more suitable for knowledge graphs, which often introduce noise and lack of coverage.
The solution proposed by Yasunaga et al. [15] aims to combine the use of both data sources, but this presents two challenges. Given the QA context (question and answer choices) the model needs to:

- Identify the knowledge from a large KG
- Merge the KG with the QA context to perform joint reasoning

Some existing models take a portion of the KG considering the entities of QA context and their few-hop neighbors but introduce many irrelevant entities and consequently noise.
Other methods consider the QA context and the KG separately, applying the LM individually on the QA and a GNN on the KG, limiting the capacity and power of the model.

The solution proposed by Yasunaga et al. [15] is an end-to-end LM+KG model that addresses the two challenges listed above through the following technique.
The QA context is encoded using an LM, and then a KG subgraph is retrieved, considering the few-hop neighbors of the entities contained in the QA context (named topic entities). Since each entity has a different relevance considering the QA context, relevance scoring is calculated for each node of the graph: each KG entity is concatenated to the QA context. Its similarity is then calculated using a pre-trained LM.
Subsequently, a joint graph is generated, in which the QA context is explicitly represented as an additional node connected to the topic entities of the KG. Finally, a new attention-based GNN is applied, taking into account relevance scores, node types, nodes, and relationships between them, updating both the KG entities and the QA context node to leverage the gap between the two sources of information.

2.2.1 QA-GNN architecture

Given a knowledge graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $\mathcal{V}$ is the set of entity nodes in the KG, $\mathcal{E} \subseteq \mathcal{V}\mathcal{R}\mathcal{V}$ is the set of edges connecting entities, and $\mathcal{R}$ represents a set of relation types.
Given a question q and an answer choice a, each entity mentioned in the question or the answer is linked to the given KG. $\mathcal{V}_q$ is the set of entities mentioned in the question (blue entities in figure 2.5), $\mathcal{V}_a$ is the set of entities mentioned in the answer (red entities in figure 2.5) and $\mathcal{V}_{q,a} = \mathcal{V}_q \cup \mathcal{V}_a$ is the set of all entities that appear in either the question or the answer choice and is called *topic entities*.
For each question-answer pair, a subgraph $\mathcal{G}^{q,a}_{sub} = (\mathcal{V}^{q,a}_{sub}, \mathcal{E}^{q,a}_{sub})$ is extracted from the graph G, including all k-hop neighbors of each topic entity.

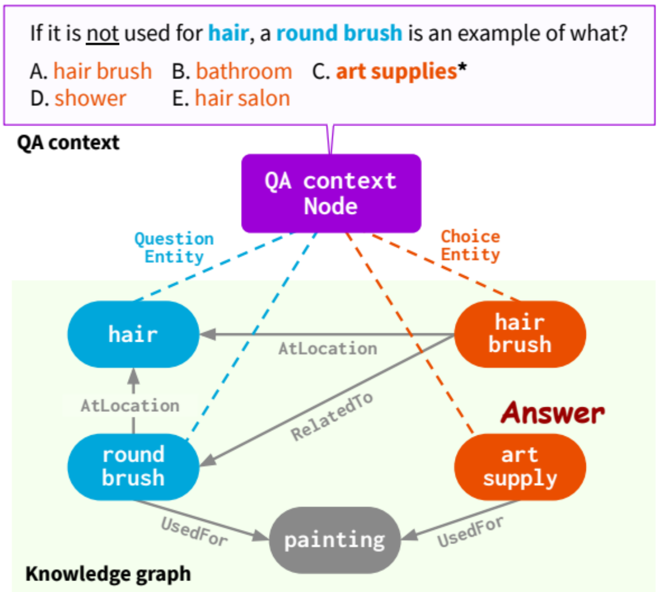

Figure 2.5: Example of the QA context and its integration with a knowledge graph.
From: *QA-GNN: Reasoning with Language Models and Knowledge Graphs for Question Answering*, 2021 [15]

As shown in figure 2.6, question q and answer a are concatenated to obtain the QA context. The LM is then used to obtain the representation of the QA context. Following, the subgraph $\mathcal{G}_{sub}$ is retrieved from the KG.
After that, the context node $\mathcal{Z}$ is added to the graph and connected to the topic entities $\mathcal{V}_{q,a}$ to obtain a joint graph over the two sources of knowledge.
To capture the relationships between $\mathcal{Z}$ and every other node, first of all, a relevance score is computed for each pair, which is used as an additional feature

for each node.
Finally, an attention-based GNN module spread information across nodes to update node features which will then be used to make the final prediction.

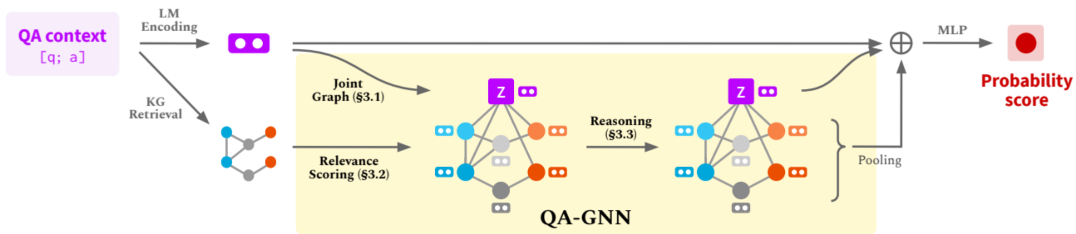

Figure 2.6: Overview of the model.
From: *QA-GNN: Reasoning with Language Models and Knowledge Graphs for Question Answering*, 2021 [15]

2.2.2 Joint graph representation

The first step is to create a merged graph with both knowledge sources, and to do that a context node $\mathcal{Z}$ is created to capture the QA context. ‡ is connected to each topic entity in $\mathcal{V}_{q,a}$ with two new relation types $r_{z,q}$ and $r_{z,a}$, which capture relationships between the context node and the topic entities, depending on whether the entities appear in the question or in the answer.
Each node embedding is initialized by the corresponding LM representation.
In the original solution, each node is also associated with one of the four node types $\mathcal{T} = Z, Q, A, O$, each indicating the context node $\mathcal{Z}$, nodes in $\mathcal{V}_q$, nodes in $\mathcal{V}_a$ and other nodes.

2.2.3 Relevance scoring

Many nodes retrieved from the original KG may be irrelevant under the QA context. As shown in figure 2.7, for example, nodes "holiday" and "riverbank" are off-topic under the current context, while "human" and "place" are generic. For this reason, their score should be much lower than more informative nodes. These irrelevant nodes may cause overfitting and uselessly complicate the model's training phase, especially when the retrieved subgraph is large.
The solution proposed by Yasunaga et al. [15] solves this problem via a node relevance scoring technique. The pre-trained language model is used to score the relevance of each node of the retrieved KG under the QA context.
For each node v, the entity embedding $text(v)$ is concatenated with the QA context embedding $text(z)$ and then is computed the relevance score as follows:

$$p_v = f_{head}(f_{enc}([text(z); text(v)])) \tag{2.13}$$

$f_{head} \circ f_{enc}$ is the probability of $text(v)$ computed by the LM. p_v is the relevance score of the node v under the current QA context, and will be used during the reasoning phase and to prune the retrieved subgraph.

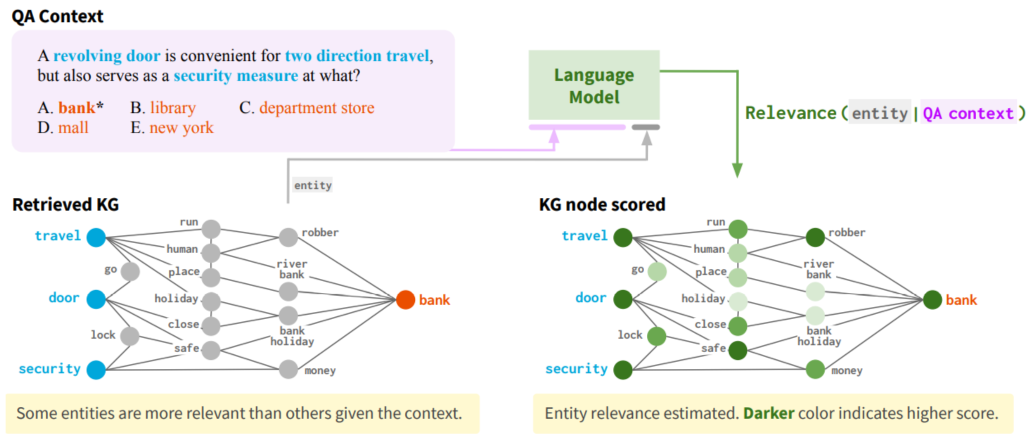

Figure 2.7: Relevance scoring of the retrieved KG.
From: *QA-GNN: Reasoning with Language Models and Knowledge Graphs for Question Answering*, 2021 [15]

2.2.4 GNN architecture

The original work exploits, to perform reasoning on the working graph, a GAT-inspired GNN. The GNN adopts a multi-layer architecture, and, in each layer l, the representation of each node v of the graph is updated using the following formula:

$$h_t^{l+1} = f_n(\sum_{s \epsilon N_t \cup t} \alpha_{st} m_{st}) + h_t^l \tag{2.14}$$

N_t is the set of neighbor nodes of node t, m_{st} is the message sent from each neighbor s to node t, and α_{st} is the attention coefficient that gives a different relevance to each message m_{st}.
The sum of every message received is then passed through a 2-layer MLP f_n with batch normalization and is finally summed to the embedding coming from the previous layer.
For each note t, h_t^0 is set using a linear transformation that maps the initial node embedding obtained by the LM.

As the GNN operates on the graph, it will leverage the gap between the QA context and the KG subgraph.

Message-passing system

As the working graph is a relational graph, the message from a source node to a target node needs to consider the relation type of the edge connecting them and, only in the original work, the node types.
For this reason, first of all, the type embedding u_t is computed for each node t, as well as the relation embedding r_{st} from node s to t. The new embeddings are computed as:

$$u_t = f_u(ut) \tag{2.15}$$

$$r_{st} = f_r(e_{st}, us, ut) \tag{2.16}$$

u_s, $u_t \epsilon \{0,1\}^{|T|}$, are one-hot vectors indicating the node types of s and t, $e_{st} \epsilon \{0,1\}^{|R|}$ is a one-hot vector indicating the relation type of the edge between node s and s, f_u is a linear transformation, and f_r is a 2-layer MLP.

