UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

Phan Minh Tam - Hoang Minh Thanh

GRADUATION THESIS REPORT ON LINK PREDICTION IN KNOWLEDGE GRAPHS

BACHELOR GRADUATION THESIS
STANDARD PROGRAM

Ho Chi Minh City, October 2020

UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

Phan Minh Tam - 18424059
Hoang Minh Thanh - 18424062

GRADUATION THESIS REPORT ON LINK PREDICTION IN KNOWLEDGE GRAPHS

BACHELOR GRADUATION THESIS
STANDARD PROGRAM

SUPERVISOR
MSc. Le Ngoc Thanh
Department of Computer Science

Ho Chi Minh City, October 2020

DECLARATION

We hereby declare that this is our own research work. The data and research results presented in this thesis are truthful and have not been duplicated from any other projects.

All research results presented in this thesis are entirely truthful and accurate.

Ho Chi Minh City, day ... month ... year ...

CANDIDATE

(Signature and full name)

ACKNOWLEDGEMENTS

We would like to express our sincere gratitude to MSc Le Ngoc Thanh for his dedicated guidance, for sharing his knowledge and experience, and for providing us with valuable solutions throughout the completion of this graduation thesis.

We would also like to extend our thanks to the faculty members of the Faculty of Information Technology, University of Science - Vietnam National University, Ho Chi Minh City, who have imparted invaluable knowledge to us during our academic journey.

Our gratitude also goes to the scientists and researchers whose work we have cited and built upon to complete our thesis.

Lastly, we would like to thank our families, friends, and others who have always supported and encouraged us during the process of working on this thesis.

Once again, our sincere thanks!

Table of Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF SYMBOLS AND ABBREVIATIONS

List of Symbols

Symbol	**Description**
$\mathcal{G}$	Graph
$\mathcal{G}_{\text{mono}}$	Homogeneous graph
$\mathcal{G}_{\text{hete}}$	Heterogeneous graph
$\mathcal{G}_{\text{know}}$	Knowledge graph
V, E	Set of vertices, set of edges
e, e_i	Entity, the i-th entity
r, r_k	Relation, the k-th relation
t_{ijk}, t_{ij}^k	An edge/triple
$\overrightarrow{e}, \overrightarrow{r}$	Entity embedding, relation embedding
$\langle h, r, t \rangle$	A triple of head entity, relation, tail entity
T^v, T^e	Set of vertex types, edge types
N_e, N_r	Number of entities, number of relations
N_{head}	Number of self-attention heads
$\mathbb{R}$	Set of real numbers
$\mathbf{E}, \mathbf{R}$	Entity embedding matrix, relation embedding matrix
$\mathbf{S}$	Training dataset
$*$	Convolution operation
σ	Non-linear activation function
$\mathbf{W}$	Weight matrix
$\Big\|\Big\|_{k=1}^{K}$	Concatenation from layer 1 to K
$\|\|$	Concatenation
$.^T$	Transpose
$\|\|W\|\|_2^2$	L2 normalization
$\vee, \wedge$	Disjunction (OR), conjunction (AND)
$\oplus$	Binary operation
$\cap$	Intersection
$\neg$	Negation

$\bigwedge_{i=1}^{n}$	Chain conjunction
γ	Margin
μ	Learning rate
ω	Number of convolutional layers
Ω	Convolutional kernel/filter set

GLOSSARY OF TERMS

Term	**Full Meaning**
AnyBURL	Anytime Bottom-up Rule Learning – an algorithm for learning bottom-up rules at any time
Knowledge Graph ($\mathcal{G}_{\text{know}}$)	A graph of knowledge – a collection of ground atoms or facts
ConvE / ConvKB	CNN-based models for graph embeddings – variants using embedding vectors for $\langle h, r, t \rangle$
Graph Attention Network (GAT)	Graph neural network using attention mechanisms to aggregate neighborhood information
KBGAT	GAT with relation embeddings – a GAT variant incorporating relation embeddings
Saturation (SAT)	Saturation level – the ratio of newly learned rules to existing rules

Chapter 1. INTRODUCTION

Nowadays, graphs have been applied in all aspects of life. Social network graphs (e.g., Facebook [45]) illustrate how individuals are connected to each other, the places we visit, and the information we interact with. Graphs are also used as core structures in video recommendation systems (e.g., YouTube [3]), flight networks, GPS navigation systems, scientific computations, and even brain connectivity analysis. Google's Knowledge Graph [15], introduced in 2012 [22], is a notable example of how information can be structured and utilized in knowledge graphs.

Effectively exploiting knowledge graphs provides users with deeper insight into the underlying data, which can benefit many real-world applications. However, in practice, new knowledge is continuously generated, and the acquired information is often incomplete or missing. This leads to the problem of knowledge graph completion or link prediction in knowledge graphs.

Most current approaches aim to predict a new edge connecting two existing nodes. Such methods help make the graph more complete—i.e., denser—by introducing additional connecting edges. However, these approaches primarily address the problem of completion rather than the challenge of integrating new knowledge into the graph, which remains an open question. Currently, research in knowledge graph completion follows two main directions: one is optimizing an objective function to make predictions with minimal error, as in RuDiK [35], AMIE [14], and RuleN [29], which are typically used in vertex or edge classification applications. The other approach generates a ranked list of k candidate triples, where the score reflects decreasing confidence, as seen in studies such as TransE [5] and ConvKB [48], which are commonly used in recommendation systems. Our approach follows this second direction of producing a candidate list.

Within these approaches, there are two main methodologies: rule-based systems such as AnyBURL [27], and embedding-based methods such as ConvE [11], TransE [5], and ComplEx [44]. With the goal of gaining a systematic understanding of these

methods, we chose to explore both directions in this thesis. For the rule-based approach, we selected AnyBURL [27], and for the graph embedding-based method, we chose KBAT [32], which employs attention mechanisms.

Our contribution in the AnyBURL method includes a Python implementation [1], along with two proposed strategies for adding new knowledge to the graph, which we term *online-to-offline* and *online-to-online*. The *online-to-offline* strategy extends AnyBURL by generating rules when a batch (set) of new knowledge is added. The *online-to-online* strategy generates rules immediately when a single new piece of knowledge (edge) is added.

For the embedding-based method, we present a review of attention mechanisms [46], their application in knowledge graphs via Graph Attention Networks (GATs) [47], and the KBAT model [32].

Our contribution in the deep learning approach includes a publicly available implementation and training process on GitHub [2], with both training code and model results openly provided.

[1] https://github.com/MinhTamPhan/mythesis

[2] https://github.com/hmthanh/GCAT

Chapter 2. RELATED WORK

In this section, we present the basic definitions of knowledge graphs in order to understand the task of link prediction in knowledge graphs, as well as other related research directions.

2.1 Definition of Knowledge Graphs

The basic definitions of knowledge graphs are compiled and categorized by Cai, Hongyun [7] and Goyal, Palash [16] as follows:

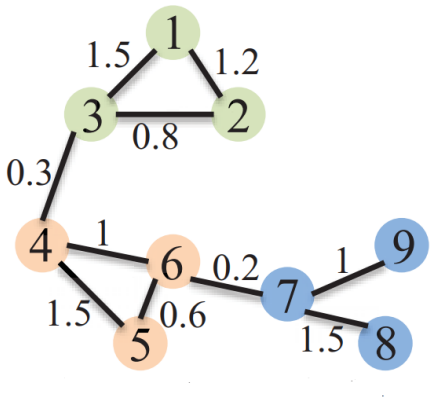


Figure 2.1: Example of an input graph

- **Definition 1 (Graph)** *$\mathcal{G} = (V, E)$, where $v \in V$ is a vertex and $e \in E$ is an edge. $\mathcal{G}$ is associated with a vertex-type mapping function $f_v : V \to T^v$ and an edge-type mapping function $f_e : E \to T^e$.*

  Here, T^v and T^e are the sets of vertex types and edge types, respectively. Each vertex $v_i \in V$ belongs to a specific type, i.e., $f_v(v_i) \in T^v$. Similarly, for $e_{ij} \in E$, $f_e(e_{ij}) \in T^e$.

- **Definition 2 (Homogeneous Graph)** *Homogeneous graph:* $\mathcal{G}_{homo} = (V, E)$ *is a graph where* $\mid T^v \mid = \mid T^e \mid = 1$. *All vertices in* $\mathcal{G}$ *belong to a single type, and all edges belong to a single type.*

- **Definition 3 (Heterogeneous Graph)** *Heterogeneous graph:* $\mathcal{G}_{hete} = (V, E)$ *is a graph where* $\mid T^v \mid > 1$ *or* $\mid T^e \mid > 1$. *That is, there is more than one type of vertex or more than one type of edge.*

- **Definition 4 (Knowledge Graph)** *Knowledge graph:* $\mathcal{G}_{know} = (V, R, E)$ *is a directed graph, where the vertex set represents entities, the relation set represents relationships between entities, and the edge set* $E \subseteq V \times R \times V$ *represents events in the form of subject-property-object triples. Each edge is a triple* $(entity_{head}, relation, entity_{tail})$ *(denoted as* $\langle h, r, t \rangle$*), expressing a relation* r *from head entity* h *to tail entity* t.

  Here, $h, t \in V$ are entities and $r \in R$ is a relation. We refer to $\langle h, r, t \rangle$ as a knowledge graph triple.

  Example: in Figure 4.7, there are two triples: ⟨Tom Cruise, born_in, New York⟩ and ⟨New York, state_of, U.S⟩. Note that entities and relations in a knowledge graph often belong to different types. Therefore, a knowledge graph can be viewed as a specific case of a heterogeneous graph.

2.2 Link Prediction in Knowledge Graphs

Link prediction, also known as knowledge graph completion, is the task of exploiting known facts (events) in a knowledge graph to infer missing ones. This is equivalent to predicting the correct tail entity in a triple $\langle h, r, ? \rangle$ (tail prediction) or the correct head entity in $\langle ?, r, t \rangle$ (head prediction). For simplicity, instead of distinguishing between head and tail prediction, we generally refer to the known entity as the *source entity* and the entity to be predicted as the *target entity*.

Most current research on link prediction in knowledge graphs is related to approaches that focus on embedding a given graph into a low-dimensional vector space. In contrast to these approaches is a rule-based method explored in [27]. Its core algorithm samples arbitrary rules and generalizes them into Horn clauses [25], then uses statistics to compute the confidence of these generalized rules. When predicting a new

link (edge) in the graph, the task is to infer whether an edge with a specific label exists between two given nodes. Many methods have been proposed to learn rules from graphs, such as in RuDiK [35], AMIE [14], and RuleN [29].

As mentioned earlier, there are two main approaches to this problem: one is optimizing an objective function to find a small set of rules that cover the majority of correct examples with minimal error, as explored in RuDiK [35]. The other approach, which we adopt in this thesis, aims to explore all possible rules and then generate a top-k ranking of candidate triples, each associated with a confidence score measured on the training set.

Our rule-based method is largely based on the *Anytime Bottom-Up Rule Learning for Knowledge Graph Completion* method [28], hereafter referred to as **AnyBURL**. As its name suggests, this method primarily focuses on completing the graph by filling in missing parts. A key limitation of this model is that when a new edge or fact is added to the graph, the entire model must be retrained. We address this issue using two strategies: the *offline-to-online* strategy, which retrains a portion of the graph only after a batch of new edges is added; and the *online-to-online* strategy, which immediately retrains the affected portion of the graph whenever a new edge is added.

In the deep learning branch of approaches, many successful techniques from image processing and natural language processing have been applied to knowledge graphs, such as Convolutional Neural Networks (CNNs [24]), Recurrent Neural Networks (RNNs [20]), and more recently, Transformers [50] and Capsule Neural Networks (CapsNets [39]). In addition, other techniques such as random walks and hierarchical structure-based models have also been explored. The common advantage of these deep learning methods on knowledge graphs is their ability to automatically extract features and generalize complex graph structures based on large amounts of training data. However, some methods focus mainly on grid-like structures and fail to preserve the spatial characteristics of knowledge graphs.

The attention mechanism, particularly the multi-head attention layer, has been applied to graphs through the Graph Attention Network (GAT [47]) model, which aggregates information about an entity based on attention weights from its neighboring entities. However, GAT lacks integration of relation embeddings and the embeddings of an entity's neighbors—components that are crucial for capturing the role of each entity. This limitation has been addressed in the work *Learning Attention-based Embeddings*

for Relation Prediction in Knowledge Graphs (**KBAT** [32]), which we adopt as the foundation for our study.

The attention mechanism is currently one of the most effective (state-of-the-art) deep learning structures, as it has been proven to substitute any convolution operation [9]. Moreover, it serves as a core component in leading models for natural language processing, such as Megatron-LM [40], and image segmentation, such as HRNet-OCR (Hierarchical Multi-Scale Attention [42]). Some recent works [10] have proposed interesting improvements based on the attention mechanism. However, these advancements have not yet been applied to knowledge graphs, which motivates us to adopt this family of methods to integrate the latest innovations into knowledge graph modeling.