The message from node s to node t is computes as:

$$m_{st} = f_m(h_s^l, u_s, r_{st}) \tag{2.17}$$

f_m is a linear transformation that takes into account the source node embedding, the source note type and the relation type between source and target node.

Node type, relation and score aware attention

The attention coefficient is used to assign a different relevance to each pair of nodes, and should consider node types, relations and relevance score.
First of all the relevance score of each node t is embedded as:

$$p_t = f_p(pt) \tag{2.18}$$

f_p is an MLP. To compute the attention score, α_{st} from node s to node t are first obtained starting from query q and key k vectors as:

$$q_s = f_q(h_s^l, u_s, p_s) \tag{2.19}$$

$$k_t = f_k(h_t^l, u_t, r_{st}) \tag{2.20}$$

f_q and f_k are both linear transformations. The final attention coefficient is computed as:

$$\Upsilon_{st} = \frac{q_s^T k_t}{\sqrt{D}} \tag{2.21}$$

$$\alpha_{st} = \frac{exp(\Upsilon_{st})}{\sum_{t' \epsilon N_s \cup s} exp(\Upsilon_{st'})} \tag{2.22}$$

2.2.5 Results obtained by the original model

QA-GNN was evaluated on three question-answering datasets. In our case, the one of interest is also involved in our experiments, namely MedQA-USMLE. Figure 2.8 compares results obtained from QA-GNN and those obtained from LM approaches.
As can be seen, QA-GNN outperforms state-of-the-art fine-tuned LMs, proving to be an effective improvement of LMs and KGs in the biomedical domain.
Our goal during the experiments will be to exceed the maximum accuracy achieved by QA-GNN (38%).

Methods	**Test**
BERT-base (Devlin et al., 2019)	34.3
BioBERT-base (Lee et al., 2020)	34.1
RoBERTa-large (Liu et al., 2019)	35.0
BioBERT-large (Lee et al., 2020)	36.7
SapBERT (Liu et al., 2020a)	37.2
SapBERT + QA-GNN (**Ours**)	**38.0**

Figure 2.8: QA-GNN test accuracy on MedQA-USMLE.

Capitolo 3

Contributions

This chapter describes the contribution to Yasunaga et al. work [15], the experiments carried out, and the results obtained, motivating each choices made.

3.1 Proposed design

The proposed design is similar to the original QA-GNN solution, as shown in figure 3.1 , with some differences due to replacing their GNN with EGAT.

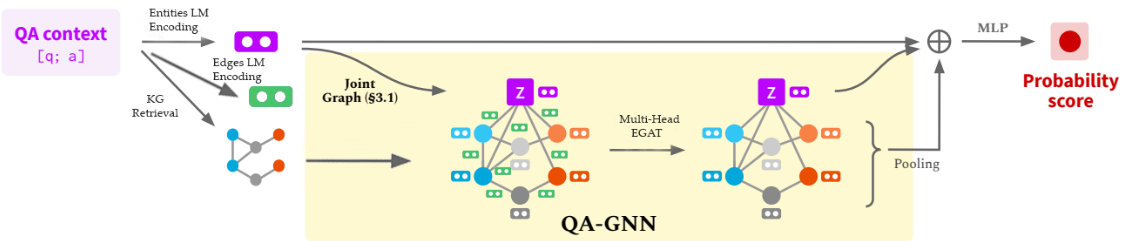

Figure 3.1: Proposed solution. The original GNN has been replaced with EGAT. Relation embeddings are also generated and relevance score is not used anymore.

Our proposed design differs from the original QA-GNN solution mainly in the following features:

- Relevance scores of the various nodes are no longer considered as node features, buto only to prune the knowledge graph.
- The typology of each node is not considered.

- The relation type relation is not considered anymore, as the edge embeddings already capture it.

In addition, our solution has undergone some modifications from the original EGAT, proposed in the experimental phase because, as we will see in sections 3.2 and 3.3, they can improve performances:

- Dummy node features/relationships created during the addition of self-loops to the graph are no longer initialized to 0, but by averaging adjacent features/nodes.
- The final aggregation performed in the merge layer is not necessarily done by concatenation, but it is up to the user to choose the reference mode (e.g., average, sum).
- The text used for the generation of edge embeddings is pre-processed in order to increase its accuracy.

3.2 Dataset preparation

For our tests, we used the MedQA dataset [53], a biomedical dataset that contains a series of questions and for each to be four associated answers, of which only one is the correct one.
The original work provides a script capable of processing the dataset and producing a subgraph for each pair question-answer, stored using the pickle module.

3.2.1 Reverse graph creation

The implemented EGAT requires as input both adjacency matrices for the standard and reversed graph. The latter is not taken into account in the original QA-GNN solution, so the preprocessing script had to be modified.
Although computationally expensive, this step can be performed only once before training. The results of the transformations can also be stored on file, not negatively impacting the solution's computational cost.

Optimizations

EGAT's original solution required as input two adjacency matrices A and A' of dimension $[N, N, M]$ and $[M, M, N]$, respectively, with N number of nodes and M number of relations. Each cell $A[i, j, k]$ has value 1 if nodes i and j are connected by relation k of the original graph, similarly for the matrix A'

which refers to the reverse graph. This mode of representation involves several problems:

- The matrices are sparse and very large, although only a few cells have value 1 and are therefore relevant
- Since it is necessary to evaluate the entire matrix, the training phase is unnecessarily slow and complex.
- Also, the phase of preprocessing comes uselessly slow down

For the reasons mentioned above, the proposed solution uses compressed adjacency matrices, which only keep track of the existing relationships, thus allowing for smaller data structures and speeding up both preprocessing and actual training times.
The new adjacency matrix is represented by two tensors, named *edge_index* and *edge_type*. The former is a tensor of dimension $[2, M]$, while the latter is a tensor of dimension $[M]$, with M number of relations.
Given $edge\_index[0][i] = x$, $edge\_index[1][i] = y$, and $edge\_type[i] = z$, nodes x and y are connected via a relation of type z.

3.2.2 Relation embeddings creation

The original solution does not consider edge features, but only node ones, generated by a script which uses the specified language model. For this reason, it was necessary to repeat the same procedure in order to also generate edge features. Node and edge embeddings can be written on file and then read to be used inside the GNN, making this process executable only once in the preprocessing phase.
The script that performs this operation is analogous to the one already present for the generation of node embeddings. The only difference is that the input received does not correspond to nodes' text but relations' text.

Optimizations

We noticed that the relations' text included non-alphanumeric characters, such as "_" used instead of space, "?" at the beginning of the sentence in some reverse edges, etc.
These are not characters that the language model is used to working with, thus making the embeddings of the relations that contain them less accurate.
Therefore, we decided to test the model during our experiments even after processing the relations' text and removing every unrecognized character.

3.3 EGAT variant with graph transformation (EGATv2)

We have also proposed a new solution, which involves transforming the input graph to aggregate edge features directly within those of the nodes. This transformation is shown in figure 3.2: for each relation, a new node is created and initialized with a function that considers the source node and the edge embeddings (in our case subtraction between embeddings). The new node is then connected to the source node, with a new edge of the same type of the original one.

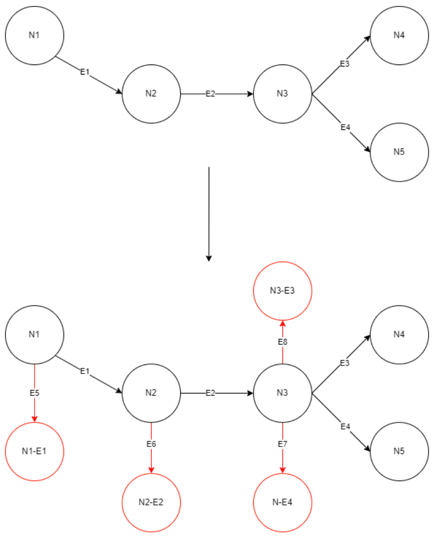

Figure 3.2: The proposed graph transformation. For each relation, a new node is created and initialized with a function that considers the source node and the edge embeddings. The now node is then connected to the source node, with a new relation of the same type of the original one.

This transformation is also reflected on the reverse graph in which additional edges become new nodes and nodes resulting from aggregation, where expected, become new edges.

3.4 Training

The experimental setup for the question-answering task is described below.

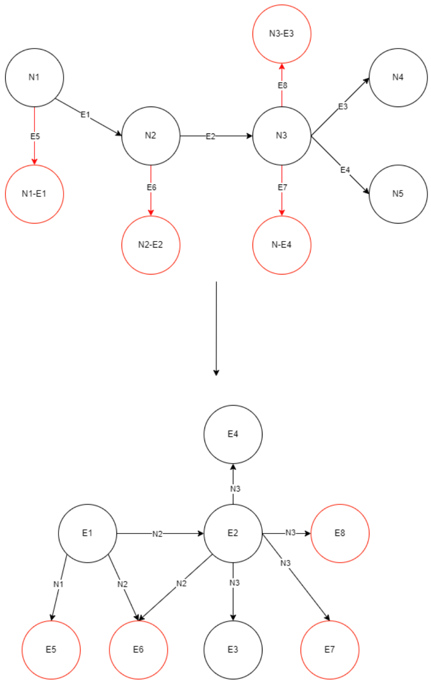

Figure 3.3: The same transfomation on the graph is reflected on the reversed graph.

3.4.1 Dataset

The dataset used for our experiments is MedQA-USMLE [53], a 4-way multiple choice QA dataset that requires biomedical and clinical knowledge, whose questions are originally from practice tests for the United States Medical License Exam (USMLE). The dataset contains 12,723 questions. As in the original QAGNN [15] training, we use the original data splits from Jin et al. [54].

3.4.2 Knowledge graphs

We used a Knowledge Graph constructed by Yasunaga et al. [15] that integrates the Disease Database portion of the Unified Medical Language System [55] and DrugBank [56]. The knowledge graph contains 9,958 nodes and 44,561 edges.
Given each QA context (question + answer), we retrieve the subgraph G_{sub} from G, with hope size $k = 2$. G_{sub} is then pruned to keep only the top 200 nodes according to the node relevance score computed.

3.4.3 Implementation and training details

We test our solution (EGAT) with dimension ($D = 200$). We set the number of layers L from $\{2, 5, 8\}$ and the number of heads K from $\{2, 4, 8\}$. We set the dropout rate to 0.2 applied to each layer. We also tested the EGAT model after removing all non-alphanumeric characters from the relations' text, and we initialized the embeddings of the additional nodes created during the addition of the self-loops with both 0 values and the average of the adjacent nodes. Entities and relation embeddings are always initialized using the representations from PubMedBert-abstract [57].
Features in the final merge layer have been both concatenated and averaged in our tests.
The EGAT solution with graph transformation (EGATv2 3.3) was tested with fixed parameters ($L = 5$, $K = 4$), additional nodes' features filled with 0 values, merge layer features averaged and alphanumeric characters not removed, but with two different GNNs: the proposed EGAT and the classic GAT, so as to verify the importance of aggregating to the node features those of the outgoing edges.
We train the model with RAdam oprimizer using one GPU (GeForce RTX 3090 Turbo).
We set the batch size ($b\_s = 128$), learning rate for the LM module ($elr = 5e - 5$), and learning rate for the GNN module ($dlr = 1e - 3$).

3.5 Results

Figure 3.4 shows the results of our two best solutions on MedQA-USMLE.

The best EGAT solution was the one with 5 layers ($L = 5$), 4 heads ($K = 5$), features of additional nodes initialized as average of the adjacent ones and final aggregation in the merge layer by averaging. The model achieved an accuracy of 42.18%, beating the previous state-of-the-art of 4.18%.

The best EGATv2 solution turned out to be one that leverages classic GAT as GNN. As explained in section 3.4.3, EGATv2 was trained with fixed parameters. The model achieved an accuracy of 42.11%, beating the previous state-of-the-art of 4.11%.

For the two best models, we also decided to run 15 additional epochs, for a total of 30, to see if performance could further increase.
The solution with EGAT, as the number of epochs increases, fails to increase the test accuracy.
EGATv2, at the twenty-first epoch, reaches an accuracy on the test set of 43.13%, beating the previous state-of-the-art of 5.13%, as shown in figure 3.5.

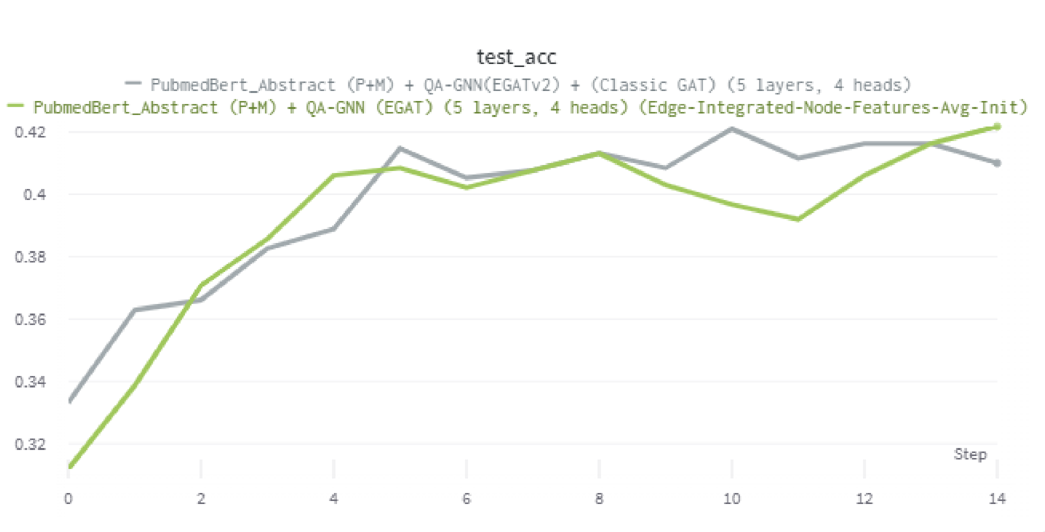

Figure 3.4: Best results obtained by our models.

Our experiments have shown that edge features can positively affect the performance of a GNN.
Moreover, since graphs extracted from the MeQA-USMLE dataset include only 14 types of relations (34 considering the inverse ones), we also believe our approach may be even more influential on graphs in which the number of distinct relation types is higher.

3.5.1 Ablations

We conduct ablation studies on the MeQA-USMLE dataset to show the importance of each parameter and strategy applyed to our model. For all these experiments, we train the model (EGAT/EGATv2) for 15 epochs and compare it to the base EGAT solution.

Baseline

The EGAT base solution has the following features: $L = 5$, $K = 4$, additional nodes' features initialized with 0 values, feature averaged in the merge layer and non-alphanumeric characters not removed.

As shown in figure 3.6, our base solutions reach, after at its 15-th epoch, 41.4% accuracy on the test set. This result will be considered as our lower-bound and compared to other trainings to show the importance of each parameters and strategy applyed to our model.

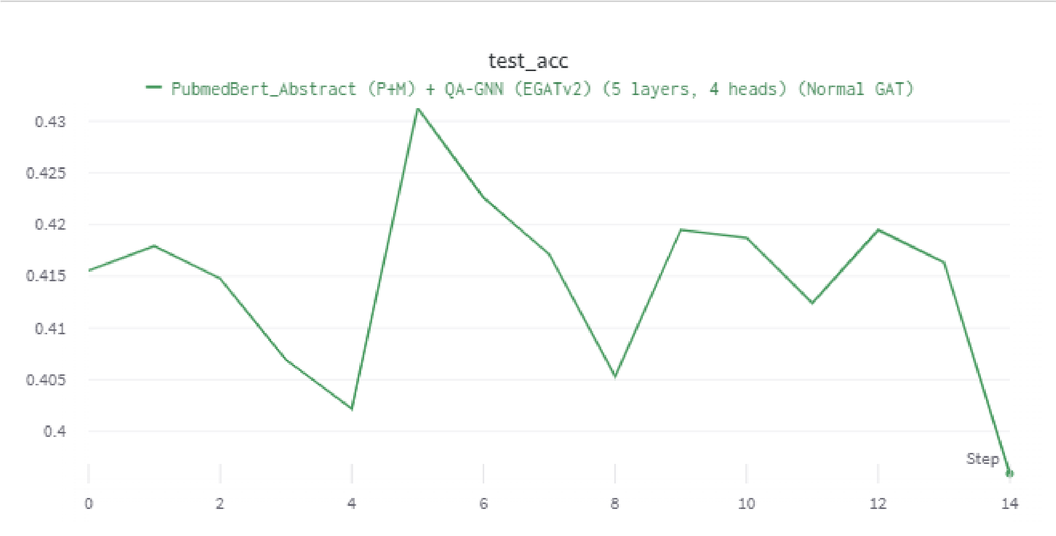

Figure 3.5: Results obtained by EGATv2 after 30 epochs.

The importance of the number of EGAT layers and heads

The following paragraphs show the performance obtained by changing the number of layers(L) and/or heads (K).

Changing number of layers: The model shown in figure 3.7 has a fixed number of heads ($K = 4$) and a variable number of layers $L \epsilon \{2, 5, 8\}$. Dummy node features are initialized with zero values, features in the merge layer are averaged, and non-alphanumeric characters are not removed.

The 2-layers solution ($L = 2$) performed best, achieving an accuracy on the test set of 41.87% and thus improving the base solution by 0.47%.

Changing number of heads: The model shown in figure 3.8 has a fixed number of layers ($L = 5$) and a variable number of heads $K \epsilon \{2, 4, 8\}$. Dummy node features are initialized with zero values, features in the merge layer are averaged, and non-alphanumeric characters are not removed.

The solution with two heads ($K = 2$) performed best, achieving an accuracy on the test set of 41.95% and thus improving the base solution by 0.55%.

Changing number of layers and heads: As shown in figure 3.9, the best performance was achieved by the solution with 5 layers ($L = 5$) and 4 heads ($K = 4$) which is the same configuration used in the baseline solution in terms of layers and heads. This setup achieved an accuracy on the test set of 41.95%, improving the baseline by 0.55%.

Figure 3.6: Results of the EGAT base solution.

The importance of non-alpanumeric characters

This solution uses 5 layers ($L = 5$), 4 heads ($K = 4$), additional node features are initialized with 0 values, features are averaged in the merge layer and non-alphanumeric characters are removed.

As shown in figure 3.10, removing non-alphanumeric characters improves the accuracy on the test set by 0.39%, achieving its maximum accuracy (41.79%) at its 14-th epoch.

The importance of of the final merge logic

This comparison was made at an advanced stage of development. For this reason the two solutions share the number of layers and heads ($L = 5$ and $K = 4$), do not provide for the removal of non-alphanumeric characters and initialize the features of additional nodes by averaging adjacent ones. The difference between the two models is the logic with which the features are merged into the final merge layer: in one case by concatenating them and in the other by averaging.

As shown in figure 3.12, averaging features in the final merge layer is more efficient than concatenating them. Specifically, the solution that exploits feature averaging achieves an accuracy of 42.18% on the test set, 1.17% higher than the solution that uses concatenation. Despite this, the latter reaches maximum accuracy after only 4 epochs, compared to 15 for the best solution.

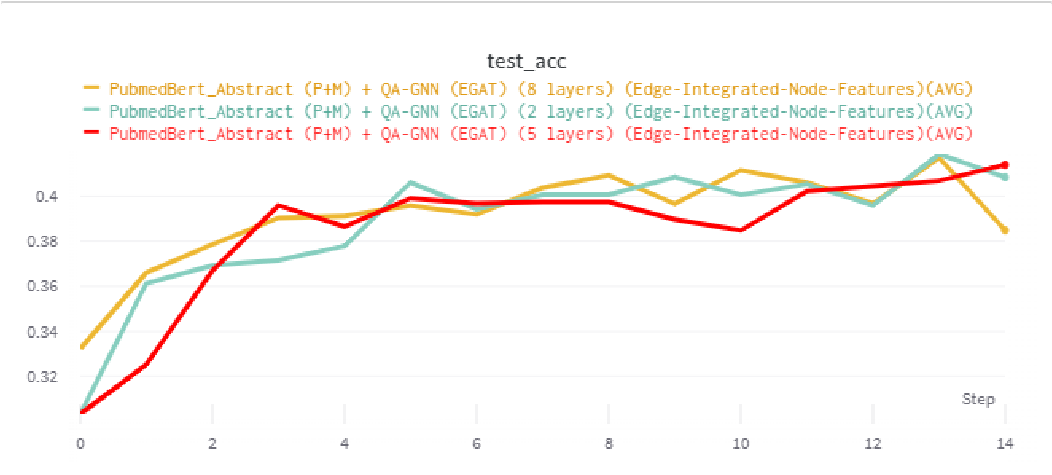

Figure 3.7: Results of the EGAT solution with different number of layer ($L\epsilon\{2,5,8\}$).