2.3 Research Areas in Knowledge Graphs

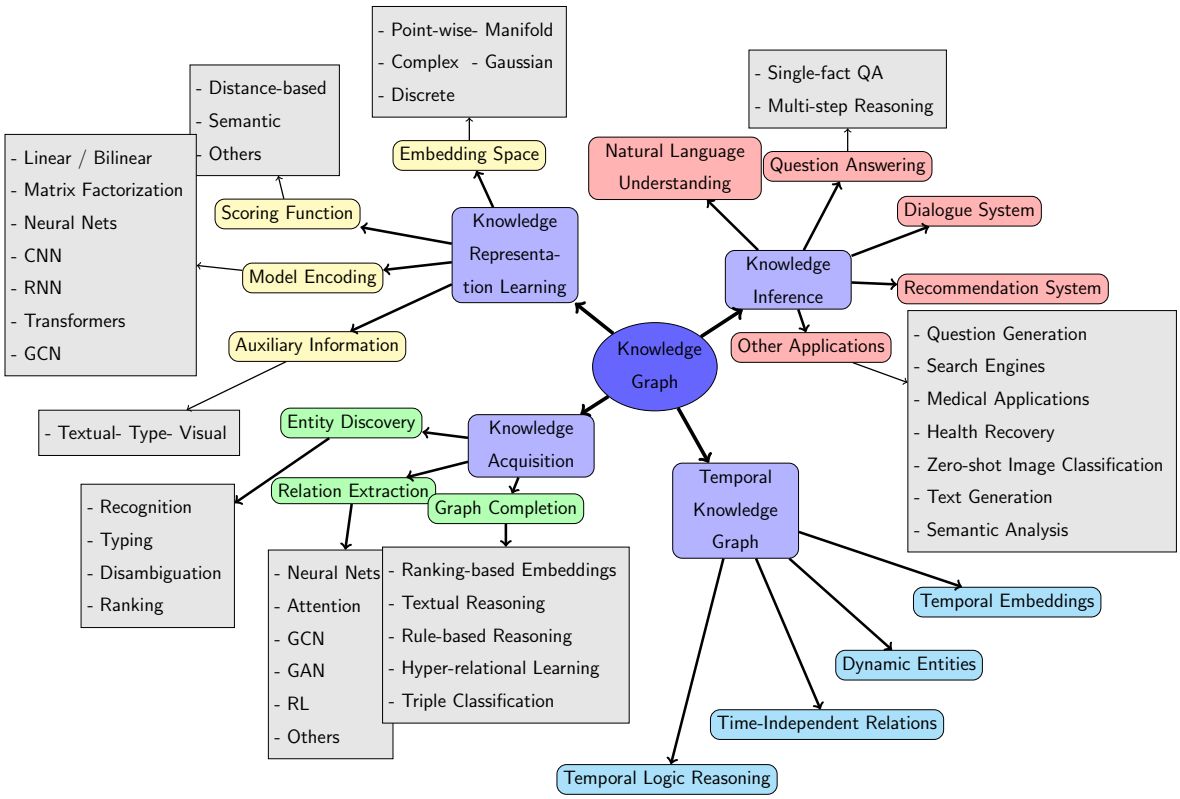


Figure 2.2: A taxonomy of research areas in knowledge graphs

Knowledge representation has a long-standing history in logic and artificial intelligence. In the context of knowledge graphs, four major research areas have been categorized and summarized in the survey [22], including: Knowledge Representation

Learning, Knowledge Acquisition, Temporal Knowledge Graphs, and Knowledge-aware Applications. All research categories are illustrated in Figure 2.2.

Knowledge Representation Learning

Knowledge representation learning is an essential research topic in knowledge graphs that enables a wide range of real-world applications. It is categorized into four subgroups:

- *Representation Space* focuses on how entities and relations are represented in vector space. This includes point-wise, manifold, complex vector space, Gaussian distribution, and discrete space embeddings.
- *Scoring Function* studies how to measure the validity of a triple in practice. These scoring functions may be distance-based or similarity-based.
- *Encoding Models* investigate how to represent and learn interactions among relations. This is currently the main research direction, including linear or non-linear models, matrix factorization, or neural network-based approaches.
- *Auxiliary Information* explores how to incorporate additional information into embedding models, such as textual, visual, and type information.

Knowledge Acquisition

Knowledge acquisition focuses on how to extract or obtain knowledge based on knowledge graphs, including knowledge graph completion, relation extraction, and entity discovery. Relation extraction and entity discovery aim to extract new knowledge (relations or entities) into the graph from text. Knowledge graph completion refers to expanding an existing graph by inferring missing links. Research directions include embedding-based ranking, relation path reasoning, rule-based reasoning, and hyper-relational learning.

Entity discovery tasks include entity recognition, disambiguation, typing, and ranking. Relation extraction models often employ attention mechanisms, graph convolutional networks (GCNs), adversarial training, reinforcement learning (RL), deep learning, and transfer learning, which is the foundation of the method proposed in our work.

In addition, other major research directions in knowledge graphs include **temporal knowledge graphs** and **knowledge-aware applications**. Temporal knowledge

graphs incorporate temporal information into the graph to learn temporal representations. Knowledge-aware applications include natural language understanding, question answering, recommendation systems, and many other real-world tasks where integrating knowledge improves representation learning.

Chapter 3. RULE-BASED METHOD

In this chapter, we describe how the problem is reformulated using the rule-based approach AnyBURL, including the rule (path) sampling algorithm and the rule generalization algorithm used to store learned knowledge in the model. We also present our improvements to the training process when new knowledge (edges) is added to the graph.

3.1 Horn Clauses

In mathematical logic, an **atomic formula** [19], also simply called an **atom**, is a formula that contains no logical connectives such as conjunction ($\wedge$), disjunction ($\vee$), or biconditional ($\Leftrightarrow$). It is a formula with no proper subformulas—meaning that an atom cannot be decomposed into smaller atoms. Thus, atomic formulas are the simplest expressions used to construct logical rules. Compound formulas are formed by combining atomic formulas using logical connectives.

A **literal** [6] is either an atomic formula or the negation of one. This concept primarily arises in classical logic theory. Literals are classified into two types: A **positive literal** is simply an atomic formula (e.g., x). A **negative literal** is the negation of an atomic formula (e.g., $\neg x$). Whether a literal is considered positive or negative depends on its defined form.

A clause is either a single literal or a disjunction of two or more literals. In **Horn form**, a clause contains at most one positive literal. Note: Not all propositional logic formulas can be converted into Horn form. A clause with no literals is sometimes referred to as a *unit clause*, and a unit clause without variables is often called a *fact* [25]. An atomic formula is referred to as a *ground* or *ground atom* if it is constructed entirely from unit clauses. All possible ground atoms that can be formed from a set of function and predicate symbols make up the Herbrand base for those symbols [1].

3.2 Definition of Logical Graph Language

Unlike general definitions of knowledge graphs commonly used in graph embedding methods, our rule-based approach treats the graph as a formal language. Below are the formal language definitions of the knowledge graph.

A knowledge graph $\mathcal{G}_{\text{know}}$ is defined over a vocabulary $\langle \mathbb{C}, \mathbb{R} \rangle$, where $\mathbb{C}$ is the set of constants and $\mathbb{R}$ is the set of binary predicates. Then, $\mathcal{G}_{\text{know}} = \{r(a, b) \mid r \in \mathbb{R}, a, b \in \mathbb{C}\}$ is the set of *ground atoms* or *facts*. A binary predicate is referred to as a relation, and a constant (or referenced constant) is referred to as an entity, corresponding to a data entry in the training set. In what follows, we use lowercase letters for constants and uppercase letters for variables. This is because we do not learn arbitrary Horn rules; instead, we focus only on those rule types that can be generalized as discussed below.

We define a rule as $h(c_0, c_n) \leftarrow b_1(c_0, c_1), \ldots, b_n(c_n, c_{n+1})$, which is a path of ground atoms of length n. Here, $h(\ldots)$ is referred to as the *head atom*, and $b_1(c_0, c_1), \ldots, b_n(c_n, c_{n+1})$ are referred to as the *body atoms*. We distinguish the following three types of rules:

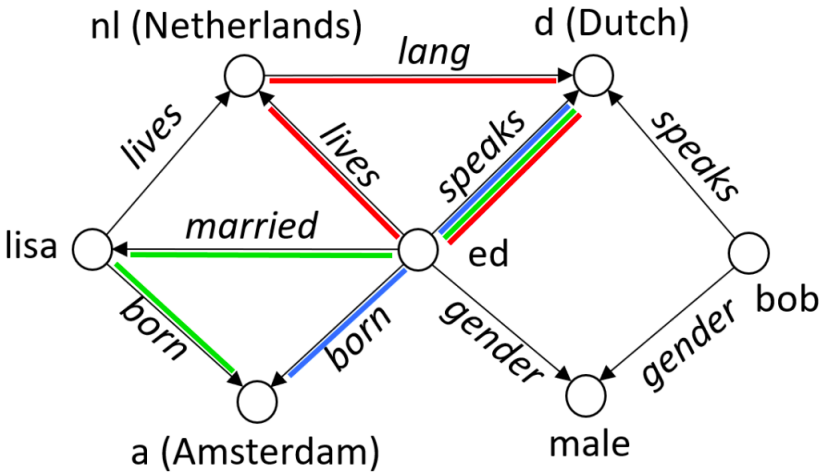
- *Binary rules* ($\mathbf{B}$): Rules in which the head atom contains two variables. - *Unary rules ending in a dangling node* ($\mathbf{U_d}$): Rules where the head atom contains only one variable, and the rule ends in a body atom that contains only variables (no constants). - *Unary rules ending in a constant* ($\mathbf{U_c}$): Rules where the head atom also contains only one variable, but the rule ends with an atom that may link to an arbitrary constant. If that constant matches the constant in the head atom, the rule forms a cyclic path.

$$
\begin{aligned}
B: &\quad h(A_0, A_n) \leftarrow \bigwedge_{i=1}^{n} b_i(A_{i-1}, A_i) \\
U_d: &\quad h(A_0, c) \leftarrow \bigwedge_{i=1}^{n} b_i(A_{i-1}, A_i) \\
U_c: &\quad h(A_0, c) \leftarrow \bigwedge_{i=1}^{n-1} b_i(A_{i-1}, A_i) \wedge b_n(A_{n-1}, c')
\end{aligned}
\tag{3.1}
$$

We refer to rules of these types as path rules because the body atoms (the part after the $\leftarrow$ symbol) form a path. Note that this also includes variants of rules where the variables are reversed within the atoms. Given a knowledge graph $\mathcal{G}_{\text{know}}$, a path of length n is a sequence of n triples $p_i(c_i, c_{i+1})$ where either $p_i(c_i, c_{i+1}) \in \mathcal{G}_{\text{know}}$ or $p_i(c_{i+1}, c_i) \in \mathcal{G}_{\text{know}}$, with $0 \leq i \leq n$. The abstract rule patterns presented above are

considered to have length n, as their body atoms can be instantiated into a path of length $n-1$. For example, in Figure 3.1 [1], when sampling paths of length 3, we can obtain the following two rules: the rule marked in green and the rule marked in red.

$$\text{(green)} \quad speaks(ed, d) \leftarrow married(ed, lisa) \wedge born(lisa, a)$$
$$\text{(red)} \quad speaks(ed, d) \leftarrow lives(ed, nl) \wedge lang(nl, d)$$



Source: adapted from Anytime Bottom-Up Rule Learning

Figure 3.1: Example of a knowledge graph

In addition, rules of type B and U_c are also referred to as closed-path rules. These are utilized by the AMIE model, described in [13, 14]. Rule U_d is considered an open rule or an acyclic path rule, since A_n is a variable that appears only once. For example:

$$speaks(X, Y) \leftarrow lives(X, Y) \quad (1)$$
$$lives\_in\_city(X, Y) \leftarrow lives(X, A), within(Y, A) \quad (2)$$
$$gen(X, female) \leftarrow married(X, A), gen(A, male) \quad (3)$$
$$profession(X, actor) \leftarrow acted\_in(X, A) \quad (4)$$

Rule (1) is a **B**-type rule (binary rule). This rule states that a person (entity) X speaks language Y if person X lives in country Y. Clearly, this is a general rule: whenever entity X has an edge to entity Y labeled *lives*, we can infer the existence of another edge labeled *speaks* between X and Y.

Rules (2) and (3) are both U_c-type rules. Rule (2) states that a person X lives in city Y if person X lives in country A and city Y is located in country A. Rule (3)

[1] http://web.informatik.uni-mannheim.de/AnyBURL/2019-05/meilicke19anyburl.pdf

states that a person X is female if they are married to person A and person A is male. In rule (3), there is no cycle formed in the graph, unlike in rule (2), where node Y is repeated both in the *head atom* and as the final node in the *body atoms*.

Rule (4) is a U_d-type rule, which states that a person X is an actor if they acted in a movie A.

All rules under consideration are filtered based on a score called the rule's confidence, which is computed on the training dataset. This confidence score is defined as the number of *body atom* paths that lead to the *head atom*, divided by the total number of paths that contain only the body atoms.

For example, consider the following rule: $gen(X, female) \leftarrow married(X, A), gen(A, male)$. We first count all entity pairs that satisfy the relations $married(X, A), gen(A, male)$—this is the number of paths containing the body atoms. Then we count how many of those pairs also satisfy the inferred relation $gen(X, female)$; this is the number of body atom paths that lead to the head atom. The confidence score of the rule is the ratio of the latter to the former.

3.3 AnyBURL Algorithm

In this section, we describe the core algorithm of the AnyBURL method, as originally introduced in [27], as well as our two extended algorithms designed to handle situations where the graph is incrementally updated with one or more new facts (edges). Additionally, we briefly describe how rules are initialized and how rule confidence is computed using sampling on the training set, including the issue of confidence estimation during prediction when sampling is used.