The importance of the new nodes' features initialization logic

This solution uses the same parameter of the base one, so 5 layers ($L = 5$), 4 heads ($K = 4$), features are averaged in the merge layer and non-alphanumeric characters are not removed, but additional node features are initialized with the average of the adjacent nodes (for the reverse graph, with the average of adjacent relations).

As shown in figure 3.12, this solution improves the accuracy on the test set by 0.78%, achieving its maximum accuracy (42.18%) at its 15-th epoch.

EGATv2

The EGAT solution with graph transformation (3.3) was tested with fixed parameters ($L = 5$,$K = 4$), additional nodes' features filled with 0 values, merge layer features averaged and alphanumeric characters not removed, but with two different GNNs: the proposed EGAT and the classic GAT.

EGAT As shown in figure 3.13, aggregating the edge features directly within those of nodes can lead to an increase in performance; specifically, the solution that exploits the graph transformation explained in section 3.3 achieves an accuracy on the test set of 41.95%, 0.55% higher than the baseline solution.

GAT As shown in figure 3.14, aggregating the edge features directly within those of nodes and using the classic GAT as GNN further increases performance, reaching an accuracy on the test set of 42.11%, 0.71% higher than the baseline

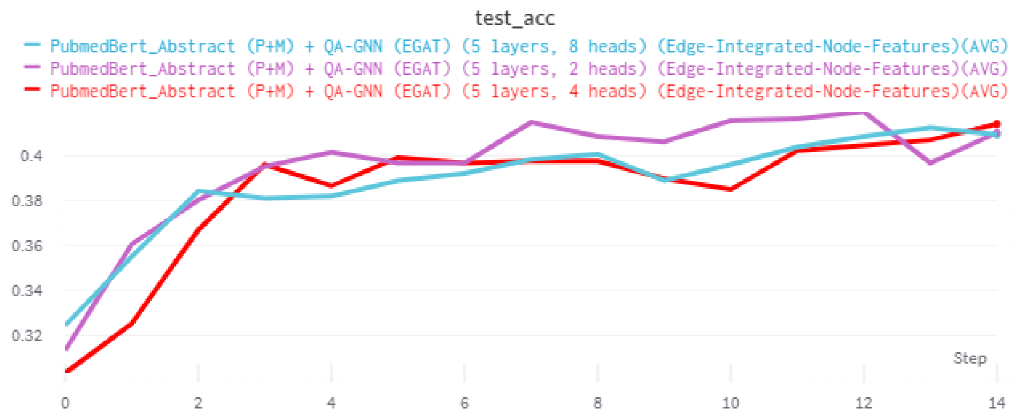

Figure 3.8: Results of the EGAT solution with different number of heads ($K \epsilon \{2, 4, 8\}$).

solution and 0.16 higher than the solution with EGAT as GNN.
This suggests to us that the combined use of EGAT with graph transformation leads to the introduction of redundant information (edge features), in GAT not present, causing overfitting and a decrease in overall performance.

Figure 3.9: Results of the EGAT solution with different number of heads and layers ($L = 2$, $K = 8$).

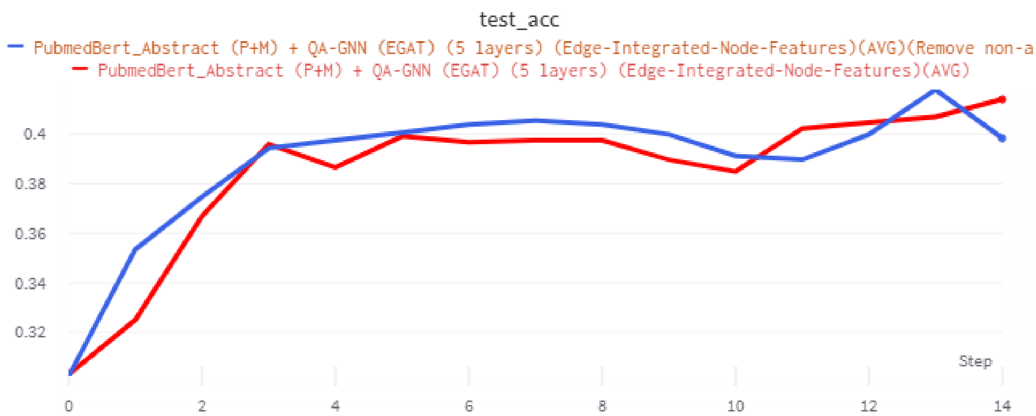

Figure 3.10: Results of the EGAT solution with non-alphanumeric characters removed from relations' text.

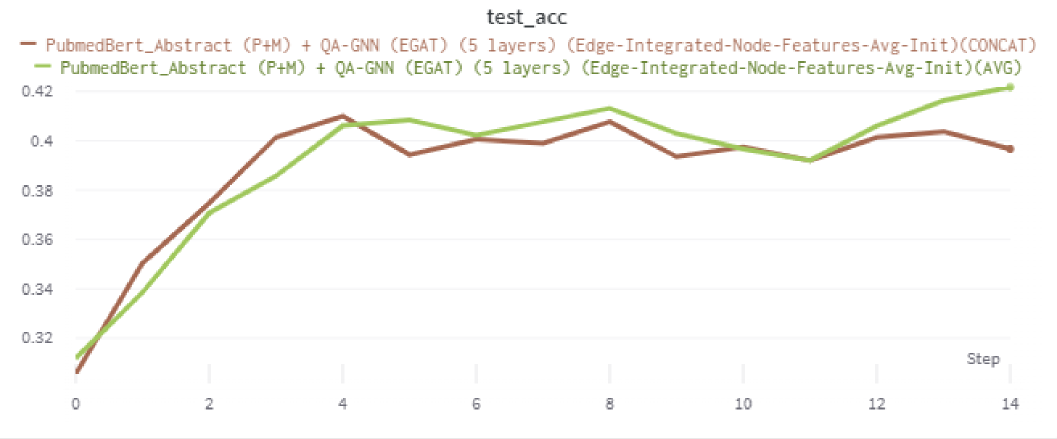

Figure 3.11: Results of the EGAT solution with a different merge logic in the final layer.

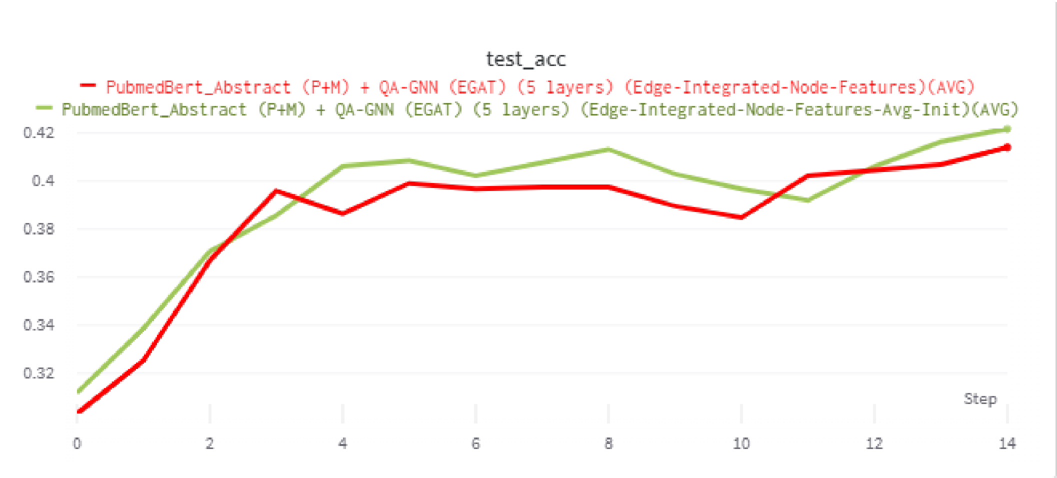

Figure 3.12: Results of the EGAT solution with features of additional nodes initialized as average of the adjacent ones.

Figure 3.13: Results of the solution with graph transformation, using EGAT as GNN.

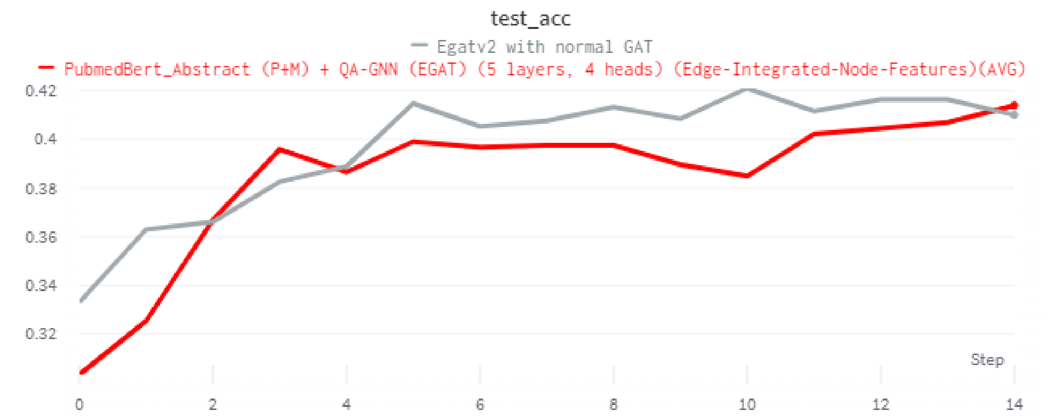

Figure 3.14: Results of the solution with graph transformation, using classic GAT as GNN.