3.3.1 AnyBURL

Algorithm 1 Anytime Bottom-up Rule Learning

```
1:  procedure AnyBURL(G_know, S, SAT, Q, TS)
2:      n = 2
3:      R = ∅
4:      loop
5:          R_s = ∅
6:          start = currentTime()
7:          repeat
8:              p = samplePath(G_know, n)
9:              R_p = generateRules(p)
10:             for r ∈ R_p do
11:                 score(r, s)
12:                 if Q(r) then
13:                     R_s = R_s ∪ {r}
14:         until currentTime() > start + ts
15:         R'_s = R_s ∩ R
16:         if | R' | / | R |> SAT then
17:             n = n + 1
18:     R = R_s ∩ R
        return R
```

The input of the algorithm consists of $\mathcal{G}_{\text{know}}, S, SAT, Q, TS$. The output is the set R of learned rules. Here, $\mathcal{G}_{\text{know}}$ is a knowledge graph derived from the training dataset. S is a parameter indicating the sample size used during each sampling iteration on the training data for confidence computation. SAT denotes the saturation level of the rules generated in each iteration; this saturation is calculated based on the number of **new** rules learned in the current iteration relative to the total number of rules already learned. If this value is below the saturation threshold, we consider that there is still potential to discover rules of length n. Otherwise, we increase the rule length and continue the rule mining process. Q is a threshold used to determine whether a newly generated rule should be added to the result set. TS indicates the total learning time of the algorithm.

We start with $n = 2$, which corresponds to rules of path length 2, since a valid path rule requires at least one literal in the head atom and one in the body atoms. In the rule sampling step (*samplePath*), we simply select a random node in the graph, traverse all possible paths from that node with length n, and then randomly select one of the traversed paths.

3.3.2 Generate Rules

Algorithm 2 Generate Rules(p)

```
1:  procedure GENERATE_RULES(P)
2:      generalizations = ∅
3:      is_binary_rule = random.choices([true, false])
4:      if is_binary_rule then
5:          replace_all_head_by_variables(p)
6:          replace_all_tail_by_variables(p)
7:          add(generalizations, p)
8:      else:
9:          replace_all_head_by_variables(p)
10:         add(generalizations, p)
11:         replace_all_tail_by_variables(p)
12:         add(generalizations, p)
        return generalizations
```

In this algorithm, we substitute constants into the head and tail of all path rules from the sampled rule in the previous step if the rule to be learned is a binary rule. Otherwise, we substitute either the head or the tail and then add the rule to the return set. We then sample a set of rules from the training set and compute their confidence scores as described in Subsection 3.3.2. To reduce computational cost, we choose to sample from the training set for this calculation.

When making predictions for rule candidates, we recompute confidence by incorporating an estimated number of incorrect rules not observed during sampling. For our model, after experimenting with the parameter in the range $[5, 10]$, we found that this yields the best results.

3.4 Extended AnyBURL Algorithm

3.4.1 Algorithm 3: Offline-to-Online Learning

Algorithm 3 BatchAnyBURL Learning batch size

1: **procedure** BATCHANYBURL($\mathcal{G}_{know}$, SAT, Q, TS, BATCH_EDGE)
2: $is\_connected = add(\mathcal{G}_{\text{know}}, batch\_edge)$
3: **if** $is\_connected$ **then**
4: $G' = \mathcal{G}_{\text{know}} \oplus batch\_edge$
5: **else**
6: $G' = batch\_edge$
7: $n = 2$
8: $R = \emptyset$
9: **loop**
10: $R_s = \emptyset$
11: $start = currentTime()$
12: **repeat**
13: $p = samplePath(G', n)$
14: $R_p = generateRules(p)$
15: **for** $r \in R_p$ **do**
16: $score(r, G')$
17: **if** $Q(r)$ **then**
18: $R_s = R_s \cup \{r\}$
19: **until** $currentTime() > start + ts$
20: $R'_s = R_s \cap R$
21: **if** $\mid R' \mid / \mid R \mid > SAT$ **then**
22: $n = n + 1$
23: $R = R_s \cap R$
return R

This algorithm is our proposed extension to avoid retraining the entire model when a new set of knowledge is added to the graph. When new knowledge is added, we first check whether it is connected to the existing knowledge in the graph (i.e., connectivity). If it is, we perform the $\oplus$ operation by combining all elements in *batch_edge* with the

connected components in the graph, up to a path length of 5. If there is no connectivity, we use all elements in *batch_edge* and repeat the steps of the Anytime Bottom-up Rule Learning algorithm.

3.4.2 Algorithm 4: Online-to-Online Learning

Algorithm 4 EdgeAnyBURL

procedure EDGEANYBURL($\mathcal{G}_{\text{KNOW}}$, S, SAT, Q, TS, EDGE)
  $is\_connected = add(\mathcal{G}_{\text{know}}, edge)$
  $R = \emptyset$
  if $is\_connected$ **then**
    $n = 2$
    $R_s = \emptyset$
    repeat
      $p = samplePath(edge, n)$
      $R_p = generateRules(p)$
      for $r \in R_p$ **do**
        $score(r, s)$
        if $Q(r)$ **then**
          $R_s = R_s \cup \{r\}$
    until $currentTime() > start + ts$
    $R'_s = R_s \cap R$
    if $\mid R' \mid / \mid R \mid > SAT$ **then**
      $n = n + 1$
    $R = R_s \cap R$
  return R

This algorithm is a complementary component to Algorithm 3. We refer to it as online-to-online because when a new edge (i.e., new knowledge) is added to the graph, we immediately perform learning on the path rules related to that edge—unlike in Algorithm 3, where learning is triggered only after a sufficient amount of new knowledge has been added.

Chapter 4. DEEP LEARNING-BASED METHODS

In this chapter, we present Knowledge Graphs and describe the task of Graph Embedding, providing an overview of current graph embedding techniques. We will revisit the attention mechanism and explain how it is applied to knowledge graphs through the Graph Attention Network (GAT) model [47]. Additionally, we present an improved method based on the graph attention model—KBGAT [32]—which incorporates relation information and neighboring relations.

4.1 Graph Embedding

In the real world, representing entities and relations as vectors can be intuitively understood as the process of mapping features and attributes of an object into a lower-dimensional space, with each component representing a specific unit-level feature.

For example, we know Donald Trump is 1.9 meters tall and has a wife named Melania. Thus, we could represent the entity "Donald Trump" as a vector:

$$\overrightarrow{e_{\text{Trump}}} = [1.9_{\text{height}}, 0_{\text{area}}, 1_{\text{wife is Melania}}, 0_{\text{wife is Taylor}}].$$

For features that cannot be measured or have no value (e.g., `.area`), we assign 0. For categorical features without magnitude (e.g., `.wife`), we represent them using probabilities of unit features (e.g., `.wife is Melania`, `.wife is Taylor`). Therefore, any object in the real world can be *embedded* as a vector in an interpretable way.

To understand graph embedding techniques, we begin with several fundamental definitions:

- **Definition 5 (First-Order Proximity)** *First-order proximity between vertex v_i and vertex v_j is the edge weight $A_{i,j}$ of the edge e_{ij}.*

  Two vertices are more similar if they are connected by an edge with a higher weight. Thus, the first-order proximity between v_i and v_j is denoted as $s_{ij}^{(1)} =$

$A_{i,j}$. Let $s_i^{(1)} = \left[s_{i1}^{(1)}, s_{i2}^{(1)}, \ldots, s_{i|V|}^{(1)}\right]$ represent the first-order proximities between v_i and other vertices.

Using the graph in Figure 2.1 as an example, the first-order proximity between v_1 and v_2 is the weight of edge e_{12}, denoted as $s_{12}^{(1)} = 1.2$. The vector $s_1^{(1)}$ records the edge weights connecting v_1 to all other vertices in the graph, i.e.,

$$s_1^{(1)} = \left[0, 1.2, 1.5, 0, 0, 0, 0, 0, 0\right].$$

- **Definition 6 (Second-Order Proximity)** *Second-order proximity $s_{ij}^{(2)}$ between vertex v_i and v_j is defined as the similarity between v_i''s first-order neighborhood vector $s_i^{(1)}$ and v_j''s vector $s_j^{(1)}$.*

  For example, in Figure 2.1, the second-order proximity $s_{12}^{(2)}$ is the similarity between $s_1^{(1)}$ and $s_2^{(1)}$. As introduced above:

$$s_1^{(1)} = \left[0, 1.2, 1.5, 0, 0, 0, 0, 0, 0\right], \quad s_2^{(1)} = \left[1.2, 0, 0.8, 0, 0, 0, 0, 0, 0\right].$$

  We compute the cosine similarity:

$$s_{12}^{(2)} = \cos(s_1^{(1)}, s_2^{(1)}) = 0.43, \quad s_{15}^{(2)} = \cos(s_1^{(1)}, s_5^{(1)}) = 0.$$

  We observe that the second-order proximity between v_1 and v_5 is 0 because they share no common 1-hop neighbors. v_1 and v_2 share a common neighbor v_3, thus their second-order proximity $s_{12}^{(2)}$ is greater than 0.

  Higher-order proximities can be defined similarly. For example, the k-th order proximity between v_i and v_j is the similarity between $s_i^{(k-1)}$ and $s_j^{(k-1)}$.

- **Definition 7 (Graph Embedding)** *Given a graph input $\mathcal{G} = (V, E)$ and a predefined embedding dimension d where $d \ll |V|$, the graph embedding problem is to map $\mathcal{G}$ into a d-dimensional space while preserving as much graph property information as possible. These properties can be quantified using proximity measures such as first-order and higher-order proximity. Each graph is represented either as a d-dimensional vector (for the entire graph) or a set of d-dimensional vectors where each vector encodes a part of the graph (e.g., node, edge, substructure).*

Graph embedding is the process of transforming graph features into vectors or sets of low-dimensional vectors. The more effective the embedding, the higher the accuracy in subsequent graph mining and analysis tasks. The biggest challenge in graph embedding depends on the problem setting, which includes both the embedding input and output, as illustrated in Figure 4.1.

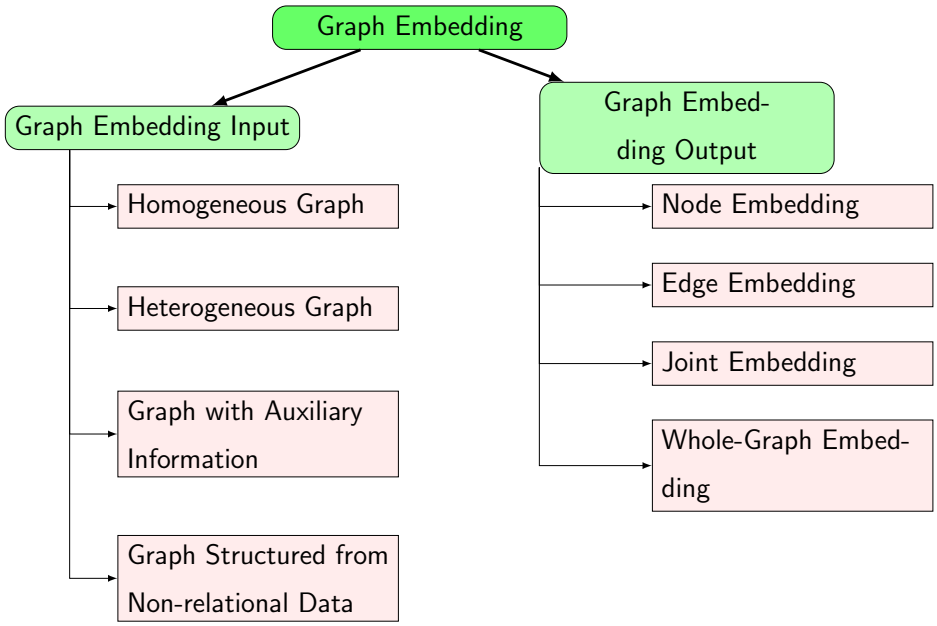


Figure 4.1: Graph Embedding Techniques

Based on the embedding input, we categorize the surveyed methods in [7] as follows: Homogeneous Graph, Heterogeneous Graph, Graph with Auxiliary Information, and Graph Constructed from Non-relational Data.

Different types of embedding inputs preserve different information in the embedding space and therefore pose different challenges for the graph embedding problem. For example, when embedding a graph with only structural information, the connections between nodes are the primary target to preserve. However, for graphs with node labels or entity attribute information, auxiliary information provides additional context for the graph and can therefore also be considered during the embedding process. Unlike embedding input, which is fixed and provided by the dataset, the embedding output is task-specific.

For instance, the most common embedding output is **node embedding**, which represents each node as a vector that reflects similarity between nodes. Node embeddings are beneficial for node-related tasks such as node classification, node clustering, etc.

However, in some cases, the tasks may involve more fine-grained graph components such as node pairs, subgraphs, or the entire graph. Therefore, the first challenge of embedding is to determine the appropriate type of embedding output for the application of interest. Four types of embedding outputs are illustrated in Figure 2.1, including: **Node Embedding** (4.2), **Edge Embedding** (4.3), **Hybrid Embedding** (4.4), and **Whole-Graph Embedding** (4.5). Different output granularities have distinct criteria and present different challenges. For example, a good node embedding retains similarity with its neighbors in the embedding space. Conversely, a good whole-graph embedding represents the entire graph as a vector that preserves graph-level similarity.

4.1.1 Graph Embedding Problem Settings

While the input is determined by the type of information to be preserved, the output varies depending on the downstream graph mining task. Therefore, we discuss in more detail the embedding methods based on the type of output required by the embedding problem.

Node Embedding

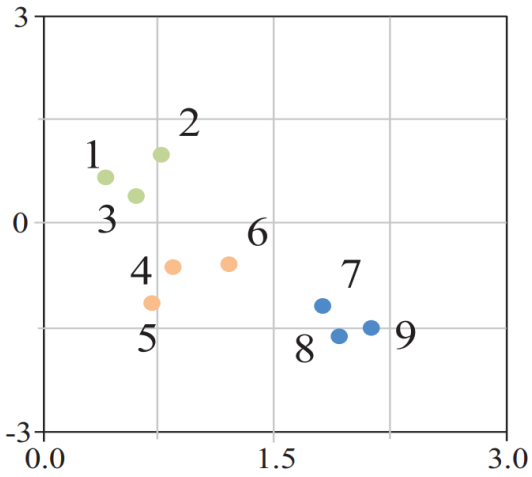


Figure 4.2: Node embedding with each vector representing node features

Node embedding represents each node as a low-dimensional vector. Nodes that are *close* in the graph have similar vector representations. The difference among graph embedding methods lies in how they define the *closeness* between two nodes. First-order proximity (Definition 5) and second-order proximity (Definition 6) are two commonly used metrics to measure pairwise node similarity. Higher-order proximity has also been

explored to some extent. For example, capturing k-step (k = 1, 2, 3, ⋯) neighborhood relationships during embedding is discussed in the study by Cao, Shaosheng [8].

Edge Embedding

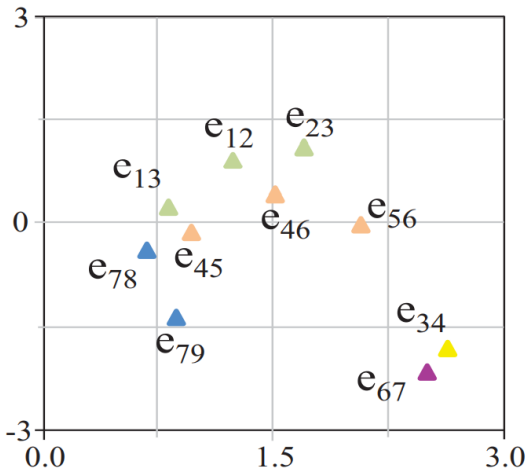


Figure 4.3: Edge embedding with each vector representing edge features

In contrast to node embedding, edge embedding aims to represent an edge as a low-dimensional vector. Edge embedding is useful in two cases:

First, in knowledge graph embedding. Each edge is a triple $\langle h, r, t \rangle$ (Definition 4). The embedding is learned to preserve the relation r between h and t in the embedding space so that a missing entity or relation can be accurately predicted given the other two components in $\langle h, r, t \rangle$.

Second, some methods embed a pair of nodes as a vector to compare it with other node pairs or to predict the existence of a link between the two nodes. Edge embedding benefits graph analyses that involve edges (node pairs), such as link prediction, relation prediction, and knowledge-based entity inference.

Hybrid Embedding

Hybrid embedding refers to embedding combinations of different graph components, e.g., node + edge (i.e., substructure), or node + parts. Substructure or part embeddings can also be derived by aggregating individual node and edge embeddings. However, such *indirect* approaches are not optimized to capture the structure of the graph. Moreover, node and part embeddings can reinforce each other. Node embeddings improve by learning from high-order neighborhood attention, while part embeddings become more accurate due to the collective behavior of their constituent nodes.

Whole-Graph Embedding

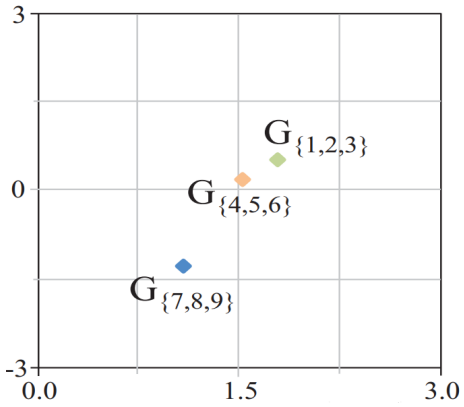


Figure 4.4: Embedding a graph substructure

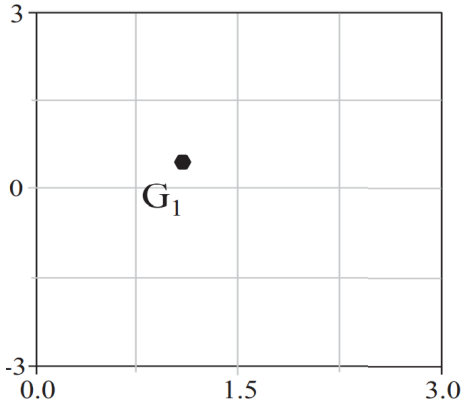


Figure 4.5: Whole-graph embedding

Whole-graph embedding is typically used for small graphs such as proteins, molecules, etc. In this case, an entire graph is represented as a vector, and similar graphs are embedded close to each other. Whole-graph embedding is useful for graph classification tasks by providing a simple and effective solution for computing graph similarity. To balance embedding time (efficiency) and information retention (expressiveness), **hierarchical graph embedding** [31] introduces a hierarchical embedding framework. It argues that accurate understanding of global graph information requires processing substructures at multiple scales. A graph pyramid is formed where each level is a coarsened graph at a different scale. The graph is embedded at all levels and then concatenated into a single vector. Whole-graph embedding requires collecting features from the entire graph, thus is generally more time-consuming than other settings.

4.1.2 Graph Embedding Techniques

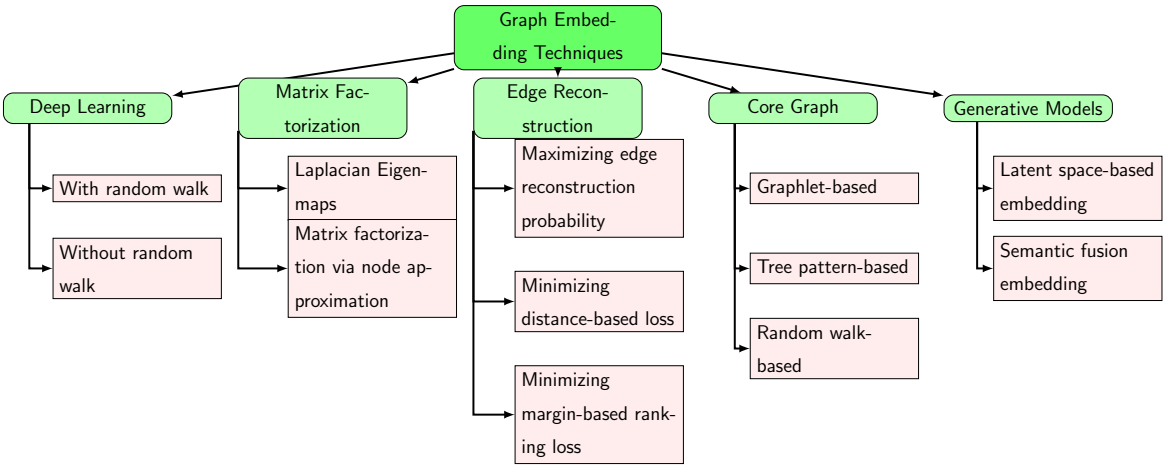


Figure 4.6: Graph Embedding Techniques

In this section, we categorize graph embedding methods based on the techniques used. As previously stated, the goal of graph embedding is to represent a graph in a lower-dimensional space while preserving as much of the original graph information as possible. The main differences among embedding techniques lie in how they define which intrinsic graph properties should be preserved. Since the primary focus of our work is on graph embedding methods based on deep learning, we provide only brief overviews of the other method categories.

Deep Learning

In this section, we present in detail the research directions of deep learning tech-

niques, including: using random walks and not using random walks. Deep learning techniques are widely used in graph embedding due to their speed and efficiency in automatically capturing features. Among these deep learning-based methods, three types of input-based graph settings (excluding graphs constructed from non-relational data) and all four output types (as shown in Figure 4.1) can adopt deep learning approaches.

Deep learning techniques with random walks

In this category, the second-order proximity (Definition 6) in the graph is preserved in the embedding space by maximizing the probability of observing a vertex's neighborhood conditioned on its embedding vector. The graph is represented as a set of samples obtained via random walks, and then deep learning methods are applied to ensure the structural properties (i.e., path-based information) are preserved. Representative methods in this group include: DeepWalk [36], LINE [41], Node2Vec [17], Anonymous Walk [21], NetGAN [4], etc.

Deep learning techniques without random walks

This approach applies multi-layer learning structures effectively and efficiently to transform the graph into a lower-dimensional space. It operates over the entire graph. Several popular methods have been surveyed and presented in [38], as follows:

- **Convolutional Neural Networks (CNNs)**

  This model uses multiple convolutional layers: each layer performs convolution on the input data using low-dimensional filters. The result is a feature map, which then passes through a fully connected layer to compute the probability values. For example, **ConvE** [11]: each entity and relation is represented as a low-dimensional vector (d-dimensional). For each triple, it concatenates and reshapes the head h and relation r embeddings into a single input $[h, r]$ with shape $d_m \times d_n$. This is then passed through a convolutional layer with a filter ω of size $m \times n$, followed by a fully connected layer with weights W. The final result is combined with the tail embedding t using a dot product. This architecture can be considered a *multi-class classification* model.

  Another popular model is **ConvKB** [33], which is similar to ConvE but concatenates the three embeddings h, r, and t into a matrix $[h, r, t]$ of shape $d \times 3$. It is then passed through a convolutional layer with T filters of size 1×3, resulting in a feature map of size $T \times 3$. This is further passed through a fully connected

layer with weights **W**. This architecture can be considered a binary classification model.

- **Recurrent Neural Networks (RNNs)**

  These models apply one or more recurrent layers to analyze the entire path (a sequence of events/triples) sampled from the training set, instead of treating each event independently. For example, RSN [18] notes that traditional RNNs are unsuitable for graphs because each step only takes the relation information without considering the entity embedding from the previous step. Therefore, it fails to clearly model the transitions among entity-relation paths. To address this, they propose RSN (Recurrent Skipping Networks [18]): at each step, if the input is a relation, a hidden state is updated to reuse the entity embedding. The output is then dot-multiplied with the target embedding vector.

- **Capsule Neural Networks**

  Capsule networks group neurons into "capsules" , where each capsule encodes specific features of the input, such as representing a particular part of an image. One advantage of capsule networks is their ability to capture spatial relationships that are lost in conventional convolution. Each capsule produces feature vectors. For instance, **CapsE** [48]: each entity and relation is embedded into vectors, similar to ConvKB. It concatenates the embeddings h, r, and t into a matrix of shape $d \times 3$, then applies E convolutional filters of size 1×3, resulting in a $d \times E$ matrix. Each i-th row encodes distinct features of $h[i]$, $r[i]$, and $t[i]$. This matrix is then fed into a capsule layer, where each capsule (4.1.2) processes a column, thus receiving feature-specific information from the input triple. A second capsule layer is used to produce the final output.

- **Graph Attention Networks (GATs)**

  This category uses the attention mechanism [46], which has achieved notable success in NLP tasks. For each embedding vector, information from neighboring entities is aggregated using attention weights. These are then combined and passed through a fully connected layer with learnable weights to obtain the final embeddings. For example, GAT [47] applies multi-head attention to each training triple to generate an embedding vector. This embedding is then transformed via a weight matrix to produce a higher-dimensional vector that aggregates information

from neighboring nodes in the original triple. An improved version, KBGAT [32], incorporates the relation embedding into the attention mechanism. These methods will be discussed in detail in the subsequent sections.

- **Other methods**

  There are also other approaches, such as autoencoder-based techniques like Structural Deep Network Embedding (SDNE) [49].

Matrix Factorization

Matrix factorization-based graph embedding represents the structural characteristics of a graph (e.g., similarity or proximity between vertex pairs) in the form of a matrix and then factorizes this matrix to obtain vertex embeddings. The input for this category of methods is typically high-dimensional non-relational features, and the output is a set of vertex embeddings. There are two matrix factorization-based graph embedding methods: Graph Laplacian Eigenmaps and Node Proximity Matrix Factorization.

- *Graph Laplacian Eigenmaps*

  This approach preserves graph properties by analyzing similar vertex pairs and heavily penalizes embeddings that place highly similar nodes far apart in the embedding space.

- *Node Proximity Matrix Factorization*

  This approach approximates neighboring nodes in a low-dimensional space using matrix factorization techniques. The objective is to preserve neighborhood information by minimizing the approximation loss.

Edge Reconstruction

The edge reconstruction method builds edges based on the vertex embeddings so that the reconstructed graph is as similar as possible to the input graph. This method either maximizes the edge reconstruction probability or minimizes edge reconstruction loss. Additionally, the loss can be distance-based or margin-based ranking loss.

- *Maximize Edge Reconstruction Probability*

  In this method, a good vertex embedding maximizes the likelihood of generating observed edges in the graph. In other words, a good vertex embedding should

allow for reconstructing the original input graph. This is achieved by maximizing the generative probability of all observed edges using vertex embeddings.

- *Minimize Distance-Based Loss*

  In this approach, embeddings of neighboring nodes should be as close as possible to the observed neighboring nodes in the original graph. Specifically, vertex proximity can be measured using their embeddings or heuristically based on observed edges. The difference between these two types of proximity is then minimized to ensure consistent similarity.