Capitolo 4

Implementation

The following chapter shows each model implemented for this thesis, some of which are used on the practical side during the training (e.g., EGAT). In contrast, others have been necessary to deeply understand the functioning of some GNNs (e.g., GAT) and development frameworks.
The code produced during the thesis can be found at the following links:

- Github repository
- Google Colab notebooks

4.1 Preliminary Technical Choices

Python is the most popular programming language for data science. It's an open source, general-purpose and intuitive language born in 1991 which supports different paradigms.
For these reasons it is the most widely used programming language for data science, because it is fast and therefore the best option for data manipulation. Moreover, the numerous packages contained in python make it easier for programmers to implement complex models and operate on data.
All implementations are done using the following framework and libraries:

- **PyTorch**[58]: an open source machine learning framework based on torch library which has greatly simplified data processing, model creation, and model training.
- **PyTorch Geometric**[59]: a library built upon PyTorch to easily write and train GNNs.
- **Deep Graph Library**[60]: frequently abbreviated as DGL. A framework agnostic, efficient and scalable library to create and train GNNs.

4.2 GAT

The following subsections show GAT's implementations, exploiting PyTorch and DGL.

4.2.1 PyTorch

```
class GraphAttentionLayer(Module):
  def __init__(self, in_features: int, out_features: int, n_heads:
      int, is_concat: bool = True, dropout: float = 0.6,
      leaky_relu_negative_slope: float = 0.2):

    self.linear = nn.Linear(in_features, self.n_hidden * n_heads,
       bias=False)
    self.attn = nn.Linear(self.n_hidden * 2, 1, bias=False)
    self.activation =
       nn.LeakyReLU(negative_slope=leaky_relu_negative_slope)
    self.softmax = nn.Softmax(dim=1)
    self.dropout = nn.Dropout(dropout)
```

In order to instantiate the model, the following parameters are required:

- **in_features**: the number of input features per node.
- **out_features**: the number of output features per node.
- **n_heads**: number of atttention heads.
- **is_concat**: merge logic for attention heads' output. Default is concatenation (true), otherwise feature are averaged.
- **dropout**: dropout probability to regularize the training process.
- **leaky_relu_negative_slope**: negative slope for LeakyReLU to avoid the "dying ReLU" problem.

The model to its inside defines five layers, thought respectively, for the following purposes:

- **self.linear**: this layer performs the initial transformation on the node features to get more expressive power (equation 1.8).
- **self.attn**: this layer computes the non-normalized attention coefficients (equation 1.10).

- **self.activation**: on the non-normalized attention coefficient, a LeakyReLU function is applied.
- **self.softmax**: the softmax layer normalize attention coefficient considering node neighbors (equation 1.11).
- **self.dropout**: at the end, a dropout regularization is applied to regularize the training process.

```
...
e = self.activation(self.attn(g_concat))
e = e.masked_fill(adj_mat == 0, float('-inf'))
a = self.softmax(e)
a = self.dropout(a)
attn_res = torch.einsum('ijh,jhf->ihf', a, g)
...
```

The forward step is very intuitive: after creating the matrix containing all possible concatenations between nodes, the previously defined layers are applied in sequence to produce the attention coefficients, and finally, for each node, the output is calculated as a weighted sum for the nodes in the neighborhood.

4.2.2 DGL

```
class GATLayer(nn.Module):
    def __init__(self, g, in_dim, out_dim):

        self.fc = nn.Linear(in_dim, out_dim, bias=False
        self.attn_fc = nn.Linear(2 * out_dim, 1, bias=False)
```

In order to instantiate the model, the following parameters are required:

- **g**: DGL graph. Contains the information for constructing the graph, including the adjacency matrix.
- **in_dim**: the number of input features per node.
- **out_dim**: the number of output features per node.

The model to its inside defines only two layers:

- **self.fc**: the layer used for the initial transformation on node features to get more expressive power (equation 1.8).

- **self.attn_fc**: The layer used to compute the non-normalized attention scores (equation 1.10).

```
def forward(self, h):
    z = self.fc(h)
    self.g.ndata['z'] = z
    self.g.apply_edges(self.edge_attention)
    self.g.update_all(self.message_func, self.reduce_func)
    return self.g.ndata.pop('h')
```

The forward step exploits three user defined functions, necessary to perform the graph convolution, executed through DGL API.
The unnormalized attention score e_{ij} (equation 1.10) is calculated using the embeddings of adjacent nodes. For this reason it can be viewed as edge data which can be calculated by the apply_edges API, which updates features of the edges by the provided function.
The update_function API is used to start the message passing procedure on all the nodes and requires two parameters: the message function and the reduce function. The first one defines how messages between nodes should be generated, while the latter defines how received messages should be aggregated.

```
def edge_attention(self, edges):
    z2 = torch.cat([edges.src['z'], edges.dst['z']], dim=1)
    a = self.attn_fc(z2)
    return {'e': F.leaky_relu(a)}
```

The edge_attention function returns the unnormalized attention scores by creating the embedding concatenation and applying the linear transformations defined in the model.

```
def message_func(self, edges):
    return {'z': edges.src['z'], 'e': edges.data['e']}
```

The message_func function defines how each message exchanged between nodes should be generated.
In this case, each node sends its embedding and the attention score computed before.

```
def reduce_func(self, nodes):
    alpha = F.softmax(nodes.mailbox['e'], dim=1)
    h = torch.sum(alpha * nodes.mailbox['z'], dim=1)
    return {'h': h}
```

The last function, called reduce_func, defines how to aggregate received messages.

In this case, the attention scores are normalized using a softmax and the neighbor embeddings are aggregated and weighted by the attention scores (equation 1.11).

4.3 GATv2

The following section shows the PyTorch implementation of Gatv2.

```
class GraphAttentionV2Layer(Module):
    def __init__(self, in_features: int, out_features: int, n_heads:
        int, is_concat: bool = True, dropout: float = 0.6,
        leaky_relu_negative_slope: float = 0.2):
        self.linear_l = nn.Linear(in_features, self.n_hidden *
            n_heads, bias=False)
        self.attn = nn.Linear(self.n_hidden, 1, bias=False)
        self.activation =
            nn.LeakyReLU(negative_slope=leaky_relu_negative_slope)
        self.softmax = nn.Softmax(dim=1)
        self.dropout = nn.Dropout(dropout)
```

The model requires the same parameters as GAT (4.2.1) to be instantiated and defines the same layers needed to perform convolution within it.

```
e = self.attn(self.activation(g_sum))
e = e.masked_fill(adj_mat == 0, float('-inf'))
a = self.softmax(e)
a = self.dropout(a)
```

The only difference from the classic version of GAT is in the order in which the operations are performed.
Specifically, in this case, the LeakyReLU activation function is first applied, and then the linear layer is used to calculate the unnormalized attention coefficients. Finally, as in GAT, the graph structure is injected using the adjacency matrix, the coefficients are normalized using a softmax, and the dropout technique is applied.

4.4 RGCN

The following section shows the DGL implementation of RGCN.

```
class RGCNLayer(nn.Module):
  def __init__(self, in_feat, out_feat, num_rels, num_bases=-1,
      bias=None, activation=None, is_input_layer=False):
```

```
    if self.num_bases <= 0 or self.num_bases > self.num_rels:
        self.num_bases = self.num_rels
    self.weight = nn.Parameter(torch.Tensor(self.num_bases,
        self.in_feat, self.out_feat))
    if self.num_bases < self.num_rels:
        self.w_comp = nn.Parameter(torch.Tensor(self.num_rels,
            self.num_bases))
```

In order to instantiate the model, the following parameters are required:

- **in_feat**: the number of input features per node.
- **out_feat**: the number of output features per node.
- **num_rels**: the total number of edges.
- **num_bases**: the number of basis used for basis decomposition and reduce the number of parameters.
- **bias**: the bias.
- **activation**: if not None, applies the given activation function to the result.
- **is_input_layer**: if set True, the layer is designed as the input one, so node features will be initialized as one-hot vectors for each node.

```
if self.num_bases <= 0 or self.num_bases > self.num_rels:
    self.num_bases = self.num_rels
self.weight = nn.Parameter(torch.Tensor(self.num_bases, self.in_feat,
                                        self.out_feat))
if self.num_bases < self.num_rels:
    self.w_comp = nn.Parameter(torch.Tensor(self.num_rels,
        self.num_bases))
```

The number of bases is used to reduce the number of trainable parameter that grows in relation to the number of relations. For this reason, if num_bases is greater than num_rels or invalid, is set to num_rels. self.weight is the weight bases matrix used for basis decomposition (V_b in equation 1.6).
self.w_comp are the linear combination coefficients for basis decomposition (a_{rb} in equation 1.6).

```
g.update_all(message_func, fn.sum(msg='msg', out='h'), apply_func)
```

The forward step can be summarized by the following call, which exploits the DGL update_all API and two user-defined functions necessary to define the exchange of messages between nodes and axtivation function applied to the result. The aggregation is a simple sum over all messages received.

```
weight = self.weight.view(self.in_feat, self.num_bases, self.out_feat)
weight = torch.matmul(self.w_comp, weight).view(self.num_rels,
    self.in_feat, self.out_feat)
```

In the forward step, the weight matrix for the basis decomposition is created by multiplying the two matrices defined within the model.

```
if self.is_input_layer:
    def message_func(edges):
        embed = weight.view(-1, self.out_feat)
        index = edges.data['rel_type'] * self.in_feat + edges.src['id']
        return {'msg': embed[index] * edges.data['norm']}
else:
    def message_func(edges):
        w = weight[edges.data['rel_type']]
        msg = torch.bmm(edges.src['h'].unsqueeze(1), w).squeeze()
        msg = msg * edges.data['norm']
        return {'msg': msg}
```

After that, depending on whether basis-decomposition is used or not, the function for message exchange is defined.
If the layer is an input layer, the matrix multiplication can be converted to be an embedding lookup based on source node id. Otherwise, the message is generated as matrices multiplication (1.5).

```
def apply_func(nodes):
    h = nodes.data['h']
    if self.bias:
        h = h + self.bias
    if self.activation:
        h = self.activation(h)
    return {'h': h}
```

The apply_func function is used to apply a specific activation function on the result. If no activation function is passed as parameter, the result will not be transformed.

4.5 EGAT

The following subsections show EGAT's implementations, exploiting PyTorch and PyTorch Geometric, showing the most relevant portions of code, and motivating the choices made.