- *Minimize Margin-Based Ranking Loss*

  In this approach, the edges in the input graph represent correlations between vertex pairs. Some vertices in the graph are often linked with related vertex sets. This method ensures that embeddings of related nodes are closer together than unrelated ones by minimizing a margin-based ranking loss.

Graph Kernels

Graph kernel methods represent the entire graph structure as a vector containing counts of basic substructures extracted from the graph. Subcategories of graph kernel techniques include: graphlets, subtree patterns, and random walk-based methods.

This approach is designed to embed whole graphs, focusing only on global graph features. The input is typically homogeneous graphs or graphs with auxiliary information.

Generative Models

A generative model is defined by specifying a joint distribution over input features and class labels, parameterized by a set of variables. There are two subcategories of generative model-based methods: embedding graphs into latent space and incorporating semantics for embedding. Generative models can be applied to both node and edge embeddings. They are commonly used to embed semantic information, with inputs often being heterogeneous graphs or graphs with auxiliary attributes.

- *Embedding Graphs into Latent Semantic Space*

  In this group, vertices are embedded into a latent semantic space where the distance between nodes captures the graph structure.

- *Incorporating Semantics for Embedding*

  In this method, each vertex is associated with graph semantics and should be embedded closer to semantically relevant vertices. These semantic relationships can be derived from descriptive nodes via a generative model.

Summary: Each graph embedding method has its own strengths and weaknesses, which have been summarized by Cai, Hongyun [7] and are presented in Table 4.1. The *matrix factorization* group learns embeddings by analyzing pairwise global similarities. The *deep learning* group, in contrast, achieves promising results and is suitable for graph embedding because it can learn complex representations from complex graph structures.

Table 4.1: Comparison of Advantages and Disadvantages of Graph Embedding Techniques

Method	Subcategory	Advantages	Disadvantages
Matrix Factorization	Graph Laplacian Eigenmaps	Considers global neighborhood structure	Requires large computation time and space
	Node Proximity Matrix Factorization		
Edge Reconstruction	Maximize edge reconstruction probability	Relatively efficient training	Only optimizes local information, e.g., first-order neighbors or ranked node pairs
	Minimize distance-based loss		
	Minimize margin-based ranking loss		
Graph Kernels	Based on graphlet	Efficient, considers only desired primitive structures	Substructures are not independent. Embedding dimensionality increases exponentially
	Based on subtree patterns		
	Based on random walk		

Method	Subcategory	Advantages	Disadvantages
Generative Model	Embedding into latent space	Embedding is interpretable	Difficult to tune distribution selection
	Incorporate semantics	Leverages multiple information sources	Requires a large amount of naturally labeled data
Deep Learning	With random walk	Efficient and fast. No need for manual feature extraction	Considers only local path content. Difficult to find optimal sampling strategy
	Without random walk		High computational cost

Graph embedding methods based on random walk in deep learning have lower computational cost compared to those using full deep learning models. Traditional methods often treat graphs as grids; however, this does not reflect the true nature of graphs. In the *edge reconstruction* group, the objective function is optimized based on observed edges or by ranking triplets. While this approach is more efficient, the resulting embedding vectors do not account for the global structure of the graph. The *graph kernel* methods transform graphs into vectors, enabling graph-level tasks such as graph classification. Therefore, they are only effective when the desired primitive structures in a graph can be enumerated. The *generative model* group naturally integrates information from multiple sources into a unified model. Embedding a graph into a latent semantic space produces interpretable embedding vectors using semantics. However, assuming the modeling of observations using specific distributions can be difficult to justify. Moreover, generative approaches require a large amount of training data to estimate a model that fits the data well. Hence, they may perform poorly on small graphs or when only a few graphs are available.

Among these methods, deep learning-based graph embedding allows learning complex representations and has shown the most promising results. Graph attention net-

works (GATs), which are based on attention mechanisms, aggregate the information of an entity using attention weights from its neighbors relative to the central entity. We believe this research direction is aligned with studies on the relationship between attention and memory [34], where the distribution of attention determines the weight or importance of one entity relative to another. Likewise, the embedding vector representing an entity is influenced by the attention or importance of its neighboring embeddings. Therefore, this is the approach we selected among the existing graph embedding methods.

4.2 Multi-head Attention Mechanism

In 2014, the multi-head attention mechanism was introduced by Bahdanau, Dzmitry [2], but it only gained widespread popularity in 2017 through the Transformer model by Vaswani, Ashish [46]. The attention mechanism is an effective method to indicate the importance of a word with respect to other words in a sentence, and it has been shown to be a generalization of any convolution operation as reported by Cordonnier, Jean-Baptiste [9]. To understand how multi-head attention is applied to graphs, in this section we will detail the mechanism so we can better understand how it is used in link prediction tasks within knowledge graphs.

4.2.1 Attention Mechanism

The input of the attention mechanism consists of two embedding matrices $\mathbf{X} = \left\{\overrightarrow{x_1}, \overrightarrow{x_2}, ..., \overrightarrow{x_{N_x}}\right\}$ and $\mathbf{Y} = \left\{\overrightarrow{y_1}, \overrightarrow{y_2}, ..., \overrightarrow{y_{N_y}}\right\}$, where each row i^{th} or j^{th} in matrix $\mathbf{X}$ or $\mathbf{Y}$ is an embedding vector $\overrightarrow{x_i} \in \mathbb{R}^{1\times D_{\text{in}}}$, $\overrightarrow{y_j} \in \mathbb{R}^{1\times D_{\text{in}}}$.

The attention mechanism transforms input vectors of D_{in} dimensions into output vectors of $D_{\text{attention}}$ dimensions to represent the importance of each of the N_x elements x with respect to all N_y elements y. Given $\mathbf{X} \in \mathbb{R}^{N_x\times D_{\text{in}}}$ and $\mathbf{Y} \in \mathbb{R}^{N_y\times D_{\text{in}}}$ as input embedding matrices, and $\mathbf{H} \in \mathbb{R}^{N_x\times D_{\text{attention}}}$ as the output embedding matrix, the attention mechanism introduced by Vaswani et al. [46] is defined as follows:

$$\mathbf{H} = \text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\Big(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\Big)\mathbf{V} \tag{4.1}$$

where $\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \mathbf{K} = \mathbf{Y}\mathbf{W}_K, \mathbf{V} = \mathbf{Y}\mathbf{W}_V$.

The weight matrices $\mathbf{W}_Q \in \mathbb{R}^{D_{\text{in}} \times D_k}$, $\mathbf{W}_K \in \mathbb{R}^{D_{\text{in}} \times D_k}$, and $\mathbf{W}_V \in \mathbb{R}^{D_{\text{in}} \times D_{\text{attention}}}$ are used to parameterize the transformation from input embedding vectors of dimension D_{in} into output embedding vectors of dimension D_k or $D_{\text{attention}}$. The term $\mathbf{Q}\mathbf{K}^T$ represents the dot product between each vector x and all vectors y. Dividing by $\sqrt{d_k}$ normalizes the result with respect to the vector dimension k. The result is then passed through the *softmax* function to enable comparison of attention scores across different pairs. We can interpret $\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)$ as the *attention coefficients*, indicating the importance of each y with respect to each x. Finally, this is multiplied with the value matrix $\mathbf{V}$ to produce the final output embedding of dimension $D_{\text{attention}}$.

If $\mathbf{X} = \mathbf{Y}$, we are computing the importance of each element with respect to other elements in the same input matrix, which is referred to as the self-attention mechanism.

4.2.2 Multi-Head Attention

The multi-head attention mechanism is a way of combining multiple attention layers to stabilize the learning process. Similar to the standard attention mechanism above, the multi-head attention mechanism transforms the initial N_x embedding vectors of dimension D_{in} into output embedding vectors of dimension $D_{\text{multi-head}}$, aggregating information from various other nodes to provide greater stability during training. Multi-head attention stacks N_{head} attention output matrices $\mathbf{H}$, and then applies a weight matrix to transform the original embedding matrix $\mathbf{X} \in \mathbb{R}^{N_x \times D_{\text{in}}}$ into a new embedding matrix $\mathbf{X}' \in \mathbb{R}^{N_x \times D_{\text{multi-head}}}$ using the following formula:

$$\begin{aligned} \mathbf{X}' &= \left(\Big\Vert_{h=1}^{N_{\text{head}}} \mathbf{H}^{(h)} \right) \mathbf{W}^O \\ &= \left(\Big\Vert_{h=1}^{N_{\text{head}}} \text{Attention}(\mathbf{X}\mathbf{W}_Q^{(h)}, \mathbf{Y}\mathbf{W}_K^{(h)}, \mathbf{Y}\mathbf{W}_V^{(h)}) \right) \mathbf{W}^O \end{aligned} \tag{4.2}$$

Here, the weight matrices $\mathbf{W}_Q^{(h)}$, $\mathbf{W}_K^{(h)} \in \mathbb{R}^{D_{\text{in}} \times D_k}$, and $\mathbf{W}_V^{(h)} \in \mathbb{R}^{D_{\text{in}} \times D_{\text{attention}}}$ correspond to each individual attention head $h \in [N_{\text{head}}]$. The output projection matrix $\mathbf{W}^O \in \mathbb{R}^{N_{\text{head}} D_{\text{attention}} \times D_{\text{multi-head}}}$ parameterizes the transformation of the concatenated output heads into the final output embedding matrix.

At this point, we have presented the attention mechanism for computing attention scores and aggregating embedding information from neighboring vectors. In the next section, we will describe how this attention mechanism is applied to knowledge graphs.

4.3 Graph Attention Network

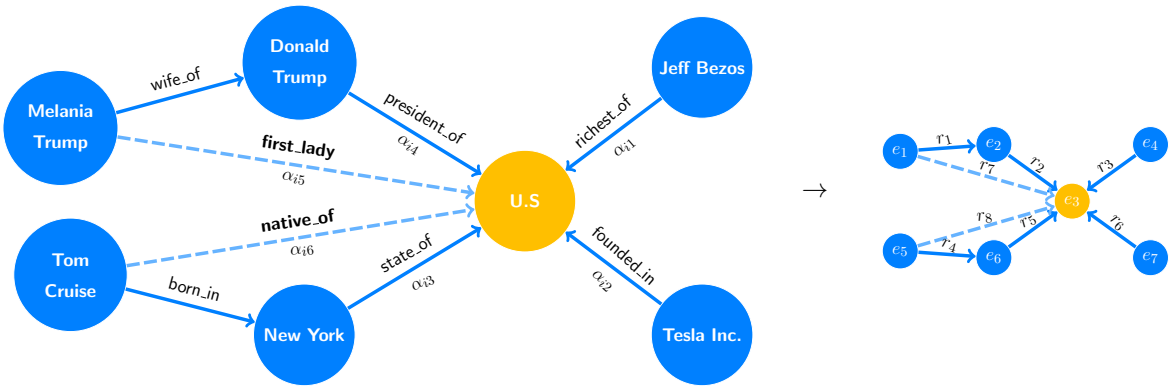


Figure 4.7: Knowledge graph and normalized attention coefficients of the entity

With the success of the *multi-head attention mechanism* in natural language processing, it has also been studied for applications in image processing models [37]. Consequently, the multi-head attention mechanism has been explored for replacing convolutional operations in knowledge graph embedding models, such as Graph Convolutional Networks (GCNs [23]). In this section, we present in detail how the attention mechanism from 4.2.1 is applied to graph embedding via the Graph Attention Network (GAT [47]) method.

The input to the *graph attention network* model is a set of embedding vectors, randomly initialized from a normal distribution, representing features of each entity: $\mathbf{E} = \left\{\overrightarrow{e_1}, \overrightarrow{e_2}, ..., \overrightarrow{e_{N_e}}\right\}$. The objective of the model is to transform this into a new output embedding matrix $\mathbf{E}'' = \left\{\overrightarrow{e''_1}, \overrightarrow{e''_2}, ..., \overrightarrow{e''_{N_e}}\right\}$ capable of aggregating embedding information from neighboring entities. Here, $\mathbf{E} \in \mathbb{R}^{N_e \times D_{\text{in}}}$ and $\mathbf{E}'' \in \mathbb{R}^{N_e \times D''}$ denote the input and output embedding matrices for the entity set, respectively. N_e is the number of entities, and D_{in}, D'' are the dimensions of the input and output embeddings.

Similar to the multi-head attention mechanism introduced in 4.2.1, the application of this mechanism to a knowledge graph follows the same logic as the *self-attention mechanism*, in which each node attends to all other nodes in the graph. However, computing attention scores between every pair of nodes in a graph is not meaningful if no relationship exists between them, and it would incur significant computational overhead. Therefore, the model applies a technique known as *masked attention*, in which all attention scores corresponding to unrelated nodes in the graph are ignored. These relevant connections are precisely defined as the first-order proximity (Definition

5) of a node in the graph. Thus, in this context, we let $\mathbf{X} = \mathbf{Y} = \mathbf{E}$ (as in 4.2.1), and the attention coefficient in the masked attention mechanism represents the importance of a neighboring node $j \in \mathcal{N}_i$ to the central node i, where $\mathcal{N}_i$ is the set of all first-order neighbors of node i (including i itself).

The application of the multi-head attention mechanism (*multi-head attention*) in 4.2 to graphs is described as follows:

$$e_{ij} = f_{\text{mask attention}}(\mathbf{W}\overrightarrow{e_i}, \mathbf{W}\overrightarrow{e_j}) \tag{4.3}$$

where e_{ij} denotes the multi-head attention coefficient of an edge (e_i, e_j) with respect to the central entity e_i in the knowledge graph $\mathcal{G}_{\text{know}}$. $\mathbf{W}$ is a weight matrix that parameterizes the linear transformation. $f_{\text{mask attention}}$ is the function applying the attention mechanism.

In the GAT model, each entity vector embedding $\overrightarrow{e_i}$ undergoes two transformation stages. The entire model consists of two transformation steps, each applying the multi-head attention mechanism as follows:

$$\overrightarrow{e_i} \xrightarrow{f^{(1)}_{\text{mask attention}}} \overrightarrow{e_i'} \xrightarrow{f^{(2)}_{\text{mask attention}}} \overrightarrow{e_i''} \tag{4.4}$$

In the first multi-head attention step ($f^{(1)}_{\text{mask attention}}$), the model aggregates information from neighboring entities and stacks them to produce vector $\overrightarrow{e_i'}$, where $\overrightarrow{e_i'} \in \mathbb{R}^{1\times D'}$. In the second step ($f^{(2)}_{\text{mask attention}}$), the multi-head attention layer is no longer sensitive to self-attention; therefore, the output is computed as an *average* instead of concatenating attention heads. The vector $\overrightarrow{e_i'}$ is then treated as the input embedding to be transformed into the final output embedding vector $\overrightarrow{e_i''}$, with $\overrightarrow{e_i''} \in \mathbb{R}^{1\times D''}$.

First, similar to the attention mechanism in 4.1, each embedding vector is multiplied by a weight matrix $\mathbf{W}_1 \in \mathbb{R}^{D_k \times D_{\text{in}}}$ to parameterize the linear transformation from D_{in} input dimensions to D_k higher-level feature dimensions:

$$\overrightarrow{h_i} = \mathbf{W}_1 \overrightarrow{e_i} \tag{4.5}$$

where $\overrightarrow{e_i} \in \mathbb{R}^{D_{\text{in}}\times 1} \rightarrow \overrightarrow{h_i} \in \mathbb{R}^{D_k \times 1}$

Next, we concatenate each pair of linearly transformed entity embedding vectors to compute the attention coefficients. The attention coefficient e_{ij} reflects the importance of the edge feature (e_i, e_j) with respect to the central entity e_i, or in other words, the importance of a neighboring entity e_j that is connected to e_i. We apply the LeakyReLU

function to extract the absolute value of the attention coefficient. Each attention coefficient e_{ij} is computed using the following equation:

$$e_{ij} = \left(\text{LeakyReLU}\left(\overrightarrow{\mathbf{W}_2}^T[\overrightarrow{h_i}||\overrightarrow{h_j}]\right)\right) \tag{4.6}$$

where $.^T$ denotes the transpose operation and $||$ represents concatenation. This is similar to 4.1, however instead of using a dot product, we use a *shared attentional mechanism* $\overrightarrow{\mathbf{W}_2}$: $\mathbb{R}^{D_k} \times \mathbb{R}^{D_k} \rightarrow \mathbb{R}$ to compute the attention scores. As mentioned in 4.3, we perform self-attention between all nodes using the masked attention mechanism to discard all irrelevant structural information.

To enable meaningful comparison between the attention coefficients of neighboring entities, a *softmax* function is applied to normalize the coefficients over all neighbors e_j that are connected to the central entity e_i: $\alpha_{ij} = \text{softmax}_j(e_{ij})$. Combining all of this, we obtain the final normalized attention coefficient of each neighbor with respect to the central entity as follows:

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\overrightarrow{\mathbf{W}_2}^T[\overrightarrow{h_i}||\overrightarrow{h_j}]\right)\right))}{\sum_{k\in\mathcal{N}_i}\exp\left(\text{LeakyReLU}\left(\overrightarrow{\mathbf{W}_2}^T[\overrightarrow{h_i}||\overrightarrow{h_k}]\right)\right))} \tag{4.7}$$

At this stage, the GAT model operates similarly to GCN [23], where the embedding vectors from neighboring nodes are aggregated and scaled by their corresponding normalized attention coefficients:

$$\overrightarrow{e_i'} = \sigma\left(\sum_{j\in\mathcal{N}_i}\alpha_{ij}\overrightarrow{h_j}\right) \tag{4.8}$$

Similar to the multi-head attention layer, we concatenate N_{head} attention heads to stabilize the training process in the first step ($f^{(1)}_{\text{mask attention}}$ 4.4) of the model:

$$\overrightarrow{e_i'} = \Big\|_{h=1}^{N_{\text{head}}}\sigma\left(\sum_{j\in\mathcal{N}_i}\alpha_{ij}^h\mathbf{W}^h\overrightarrow{h_j}\right) \tag{4.9}$$

where σ is any non-linear activation function, and α_{ij}^h is the normalized attention coefficient for edge (e_i, e_j) computed from the h^{th} attention head. Similar to equation 4.1, $\mathbf{W}^h$ is the weight matrix used for linear transformation of the input embedding vector, with each $\mathbf{W}^h$ corresponding to a different attention head. The resulting new embedding vector $\overrightarrow{e_i'} \in \mathbb{R}^{1\times D'}$, where $D' = N_{\text{head}}D_{\text{k}}$, is then used as the input for the

next attention layer. However, in the second step ($f^{(2)}_{\text{mask attention}}$ 4.4), the multi-head attention outputs are averaged instead of concatenated, as shown below:

$$\overrightarrow{e_i''} = \sigma\left(\frac{1}{N_{\text{head}}}\sum_{h=1}^{N_{\text{head}}}\sum_{j\in\mathcal{N}_i}\alpha_{ij}^{h}\mathbf{W}^{h}\overrightarrow{e_j'}\right) \tag{4.10}$$

Summary: Up to this point, we have presented how the attention mechanism aggregates a knowledge graph entity embedding vector from its neighboring embeddings and concatenates them to produce the final embedding. In the next section, we will present our complete embedding model based on the KBGAT model proposed by Nathani, Deepak[32].

4.4 KBGAT Model

In a knowledge graph, a single entity cannot fully represent an edge, as the entity can play multiple roles depending on the type of relation. For example, in Figure 4.7, Donald Trump serves both as a president and as a husband. To address this issue, the knowledge graph attention-based embedding model — KBGAT (graph attention based embeddings [32]) — improves upon the GAT model by incorporating additional information from *relations and neighboring node features* into the attention mechanism. In this section, we will detail the KBGAT model. The structure of KBGAT follows an encoder-decoder framework, where the encoder is implemented using the Graph Attention Network (GAT), and the decoder uses the ConvKB model for prediction. The steps of the KBGAT model are illustrated in 4.11.

$$\textit{entities} \xrightarrow{\text{Embedding}^{4.4.1}} \overrightarrow{e}_{\text{TransE}} \xrightarrow{\text{Embedding}^{4.4.2}} \overrightarrow{e}_{\text{KBGAT}} \xrightarrow{\text{ConvKB}^{4.4.3}} e_{\text{prob}} \tag{4.11}$$

First, the embedding vectors of each entity are initialized using the TransE model to capture the spatial characteristics among nodes and obtain the initial embeddings. These embeddings are then further trained using an encoder model to capture neighborhood features, resulting in updated embeddings. Finally, these embeddings are passed through a prediction layer using the ConvKB model. All equations presented here are based on those in the work of Nathani, Deepak[32].

4.4.1 Embedding Initialization

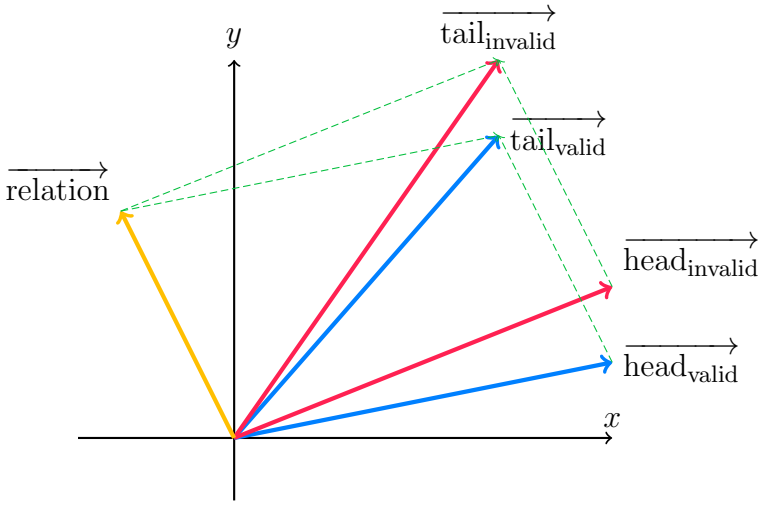


Figure 4.8: Illustration of embedding vectors in the TransE model

Similar to the Word2Vec method [30], the model *Translating Embeddings for Modeling Multi-relational Data* (TransE [5]) belongs to the group of geometric embedding methods that transform entities and relations in a knowledge graph into output embedding vectors such that:

$$\overrightarrow{\text{entity}_{\text{head}}} + \overrightarrow{relation} \approx \overrightarrow{\text{entity}_{\text{tail}}} \tag{4.12}$$

Initially, the entity and relation embedding vectors are randomly initialized using a normal distribution with dimensionality D_{in}, and then normalized according to the size of the entity and relation embedding sets.

Next, we perform sampling from the training dataset to obtain a batch of valid triples (S_{batch}). For each such triple, we sample an invalid triple by replacing either the head or the tail entity with a random entity from the entity set, yielding a batch of invalid triples (S'_{batch}). We then pair each valid triple with an invalid one to form the training batch (T_{batch}). Finally, we update the embedding vectors to satisfy the condition in 4.12.

Algorithm 5 TransE Embedding Learning Algorithm [5]

Input : Training set $S = (h, r, t)$, entity set E, relation set R, margin γ, embedding dimension D_{in}

Initialize

$\overrightarrow{r} \leftarrow \text{uniform}(-\frac{6}{\sqrt{D_{\text{in}}}}, \frac{6}{\sqrt{D_{\text{in}}}})$ for each relation $r \in R$

$\overrightarrow{r} \leftarrow \frac{\overrightarrow{r}}{\|\overrightarrow{r}\|}$ for each $r \in R$

$\overrightarrow{e} \leftarrow \text{uniform}(-\frac{6}{\sqrt{D_{\text{in}}}}, \frac{6}{\sqrt{D_{\text{in}}}})$ for each entity $e \in E$

loop

  $\overrightarrow{e} \leftarrow \frac{\overrightarrow{e}}{\|\overrightarrow{e}\|}$ for each $e \in E$

  $S_{\text{batch}} \leftarrow \text{sample}(S, b)$ // sample minibatch of size b

  $T_{\text{batch}} \leftarrow \varnothing$

  for $(h, r, t) \in S_{\text{batch}}$ **do**

    $(h', r, t') \leftarrow \text{sample}(S'_{(h,r,t)})$ // sample from invalid triple set

    $T_{\text{batch}} \leftarrow T_{\text{batch}} \cup \left\{ \left((h, r, t), (h', r, t') \right) \right\}$

Update embeddings

$$\sum_{\left((h,r,t),(h',r,t')\right) \in T_{\text{batch}}} \nabla[\gamma + d(\overrightarrow{h} + \overrightarrow{r}, \overrightarrow{t}) - d(\overrightarrow{h'} + \overrightarrow{r}, \overrightarrow{t'})]_+$$

Output : A set of embedding vectors with dimension D_{in} representing entities and relations

The TransE model proposed by Bordes, Antoine [5] is presented in Algorithm 5.

The input of the TransE model is a training dataset where each element is a triple (h, r, t). Here, $h, t \in E$ are the head and tail entities, and $r \in R$ is the relation. $\overrightarrow{e}$ and $\overrightarrow{r}$ are the embedding vectors of entities and relations respectively, and $\|\overrightarrow{e}\|$ and $\|\overrightarrow{r}\|$ denote the cardinalities of the entity and relation sets. S and S_{batch} represent the full training dataset and a sampled batch from it, respectively. T_{batch} is a batch containing both valid and invalid triples used to compute the loss function 4.13.