4.5.1 PyTorch

EGAT layer

As described previously, each EGAT layer is composed by two blocks in a symmetrical design, to update in an equivalent way both nodes and edges features: node and edge attention block.

```
class EGATLayer(nn.Module):
  def __init__(self, in_features_nodes: int, in_features_edges: int,
      out_features_nodes: int, out_features_edges: int):
    self.linear_nodes = nn.Linear(in_features_nodes,
        out_features_nodes, bias=False)
    self.linear_edges = nn.Linear(in_features_edges,
        out_features_edges, bias=False)
    self.attn_nodes = nn.Linear(out_features_nodes * 2 +
        out_features_edges, 1, bias=False)
    self.attn_edges = nn.Linear(out_features_edges * 2 +
        out_features_nodes, 1, bias=False)
    self.activation = nn.LeakyReLU(negative_slope=0.02)
    self.softmax = nn.Softmax(dim=1)
```

In order to instantiate the model the following parameters are required as input:

- **in_features_nodes**: The number of input features per node
- **in_features_edges**: The number of input features per edge
- **out_features_nodes**: The number of output features per node
- **out_features_edges**: The number of output features per edge

The model exploits 6 distinguished layers, necessary to perform the operations listed in chapter 2.1:

- The first two are linear layer for initial node and edges transformation, used to get a more expressive power and obtain higher level features (equation 2.1 and 2.2). The mapping is obviously made taking into account the number of features in input and output.

- The third is a linear layer to compute the attention score in the node attention block (equation 2.5). The size of the layer's input is out_features_nodes * 2 + out_features_edges because the attention score is computed from the concatenation of the source node, the target node and the edge connecting them, after the previous transformation.

- The fourth is a linear layer to compute the attention score in the edge attention block. In this case, the size of the layer's input is out_features_edges * 2 + out_features_nodes because the attention score is computed from the concatenation of the source edge, the target edge and the node between them.

- The fifth layer refers to equation 2.6 and, through the use of a LeakyReLU activation function, allows to compute the not-normalized attention scores. The negative slope prevents the dying ReLU [61] problem because it allows backpropagation also on negative values.

- The last layer exploits a softmax function to normalize the attention scores.

```
def forward(self, nodes_features: torch.Tensor, edges_features:
    torch.Tensor, edges_mapping_matrix:torch.Tensor,
    is_node_attention: bool):
```

The forward step requires in input nodes and edges features, the adjacency matrix and a boolean indicating if the node or edge attention should be performed. The computation is pretty straightforward and after concatenating the matrices of nodes and edges to obtain every possible concatenations, it uses the layers explained before to compute the attention score and the new features, which will be the input of the subsequent layer. In the node attention block, the edge-integrated node features are also computed (equation 2.9).

EGAT with multiple layers

```
class EGAT(nn.Module):
    def __init__(self, in_features_nodes: int,
                 in_features_edges: int,
                 out_features_nodes: int,
                 out_features_edges: int,
                 n_layers : int):
```

A complete EGAT layer includes different single EGAT layers where the input of each one is the output of the previous. For this reason, the parameters required to instantiate the model are the same as those required to instantiate a single layer, with the addition of n_layers which indicates the total number of layers.

```
self.layers = nn.ModuleList()
self.layers.append(EGATLayer(in_features_nodes,in_features_edges,
    out_features_nodes, out_features_edges))
for i in range(n_layers - 1):
  self.layers.append(EGATLayer(out_features_nodes,
      out_features_edges, out_features_nodes,
      out_features_edges))
```

The model simply creates a list oh layers which is then populated with n_layers - 1 single EGAT layers in a loop and an extra EGAT layer. The reason is that the first EGAT layer has input sizes equal to the model input, while every other layer has input sizes equal to the output size of the previous layer, both for nodes and edges.

```
def forward(self, nodes_features: torch.Tensor, edges_features:
    torch.Tensor, edges_mapping_matrix:torch.Tensor,
    nodes_mapping_matrix:torch.Tensor):
  merged_nodes_feats =
      torch.empty(nodes_features.shape[0],self.out_features_nodes)
  for layer in self.layers:
    nodes_features, tmp_merged_nodes_feats, nodes, edges =
        layer(nodes_features, edges_features,
        edges_mapping_matrix, True)
    edges_features = layer(edges, nodes, nodes_mapping_matrix,
        False)
    merged_nodes_feats = torch.cat([merged_nodes_feats,
        tmp_merged_nodes_feats], dim = 1)
  return merged_nodes_feats
```

The forward step executes the forward step two times for each layer; the first computes new nodes features using node attention block while the second computes new edges features using edge attention block. The first forward step also return tmp_merged_nodes_feats, which represents edge-integrated node features that are finally concatenated and used as output of the EGAT.
For this computation, nodes and edges features, and adjacency matrices for both node and edge attention blocks are required.

Multi-head EGAT

As explained in the chapter 2.1, EGAT takes inspiration from GAT, more specifically it exploits a multi-head attention mechanism. K different EGAT models (called heads) are created, each one is independent from the others and produces it's own features. These output are at the end merged in a final layer, typically. with a concatenation.

```
class MultiHeadEGAT(nn.Module):
    def __init__(self, in_features_nodes: int,
                 in_features_edges: int,
                 out_features_nodes: int,
                 out_features_edges: int,
                 n_layers : int,
                 n_heads:int):
```

To create the model the required parameters are the same of the EGAT with multiple layers, because they will be used to create each single head. There is also one additional parameters, n_heads, which indicates the number of heads that will be created.

```
      self.layers = nn.ModuleList()
      for i in range(n_heads):
        self.layers.append(EGAT(in_features_nodes,in_features_edges,
            out_features_nodes, out_features_edges, n_layers))
```

The model simply creates a list of EGAT with multiple layers, each with the same parameters.

```
    def forward(self, nodes_features: torch.Tensor, edges_features:
        torch.Tensor, edges_mapping_matrix:torch.Tensor,
        nodes_mapping_matrix:torch.Tensor):
      head_outs = [layer(nodes_features, edges_features,
          edges_mapping_matrix,nodes_mapping_matrix) for layer in
          self.layers]
      return torch.cat(head_outs, dim=1)
```

The forward step requires as input the initial nodes and edges feature as well as both adjacency matrices of the graph and the "reversed" graph, used respectively by the node and edge attention blocks.
The method collects in the variable head_outs the output of each single heads and finally concatenates them to obtain the final result (equation 2.12).

4.5.2 PyTorch Geometric

The original implementation suffered from numerous problems, including:

- Graphs used in the dataset contain an average of 800 nodes and a variable number of relations, in general, highly connected graphs. The initial operation of merging nodes and edges feature matrices to obtain their concatenation led to the creation of matrices with too many entries(e.g., 400000000 entries), as all possible combinations were calculated even though they were present in the graph.
  This problem made it impossible to load them on the GPU, thus making the implementation unsuitable for our dataset.

- Using adjacency matrices of size $[N, N, M]$, with N number of nodes and M number of relations, made the computation of the same during the dataset preprocessing very slow, which, although it can be performed only once in the initial phase, would still slow down the experiments.

For these reasons it was decided to implement EGAT using PyTorch Geometric, as it provides a complex framework able to simplify the creation of any GNN and optimize its training.

EGAT with multiple layers

```
self.gnn_layers = nn.ModuleList([EGAT(args, hidden_size,
    hidden_size, n_ntype, n_etype, self.edge_encoder) for _ in
    range(k)])
```

Within the class used to instantiate the GNN, k EGAT layers are created, which share the required parameters for their creation.

```
class EGAT(MessagePassing):
    def __init__(self, args, emb_dim, transform_dim, n_ntype,
        n_etype, edge_encoder, head_count=4, aggr="add"):
        self.dim_per_head = emb_dim // head_count
        self.transform_dim = transform_dim
        self.linear_nodes = nn.Linear(emb_dim, self.transform_dim,
            bias=False)
        self.linear_edges = nn.Linear(emb_dim, self.transform_dim,
            bias=False)
        self.activation = nn.LeakyReLU(negative_slope=0.02)
        self.attn_nodes = nn.Linear(int(transform_dim * 3 /
            head_count), 1, bias=False)
        self._alpha = None
```

The EGAT class defines all the layers needed for the operations performed in the original paper, starting from the initial ones on nodes and edges up to those needed to compute attention coefficients and new features.

The MessagePassing base class provides a series of functionalities that facilitate the GNN creation by taking care of message propagation. We only had to define the message() method, to specify how propagated messages should be created and how they should be aggregated, in this case, "add" to respect the aggregation originally proposed by EGAT (equation 2.8).

```
def forward(self, x, edge_index, edge_type, node_type,
    rel_features, is_node_attention,
    return_attention_weights=False):
    ...
    new_rel_features = torch.zeros(x.size(0),
        self.emb_dim).to(edge_index.device)
    new_rel_features = torch.cat((rel_features, new_rel_features),
        dim = 0)
    loop_index = torch.arange(0, x.size(0), dtype=torch.long,
        device=edge_index.device)
    loop_index = loop_index.unsqueeze(0).repeat(2, 1)
    edge_index = torch.cat([edge_index, loop_index], dim=1)

    if is_node_attention == True:
        x = self.linear_nodes(x)
        rel_features = self.linear_edges(rel_features)

    x1 = (x, x)
    aggr_out = self.propagate(edge_index, x=x1, rel_features =
        new_rel_features, is_node_attention = is_node_attention,
        is_merged = False) #[N, emb_dim]

    if is_node_attention:
        merged_features = self.propagate(edge_index, x=x1,
            rel_features = new_rel_features, is_node_attention =
            is_node_attention, is_merged = True) #[N, emb_dim]
        merged_features = self.merged_mlp(merged_features)
    ...
```

The forward step, after adding a self-loop to each node, performs the initial transformation on node and edge features and invokes the propagate method of the MessagePassing class, which automatically invokes the message() and update() methods to start the message exchange process and update node features.

For the node attention block, the propagate method is called twice, one to produce node features for the next EGAT layer, the other to compute edge-integrated node features used in the final merge layer as the final output of the model.

```
def message(self, edge_index, x_i, x_j, rel_features, is_merged):
    #[E, n_heads, emb_dim * 3 / n_heads]
    concat = torch.cat((x_i, x_j, rel_features), dim =
        1).view(rel_features.shape[0], self.head_count, -1)
    #[E, n_heads , 1]
    e = self.attn_nodes(concat)
    # [E, n_heads , 1]
    e = self.activation(e)
    #[E,n_heads]
    e = e.sum(dim = 2)
    alpha = softmax(e, src_node_index)

    if is_merged:
        merged_concat = torch.cat((x_j, rel_features), dim =
            1).view(rel_features.shape[0], self.head_count, -1)
        merged_features = (merged_concat * alpha.view(-1,
            self.head_count, 1))
        return merged_features.view(-1, self.emb_dim * 2)

    out = x_j.view(rel_features.shape[0], self.head_count, -1) *
        alpha.view(-1, self.head_count, 1)
    return out.view(-1, self.emb_dim)
```

The message() method constructs messages to node i for each edge in (i, j) and can take any argument which was initially passed to propagate(). In addition, features can be automatically mapped to the respective nodes i and j by appending $\_i$ or $\_j$ to the variable name (in this case x_i and x_j). For this reason, the two tensors x_i and x_j have a shape equal to $[N\_REL, NODE\_SIZE]$, and in each position x_i represent the embedding of the source node for the given edge, while x_j the embedding of the target node.

The method produces the needed concatenation between nodes and edges for each relation. After computing, the attention scores produce the new output automatically updated for each node in the graph.
In this implementation, nodes and edges concatenation is reshaped to $[E, N\_HEADS, DIM]$; thus, independent computation of each head is implemented by adding a third dimension to the matrix, rather than instantiating numerous independent layers.

The final "if" construct produces edge-integrated node features for the final aggregation. It is computed as the standard output except that features taken into account correspond only to the concatenation between target nodes and edges.

```
i = 0
for layer in self.gnn_layers:
    _X, merged_features, nodes, edges = layer(_X, edge_index,
        edge_type, _node_type, _rel_fatures, True)
    new_nodes = torch.empty(reverse_edge_type.shape[0],
        self.hidden_size).to(edge_index.device)
    new_nodes = nodes[reverse_edge_type]
    _rel_fatures = layer(edges, reverse_edge_index,
        reverse_edge_type, edge_type, new_nodes, False )
    _X = self.activation(_X)
    _X = F.dropout(_X, self.dropout_rate, training = self.training)
    merged_features = self.activation(merged_features)
    merged_features = F.dropout(merged_features,
        self.dropout_rate, training = self.training)
    if i == 0:
        total_merged = merged_features
    else:
        total_merged = torch.div(torch.add(total_merged,
            merged_features),2)
    i = i + 1
return total_merged
```

The MessagePassing class used to instantiate the EGAT layer, in its forward step, exploits a for loop to invoke those layers. Each one is called twice. The first one is used as a node attention block, the second one as an edge attention block, and in the end, features are merged (in the example above by averaging them) to compute the final output.

4.6 Dataset preparation

For our tests we used the MedQA dataset [53], a biomedical dataset that contains a series of questions and for each to be 4 associated answers, of which only one is the correct one.
The original work provides a script capable of processing the dataset and producing a subgraph for each pair question-answer, stored using the pickle module.

4.6.1 Creation of the reversed graph

The implemented EGAT requires as input both adjacency matrices for the normal and reversed graph, for the problem mentioned in chapter 4.5.2 represented with two distinct tensors: edge_index and edge_type. The first one is a tensor with shape [2, number_of_relations] and is populated with values between 0 and the total number of nodes, while the latter is a tensor [number_of_relations] containing values between 0 and the total number of relation types.
Given $edge\_index[0][i] = x$, $edge\_index[1][i] = y$, and $edge\_type[i] = z$, nodes x and y are connected via the z-type relation.
The original script has been edited to produce both adjacency matrices.

```
#Loop through all relations
for index in range(edge_index_.shape[1]):
    #Given a relation, get the target node
    #From the source node, get all indexes with the same node
    mask = edge_index_[0] == edge_index_[1][index].item()
    indices = mask.nonzero()
    #For each of these node, since it's in common between the
        previous and the actual relation, the orginal relation and
        the new one are connected via the node in common
    for node in indices:
        edge_index_list.append(index)
        edge_index_list1.append(node.item())
        edge_type_list.append(edge_index_[1][node].item())
```

4.6.2 Creation of relation embeddings

The original solution does not take into account edge features, but only node ones, which are generated by a script using the specified language model. For this reason it was necessary to repeat the same procedure in order to generate also edge features. Node and edges embedding are written on file and then read to be used inside the GNN.

```
# Directory where storing model embeddings
modelShortName = "pubmedbert_abstract"

# Model name from HuggingFace
model =
    "microsoft/BiomedNLP-PubMedBERT-base-uncased-abstract-fulltext"

embs = []
```

```
tensors = tokenizer(relations, padding=True, truncation=True,
    return_tensors="pt")
with torch.no_grad():
    for i, j in enumerate(tqdm(relations)):
        outputs =
            bert_model(input_ids=tensors["input_ids"][i:i+1].to(device),
            attention_mask=tensors['attention_mask'][i:i+1].to(device))
        out = np.array(outputs[1].squeeze().tolist()).reshape((1, -1))
        embs.append(out)
embs = np.concatenate(embs)

filename=f"{repo_root}/data/ddb/{modelShortName}/rel_emb.npy"
os.makedirs(os.path.dirname(filename), exist_ok=True)
np.save(filename, embs)
```

4.7 Server commands

```
FROM nvcr.io/nvidia/pytorch:20.09-py3
LABEL maintainer="DISI NLU Research Group"

# Zero interaction (default answers to all questions)
ENV DEBIAN_FRONTEND=noninteractive

# Set work directory
WORKDIR /qagnn/

# Install general-purpose dependencies
RUN apt-get update -y && \
    apt-get install -y curl \
                    git \
                    bash \
                    nano \
                    ssmtp \
                    subversion && \
    apt-get autoremove -y && \
    apt-get clean -y && \
    rm -rf /var/lib/apt/lists/*
RUN pip install --upgrade pip
RUN pip install wrapt --upgrade --ignore-installed
RUN pip install gdown
RUN apt-get update && apt-get install cron && apt-get install ssmtp
    && apt-get install lsof
```

```
# Project dependencies
RUN pip install torch==1.10.2+cu113 torchvision==0.11.3+cu113
    torchaudio===0.10.2+cu113 -f
    https://download.pytorch.org/whl/cu113/torch_stable.html
RUN pip install transformers==3.4.0
RUN pip install nltk spacy==2.1.6
RUN pip install wandb==0.12.10
RUN pip install streamlit==1.6.0
RUN pip install labml-helpers

# Back to default frontend
ENV DEBIAN_FRONTEND=dialog
```

```
pip install torch-scatter -f
    https://pytorch-geometric.com/whl/torch-1.10.2+cu113.html
pip install torch-sparse -f
    https://pytorch-geometric.com/whl/torch-1.10.2+cu113.html
pip install torch-geometric -f
    https://pytorch-geometric.com/whl/torch-1.10.2+cu113.html
```

Installing Pytorch extensions (Scatter, Geometric and Sparse) caused problems if done directly within the Dockerfile, and thus during image creation. For this reason they have been removed from the DockerFile and must therefore be run manually after the container is started.

Conclusions and Future Challenges

This thesis explored the world of edge-aware GNNs, i.e., those graph neural networks that can also consider the characteristics of the relationship between nodes. The path is finished by implementing a pre-existing model, also with an innovative variant, and applying them to a task in the biomedical field, specifically to a question-answering system on the MedQA dataset [53].

The realization of this thesis was possible thanks to the following steps. First, the pre-existing solutions have been carefully analyzed and implemented through the most popular frameworks (PyTorch, PyTorch Geometric) and innovative (DGL) to understand the theoretical concepts learned during the reading.
The re-implemented model, applied to question-answering, has obtained excellent results in beating the pre-existing solution and defining the new state-of-the-art. Furthermore, the proposed model can process edge features and be used for different tasks, such as link prediction, thus introducing an additional advantage. Moreover, the innovative model we proposed has shown how a simple transformation on a graph allows a significant boost in performance without increasing the complexity of the original GAT but simply expanding the number of nodes of an additive factor. Finally, compared to the original solution used in the QA-GNN [15] paper, our models still ignore the node-ranking score and the node typology that can further increase performance.

In future developments, we plan to integrate the node ranking score and the node typology into the developed models to establish a solution similar to the original one and verify the performance deviation. It is then intended to apply orthogonal regularization to remove redundancies and thus improve the results or equalize them using fewer layers. In addition, the representation of the square adjacency matrix is not scalable. Still, it can be approximated with random kernel functions that generate orthogonal and positive vectors, thus reducing the size by at least an order of magnitude.
We also believe that the variant that uses the transformation on the graph combined with classic GAT can be further improved. It still does not take advantage of the improvements made to the solution with EGAT (removal of non-alphanumeric characters, feature generation by averaging, etc.) and has

not been tested with different configurations in terms of layers and heads. Regardless of the GNN used, QA-GNN offers many ideas for future developments, some of which have already been analyzed. For example, the reference language models can be modified, and the parameter k, indicating the k-hop, taken into account during the Knowledge Graph extraction.

In conclusion, the goal will be to improve further the models defined and, consequently, the question-answering system, and then conclude the process with the development of an application that allows users to test the work done.

Ringraziamenti

Arrivato al termine di questo percorso di studi, che mi ha cambiato come studente, lavoratore e persona, mi sento in dovere di ringraziare tante persone che hanno reso questo traguardo possibile.

In primis desidero ringraziare il mio relatore Gianluca Moro per la competenza con la quale mi ha guidato nella realizzazione di questo lavoro. Ringrazio poi il mio co-relatore Giacomo Frisoni che, con pazienza, mi ha seguito durante l'intero sviluppo della tesi, aiutandomi e fornendomi il supporto non solo tecnico, ma anche motivazionale. Senza loro la riuscita di questo lavoro sarebbe risultata impossibile.

Ringrazio i miei genitori, per avermi sempre sostenuto indipendentemente dalle mie scelte e avermi reso la persona che sono. Poter seguire un percorso senza sentire alcuna pressione gravare sulle proprie spalle mi ha permesso di godere appieno di questo viaggio, dando il massimo ogni giorno.

Non posso dimenticare i miei amici, in quanto parte integrante delle mie giornate e in grado di rendere belle anche quelle più buie.
Un ringraziamente in particolare a Gianni e Mattia, non solo amici ma compagni di studio e di progetti, senza i quali questo percorso sarebbe sicuramente risultato più complesso e meno divertente.