A *valid triple* is one directly sampled from the training set (S_{batch}), while an *invalid triple* is constructed by corrupting a valid triple (S'_{batch}) by replacing either the head or the tail entity with a randomly selected entity from the entity set:

$$S'_{(h,r,t)} = \left\{ (h', r, t) | h' \in E \right\} \cup \left\{ (h, r, t') | t' \in E \right\} \tag{4.13}$$

To achieve the goal of learning embedding vectors such that $\overrightarrow{h} + \overrightarrow{r} \approx \overrightarrow{t}$, the model

aims for the tail embedding $\overrightarrow{t}$ of valid triples to lie close to $\overrightarrow{h} + \overrightarrow{r}$, while in invalid triples, the corrupted embedding $\overrightarrow{h'} + \overrightarrow{r}$ (or $\overrightarrow{t'}$) should lie far from $\overrightarrow{t}$ (or $\overrightarrow{h} + \overrightarrow{r}$), according to the following margin-based ranking loss function:

$$\mathcal{L} = \sum_{(h,r,t) \in S} \sum_{(h',r,t') \in S'_{(h,r,t)}} [d - d' + \gamma]_+ \tag{4.14}$$

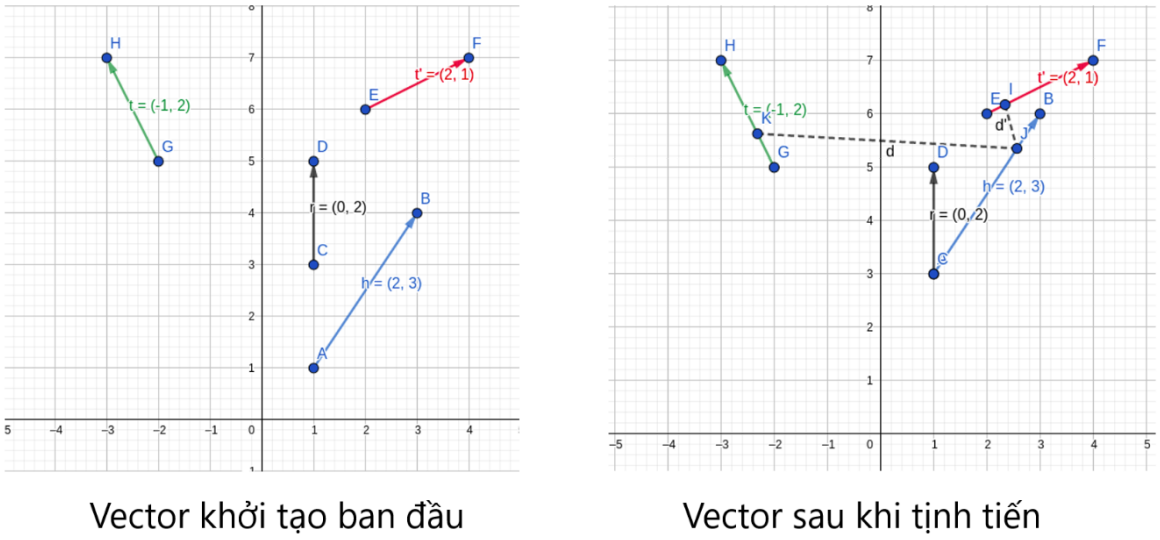


Figure 4.9: TransE embedding model

Here, $\gamma > 0$ is the margin, and h' and t' are entities sampled as defined in Equation 4.13. Δ and Δ' represent the difference vectors for the embeddings in valid and invalid triples, respectively, with $d = \left\|\Delta\right\|_1 = \left\|\overrightarrow{h} + \overrightarrow{r} - \overrightarrow{t}\right\|_1$ and $d' = \left\|\Delta'\right\|_1 = \left\|\overrightarrow{h'} + \overrightarrow{r} - \overrightarrow{t'}\right\|_1$, where $\|\cdot\|_1$ denotes the L1 norm.

As illustrated in Figure 4.9, if $d > d'$ or $d - d' > 0$, then $\overrightarrow{h} + \overrightarrow{r}$ is closer to $\overrightarrow{t}$ than to $\overrightarrow{t'}$. Since we want the embedding vectors to satisfy the condition in Equation 4.12, $\overrightarrow{h} + \overrightarrow{r}$ should be as close as possible to $\overrightarrow{t}$. That means, the closer $\overrightarrow{h} + \overrightarrow{r}$ is to $\overrightarrow{t'}$, the more incorrect it becomes.

Therefore, during training, we aim for Δ' to be as large as possible relative to Δ. If $\Delta' > \Delta$ or $d' - d > 0$, there is no need to update the embedding weights further. Hence, in the loss function 4.14, the term $[d - d' + \gamma]_+$ captures only the positive part because the negative part already satisfies the correctness of the condition in Equation 4.12 during training.

4.4.2 Encoder Model

After obtaining the embedding vectors that capture spatial features of the knowledge graph, these embeddings are passed through the next embedding layer to further aggregate the neighborhood information of each entity.

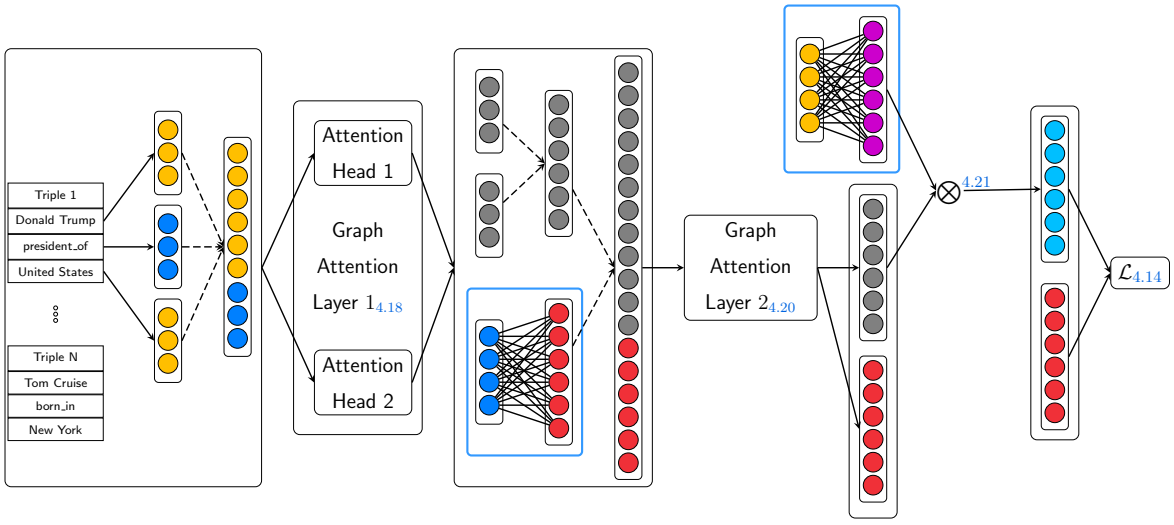


Figure 4.10: Illustration of the encoder layers in the KBGAT model

The model transforms the entity embedding matrix

$$\mathbf{E} = \left\{\overrightarrow{e_1}, \overrightarrow{e_2}, ..., \overrightarrow{e_{N_e}}\right\} \rightarrow \mathbf{E}'' = \left\{\overrightarrow{e''_1}, \overrightarrow{e''_2}, ..., \overrightarrow{e''_{N_e}}\right\},$$

with $\mathbf{E} \in \mathbb{R}^{N_e \times D_{\text{in}}}$ and $\mathbf{E}'' \in \mathbb{R}^{N_e \times D''}$.

Simultaneously, it transforms the relation embedding matrix

$$\mathbf{R} = \left\{\overrightarrow{r_1}, \overrightarrow{r_2}, ..., \overrightarrow{r_{N_r}}\right\} \rightarrow \mathbf{R}'' = \left\{\overrightarrow{r''_1}, \overrightarrow{r''_2}, ..., \overrightarrow{r''_{N_r}}\right\},$$

with $\mathbf{R} \in \mathbb{R}^{N_r \times P_{\text{in}}}$ and $\mathbf{R}'' \in \mathbb{R}^{N_r \times P''}$.

Similar to the GAT model described in Section 4.3, the model transforms the entity embedding vectors from D_{in} dimensions to D'' dimensions by aggregating neighborhood information through attention coefficients. P_{in} and P'' are the input and output dimensions of the relation embedding vectors, respectively. N_e and N_r are the sizes of the entity and relation sets in $\mathcal{G}_{know}$, respectively.

The KBGAT model concatenates the entity and relation embedding vectors according to the following structure:

$$\overrightarrow{t_{ijk}} = \mathbf{W_1}[\overrightarrow{e_i}||\overrightarrow{e_j}||\overrightarrow{r_k}] \tag{4.15}$$

Here, $\overrightarrow{t_{ijk}}$ is the embedding vector representing the triple $t_{ij}^k = (e_i, r_k, e_j)$, where e_j and r_k are the neighboring entity and the relation connecting the source node e_i to the node e_j. $\mathbf{W_1} \in \mathbb{R}^{D_k \times (2D_{\text{in}}+P_{\text{in}})}$ is a weight matrix that performs a linear transformation of the concatenated input vectors into a new vector with dimensionality D_k. These weight matrices are either randomly initialized using a normal distribution or pre-trained using the TransE model [5].

Similar to Equation 4.7 in the GAT model, we need to compute the attention coefficient for each edge with respect to a given node. Then, the *softmax* function is applied to normalize these coefficients as follows:

$$\begin{aligned}\alpha_{ijk} &= \text{softmax}_{jk}(\text{LeakyReLU}(\mathbf{W_2}\overrightarrow{t_{ijk}})) \\ &= \frac{\exp\left(\text{LeakyReLU}\left(\mathbf{W_2}\overrightarrow{t_{ijk}}\right)\right)}{\sum_{n\in\mathcal{N}_i}\sum_{r\in\mathcal{R}_{in}}\exp\left(\text{LeakyReLU}\left(\mathbf{W_2}\overrightarrow{t_{inr}}\right)\right)}\end{aligned} \tag{4.16}$$

where $\mathcal{N}_i$ denotes the set of neighbors of the central node e_i within n_{hop} hops; $\mathcal{R}_{in}$ represents the set of all relations that exist along the paths connecting the source entity e_i to a neighboring entity $e_n \in \mathcal{N}_i$. Similar to Equation 4.8, the embedding vectors $\overrightarrow{t_{ij}^k}$ are scaled by their corresponding normalized attention coefficients:

$$\overrightarrow{e_i'} = \sigma\left(\sum_{j\in\mathcal{N}_i}\sum_{k\in\mathcal{R}_{ij}}\alpha_{ijk}\overrightarrow{t_{ijk}}\right) \tag{4.17}$$

Similar to the multi-head attention mechanism in Equation 4.9, we concatenate N_{head} attention heads to stabilize the learning process:

$$\overrightarrow{e'}_i = \Big\Vert_{h=1}^{N_{\text{head}}}\sigma\left(\sum_{j\in\mathcal{N}_i}\alpha_{ijk}^{(h)}\overrightarrow{t_{ijk}^{(h)}}\right) \tag{4.18}$$

Likewise, for relation embeddings, a weight matrix $\mathbf{W}_R$ is used to perform a linear transformation from the original relation embedding of dimension P to a new dimension P':

$$\mathbf{R}' = \mathbf{R}\mathbf{W}^R; \qquad \text{where: } \mathbf{W}^R \in \mathbb{R}^{P \times P'} \tag{4.19}$$

At this stage, we have obtained two matrices: $\mathbf{H}' \in \mathbb{R}^{N_e \times D'}$ and $\mathbf{R}' \in \mathbb{R}^{N_r \times P'}$, which are the updated entity and relation embedding matrices, respectively, with new dimensions. The model proceeds through the final attention layer, taking as input the newly updated entity and relation embeddings as shown in Equation 4.10. However, if we apply multi-head attention at this final layer for prediction, the concatenation operation will no longer be *sensitive* to the self-attention mechanism. Therefore, instead of concatenating, the model averages the outputs and then applies a final non-linear activation function:

$$\overrightarrow{e_i''} = \sigma\left(\frac{1}{N_{\text{head}}}\sum_{h=1}^{N_{\text{head}}}\sum_{j\in\mathcal{N}_i}\sum_{k\in\mathcal{R}_{ij}}\alpha_{ijk}^{\prime(h)}\overrightarrow{t_{ijk}^{\prime(h)}}\right) \tag{4.20}$$

where $\alpha_{ijk}^{\prime(h)}$ and $t_{ijk}^{\prime(h)}$ denote the normalized attention coefficients and the triple embedding vectors for (e_i, r_k, e_j) in attention head (h), respectively.

Up to this point, the KBGAT model functions similarly to the GAT model in Section 4.3, but it additionally incorporates both entity embedding information and neighbor nodes up to n_{hop} hops. This results in the final entity embedding matrix $\mathbf{E}'' \in \mathbb{R}^{N_e \times D''}$ and the final relation embedding matrix $\mathbf{R}'' \in \mathbb{R}^{N_r \times P''}$.

However, after the embedding learning process, the final entity embedding matrix $\mathbf{E}''$ may lose the initial embedding information due to the **vanishing gradient** problem. To address this, the model employs residual learning, by projecting the initial embedding matrix $\mathbf{E}$ through a weight matrix $\mathbf{W}^E \in \mathbb{R}^{D_{\text{in}} \times D''}$, and then directly adding it to the final embedding, thus preserving the initial embedding information during training:

$$\mathbf{H} = \mathbf{W}^E\mathbf{E} + \mathbf{E}'' \tag{4.21}$$

Finally, the training datasets are sampled to generate valid triples and invalid triples, similar to the TransE model described above, in order to learn the embedding vectors. However, the distance between embedding vectors is computed using the L1 norm as follows:

$$d_{t_{ij}} = \left|\left|\vec{h_i} + \vec{g_k} - \vec{h_j}\right|\right|_1$$

Similarly, we train the model using a margin-based loss function:

$$L(\Omega) = \sum_{t_{ij} \in S} \sum_{t'_{ij} \in S'} \max\{d_{t'_{ij}} - d_{t_{ij}} + \gamma, 0\} \tag{4.22}$$

where $\gamma > 0$ is the margin parameter, S is the set of valid triples, and S' is the set of invalid triples, defined as:

$$S' = \underbrace{\{t^k_{i'j} | e'_i \in \mathcal{E} \setminus e_i\}}_{\text{head entity replacement}} \cup \underbrace{\{t^k_{ij'} | e'_j \in \mathcal{E} \setminus e_j\}}_{\text{tail entity replacement}} \tag{4.23}$$

The output of the KBGAT model is the set of entity and relation embedding vectors, which are subsequently fed into the ConvKB model for final prediction.