Non posso non ringraziare me stesso, perchè sebbene sia sempre vissuto in un ambiente stimolante e favorevole al mio studio, ho sempre dato il massimo e ho perserverato per il raggiungimento di questo obiettivo, che spero sia solo l'inizio di un nuovo bellisimo percorso.

Acknowledgments

At the end of this course of study, which has profoundly changed me as a student, a worker, and a person, I feel obliged to thank many people who have made this goal possible.

First of all, I would like to thank my supervisor Gianluca Moro for the competence with which he guided me in realizing this work. I would also like to thank my co-director Giacomo Frisoni. He patiently followed me throughout the development of the thesis, helping me and providing me with technical and motivational support. Without them, the success of this work would have been impossible.

I would like to thank my parents for always supporting me regardless of my choices and for making me the person I am. Following a path without feeling any pressure on my shoulders has allowed me to fully enjoy this journey, giving my best every day.

I can't forget my friends, as they are an integral part of my days and able to make even the grayest days beautiful.
A special thanks to Gianni and Mattia, not only friends but also study and project companions, without whom this journey would indeed have been more complex and less fun.

And above all, thanks to me. Even though I have always lived in a stimulating and favorable environment for my study, I have always tried hard to reach this goal, which I hope is only the beginning of a new beautiful path.

Bibliografia

[1] IBM Watson Explorer. `https://amr.isi.edu`. Accessed 01 Feb 2022.

[2] Meiqin Chen, Yuan Zhang, Xiaoyu Kou, Yuntao Li, and Yan Zhang. r-gat: Relational graph attention network for multi-relational graphs. *ArXiv*, abs/2109.05922, 2021.

[3] Stephan Oepen, Omri Abend, Lasha Abzianidze, Johan Bos, Jan Hajič, Daniel Hershcovich, Bin Li, Tim O'Gorman, Nianwen Xue, and Daniel Zeman. Mrp 2020: The second shared task on cross-framework and cross-lingual meaning representation parsing. pages 1–22, 01 2020.

[4] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[5] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[6] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In Aldo Gangemi, Roberto Navigli, Maria-Esther Vidal, Pascal Hitzler, Raphaël Troncy, Laura Hollink, Anna Tordai, and Mehwish Alam, editors, *The Semantic Web*, pages 593–607, Cham, 2018. Springer International Publishing.

[7] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[9] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. Heterogeneous graph attention network. In *The World Wide Web Conference*, WWW '19, page 2022–2032, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Hu Linmei, Tianchi Yang, Chuan Shi, Houye Ji, and Xiaoli Li. Heterogeneous graph attention networks for semi-supervised short text classification. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4821–4830, Hong Kong, China, November 2019. Association for Computational Linguistics.

[11] Zhifei Li, Hai Liu, Zhaoli Zhang, Tingting Liu, and Neal N. Xiong. Learning knowledge graph embedding with heterogeneous relation attention networks. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–13, 2021.

[12] Zhao Zhang, Fuzhen Zhuang, Hengshu Zhu, Zhiping Shi, Hui Xiong, and Qing He. Relational graph neural network with hierarchical attention for knowledge graph completion. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05):9612–9619, Apr. 2020.

[13] Taichi Ishiwatari, Yuki Yasuda, Taro Miyazaki, and Jun Goto. Relation-aware graph attention networks with relational position encodings for emotion recognition in conversations. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7360–7370, Online, November 2020. Association for Computational Linguistics.

[14] Ziming Wang, Jun Chen, and Haopeng Chen. Egat: Edge-featured graph attention network. In Igor Farkaš, Paolo Masulli, Sebastian Otte, and Stefan Wermter, editors, *Artificial Neural Networks and Machine Learning – ICANN 2021*, pages 253–264, Cham, 2021. Springer International Publishing.

[15] Michihiro Yasunaga, Hongyu Ren, Antoine Bosselut, Percy Liang, and Jure Leskovec. QA-GNN: Reasoning with language models and knowledge graphs for question answering. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 535–546, Online, June 2021. Association for Computational Linguistics.

[16] Takuo Hamaguchi, Hidekazu Oiwa, Masashi Shimbo, and Yuji Matsumoto. Knowledge transfer for out-of-knowledge-base entities : A graph neural

network approach. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 1802–1808, 2017.

[17] Xiaodong Wang, Zhen Liu, Nana Wang, and Wentao Fan. *Relational Metric Learning with Dual Graph Attention Networks for Social Recommendation*, pages 104–117. 05 2020.

[18] Yongji Wu, Defu Lian, Yiheng Xu, Le Wu, and Enhong Chen. Graph convolutional networks with markov random field reasoning for social spammer detection. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(01):1054–1061, Apr. 2020.

[19] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. Protein interface prediction using graph convolutional networks. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[20] Franco Scarselli, Sweah Liang Yong, Marco Gori, Markus Hagenbuchner, Ah Chung Tsoi, and Marco Maggini. Graph neural networks for ranking web pages. In *Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence*, WI '05, page 666–672, USA, 2005. IEEE Computer Society.

[21] P. Frasconi, M. Gori, and A. Sperduti. A general framework for adaptive processing of data structures. *IEEE Transactions on Neural Networks*, 9(5):768–786, 1998.

[22] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[23] Alessio Micheli. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009.

[24] Rikiya Yamashita, Mizuho Nishio, Richard K. G. Do, and Kaori Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*, 9:611 – 629, 2018.

[25] Yann LeCun, Y. Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–44, 05 2015.

[26] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael

Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge, 2015.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[28] William L. Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *IEEE Data Eng. Bull.*, 40(3):52–74, 2017.

[29] Giacomo Frisoni, Gianluca Moro, Giulio Carlassare, and Antonella Carbonaro. Unsupervised event graph representation and similarity learning on biomedical literature. *Sensors*, 22(1), 2022.

[30] G. W. Stewart. On the early history of the singular value decomposition. *SIAM Rev.*, 35(4):551–566, dec 1993.

[31] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, page 701–710, New York, NY, USA, 2014. Association for Computing Machinery.

[32] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, page 1067–1077, Republic and Canton of Geneva, CHE, 2015. International World Wide Web Conferences Steering Committee.

[33] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[34]

[35] Seong Jin Ahn and MyoungHo Kim. *Variational Graph Normalized AutoEncoders*, page 2827–2831. Association for Computing Machinery, New York, NY, USA, 2021.

[36] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman,

N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[37] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018.

[38] Hongyang Gao and Shuiwang Ji. Graph u-nets, 2019.

[39] Jiawei Zhang. Graph neural distance metric learning with graph-bert. *ArXiv*, abs/2002.03427, 2020.

[40] Peter Battaglia, Jessica Blake Chandler Hamrick, Victor Bapst, Alvaro Sanchez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andy Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Jayne Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *arXiv*, 2018.

[41] Danai Koutra, Joshua Vogelstein, and Christos Faloutsos. Deltacon: A principled massive-graph similarity function. 04 2013.

[42] Danai Koutra, Joshua T. Vogelstein, and Christos Faloutsos. Deltacon: A principled massive-graph similarity function, 2013.

[43] Andrzej Maćkiewicz and Waldemar Ratajczak. Principal components analysis (pca). *Computers Geosciences*, 19(3):303–342, 1993.

[44] Alaa Tharwat, Tarek Gaber, Abdelhameed Ibrahim, and Aboul Ella Hassanien. Linear discriminant analysis: A detailed tutorial. *Ai Communications*, 30:169–190,, 05 2017.

[45] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[46] Gentle Introduction to Graph Neural Networks and Graph Convolutional Networks. `https://perfectial.com/blog/graph-neural-networks-and-graph-convolutional-networks/`. Accessed 15 Feb 2022.

[47] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, jan 2014.

[48] Graph Attention Networks. `https://petar-v.com/GAT/`. Accessed 16 Feb 2022.

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

[50] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In Aldo Gangemi, Roberto Navigli, Maria-Esther Vidal, Pascal Hitzler, Raphaël Troncy, Laura Hollink, Anna Tordai, and Mehwish Alam, editors, *The Semantic Web*, pages 593–607, Cham, 2018. Springer International Publishing.

[51] Liyu Gong and Qiang Cheng. Exploiting edge features for graph neural networks. pages 9203–9211, 06 2019.

[52] Zhengdao Chen, Lisha Li, and Joan Bruna. Supervised community detection with line graph neural networks. In *International Conference on Learning Representations*, 2019.

[53] Di Jin, Eileen Pan, Nassim Oufattole, Wei-Hung Weng, Hanyi Fang, and Peter Szolovits. What disease does this patient have? a large-scale open domain question answering dataset from medical exams. *arXiv preprint arXiv:2009.13081*, 2020.

[54] Di Jin, Eileen Pan, Nassim Oufattole, Wei-Hung Weng, Hanyi Fang, and Peter Szolovits. What disease does this patient have? a large-scale open domain question answering dataset from medical exams. *Applied Sciences*, 11(14), 2021.

[55] Olivier Bodenreider. The unified medical language system (umls): Integrating biomedical terminology. *Nucleic acids research*, 32:D267–70, 02 2004.

[56] David Wishart, Yannick Djoumbou, An Chi Guo, Elvis Lo, Ana Marcu, Jason Grant, Tanvir Sajed, Daniel Johnson, Carin Li, Zinat Sayeeda, Nazanin Assempour, Ithayavani Iynkkaran, Yifeng Liu, Adam Maciejewski, Nicola Gale, Alex Wilson, Lucy Chin, Ryan Cummings, Diana Le, and Michael Wilson. Drugbank 5.0: A major update to the drugbank database for 2018. *Nucleic acids research*, 46, 11 2017.

[57] BiomedNLP-PubMedBERT-base-uncased-abstract-fulltext. https://huggingface.co/microsoft/BiomedNLP-PubMedBERT-base-uncased-abstract-fulltext. Accessed 20 Apr 2022.

[58] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[59] PyTorch Geometric. https://pytorch-geometric.readthedocs.io/en/latest/. Accessed 01 Apr 2022.

[60] Deep Graph Library. https://www.dgl.ai. Accessed 12 Apr 2022.

[61] The Dying ReLU Problem, Clearly Explained. https://towardsdatascience.com/the-dying-relu-problem-clearly-explained-42d0c54e0d24. Accessed 18 Mar 2022.