4.4.3 ConvKB Prediction Model

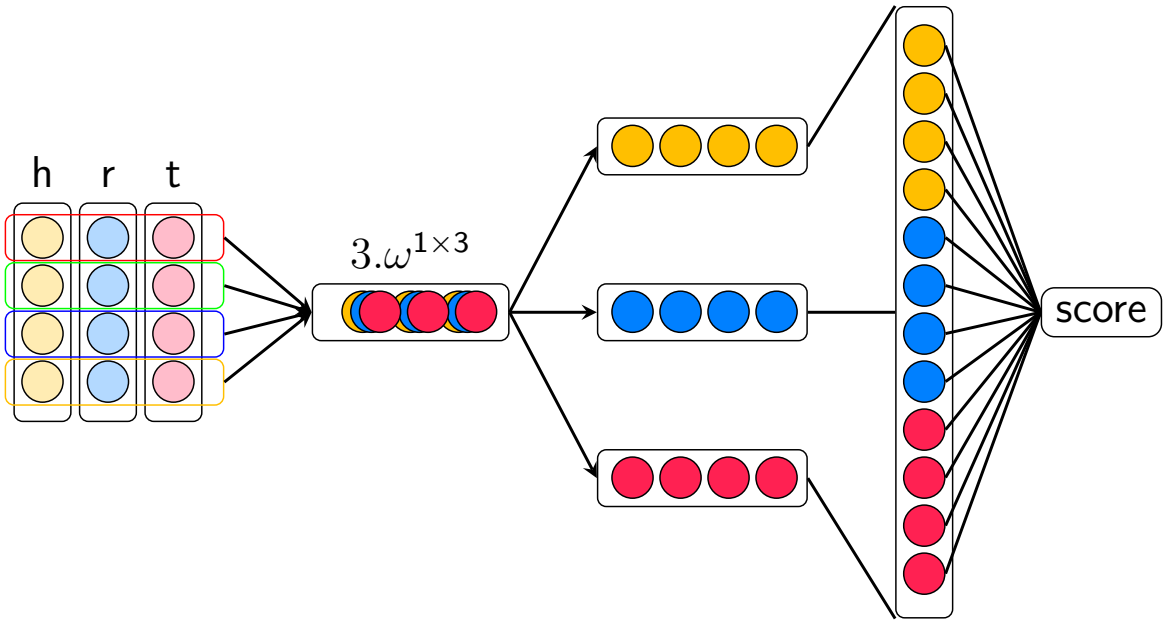


Figure 4.11: Illustration of the decoder layers of the ConvKB model with 3 filters

After mapping entities and relations into a low-dimensional space, the model employs ConvKB [33] to analyze the global features of a triple t_{ijk} across each dimension, thereby generalizing transformation patterns through convolutional layers. The scoring function with the learned feature mappings is defined as follows:

$$f(t_{ijk}) = \Big(\Big\|_{m=1}^{\Omega} \text{ReLU}([\overrightarrow{e_i}, \overrightarrow{r_k}, \overrightarrow{h_j}] * \omega^m) \Big) . \mathbf{W} \tag{4.24}$$

where ω^m denotes the m-th convolutional filter, Ω is the hyperparameter representing the number of convolutional layers, $*$ denotes the convolution operation, and $\mathbf{W} \in \mathbb{R}^{\Omega k \times 1}$ is the linear transformation matrix used to compute the final score for the triple.

The model is trained using a soft-margin loss function as follows:

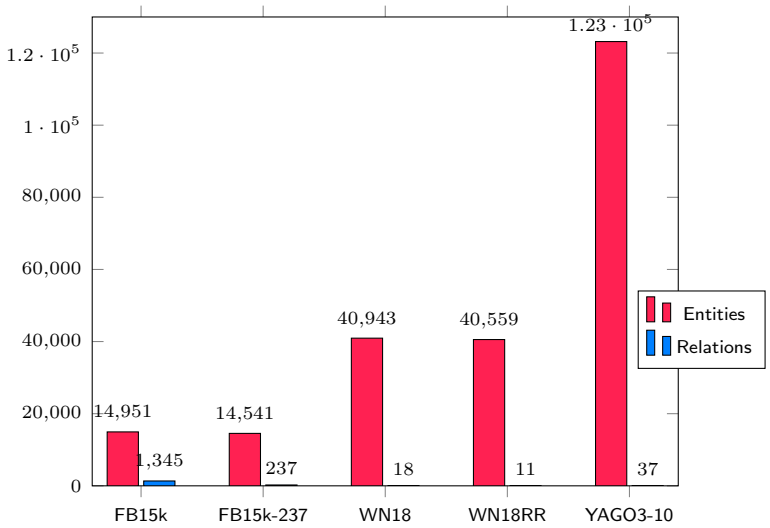
$$\mathcal{L} = \sum_{t_{ij}^k \in \{S \cup S'\}} \log(1 + \exp(l_{t_{ij}^k} \cdot f(t_{ij}^k))) + \frac{\lambda}{2} \parallel \mathbf{W} \parallel_2^2 \tag{4.25}$$

where

$$l_{t_{ij}^k} = \begin{cases} 1 & \text{for } t_{ij}^k \in S \\ -1 & \text{for } t_{ij}^k \in S' \end{cases}$$

The final output of the ConvKB model is the ranking score corresponding to each prediction.

Chapter 5. EXPERIMENTS

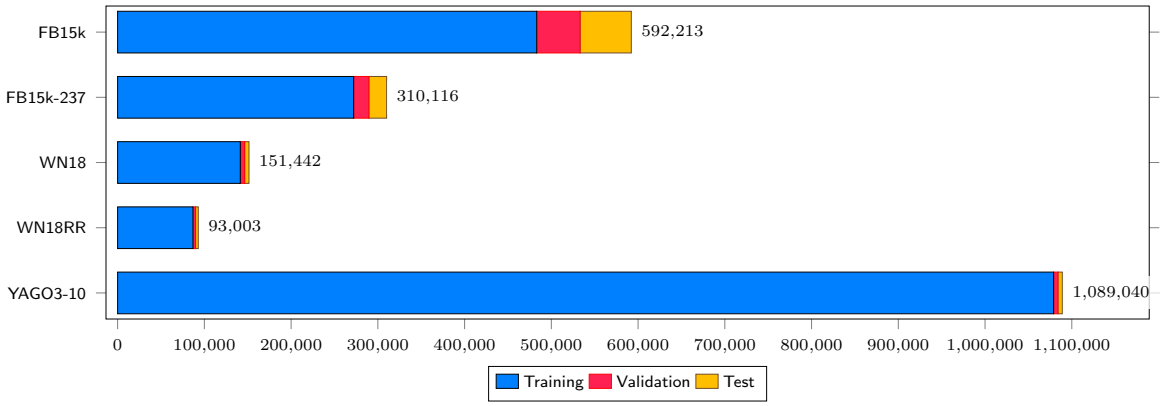


In this section, we describe the datasets used for our empirical evaluation, along with a comparison against notable existing methods as reported in Table 4.1. Additionally, we evaluate our two proposed approaches for injecting new knowledge into the knowledge graph. Specifically, we treat the test set as a batch of new knowledge to be added, and use the validation set to re-evaluate the effectiveness of our method. Detailed results are presented in Table 5.2 and Table 5.3.

			# Edges		
Dataset	Entities	Relations	Training	Validation	Test
FB15k	14,951	1,345	483,142	50,000	59,071
FB15k-237	14,541	237	272,115	17,535	20,466
WN18	40,943	18	141,442	5,000	5,000
WN18RR	40,559	11	86,835	3,034	3,134
YAGO3-10	123,182	37	1,079,040	5,000	5,000

Table 5.1: Dataset Information

5.1 Training Datasets



In our experiments, we evaluate our approach on four widely used benchmark datasets: FB15k, FB15k-237 ([43]), WN18, and WN18RR ([12]). Each dataset is divided into three subsets: training, validation, and test sets. Detailed statistics for these datasets are presented in Table 5.1.

Each dataset consists of a collection of triples in the form $\langle head, relation, tail \rangle$. FB15k and WN18 are derived from the larger knowledge bases FreeBase and WordNet, respectively. However, they contain a large number of inverse relations, which allow most triples to be easily inferred. To address this issue and to better reflect real-world link prediction scenarios, FB15k-237 and WN18RR were constructed by removing such inverse relations.

5.1.1 FB15k Dataset

This dataset was constructed by the research group of A. Bordes and N. Usunier [5] by extracting data from the Wikilinks database[1]. The Wikilinks database contains hyperlinks to Wikipedia articles, comprising over 40 million mentions of approximately 3 million entities. The authors extracted all facts related to a given entity that is mentioned at least 100 times across different documents, along with all facts associated with those entities (including child entities mentioned in the corresponding Wikipedia articles), excluding attributes such as dates, proper nouns, etc.

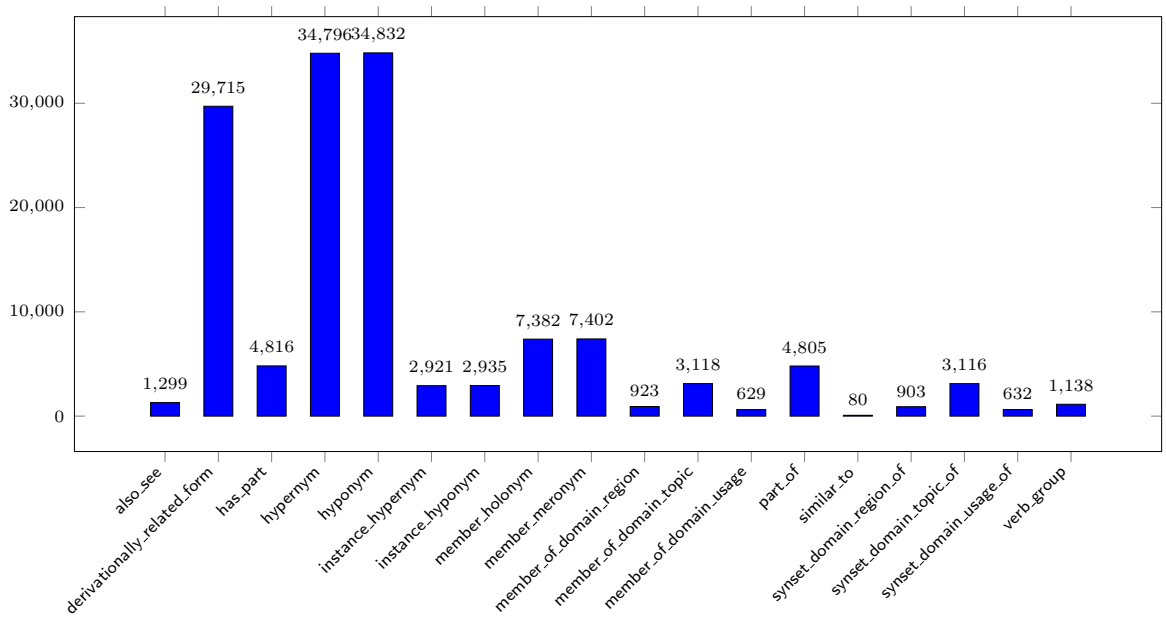
Entities with degree n were normalized by converting multi-way edges into sets of binary relations—enumerating all edges and relations between each pair of nodes.

[1] https://code.google.com/archive/p/wiki-links/

5.1.2 FB15k-237 Dataset

This dataset is a subset of FB15k, constructed by Toutanova and Chen [43], motivated by the observation that FB15k suffers from test leakage, where models are exposed to test facts during training. This issue arises due to the presence of duplicate or inverse relations in FB15k. FB15k-237 was created to provide a more challenging benchmark. The authors selected facts related to the 401 most frequent relations and eliminated all redundant or inverse relations. Additionally, they ensured that no entities connected in the training set were directly linked in the test and validation sets.

5.1.3 WN18 Dataset



This dataset was introduced by the authors of TransE [5], and is extracted from WordNet[2], a lexical knowledge graph ontology designed to provide a dictionary/thesaurus to support NLP tasks and automated text analysis. In WordNet, entities correspond to synsets (i.e., *word senses*), and relations represent lexical connections among them (e.g., "hypernym").

To construct WN18, the authors used WordNet as a starting point and iteratively filtered out entities and relations that were infrequently mentioned.

[2]https://wordnet.princeton.edu/

5.1.4 WN18RR Dataset

This dataset is a subset of WN18, constructed by DeŠmers et al.[11], who also addressed the issue of test leakage in WN18. To tackle this issue, they created the WN18RR dataset, which is significantly more challenging, by applying a similar methodology to that used in FB15k-237 [43].

5.2 Evaluation Metrics

In this section, we describe the evaluation metrics, experimental environment, and datasets used to assess the proposed method. These metrics are widely adopted for evaluating link prediction models on knowledge graphs. We compare our approach against four other state-of-the-art methods reported in [38].

5.2.0.1 Hits@K (H@K)

This metric measures the proportion of correct predictions whose rank is less than or equal to the threshold K:

$$H@K = \frac{\mid \{q \in Q : \text{rank}(q) \leq K\} \mid}{\mid Q \mid}$$

5.2.0.2 Mean Rank (MR)

This metric calculates the average rank of the correct entity in the prediction. A lower value indicates better model performance:

$$MR = \frac{1}{\mid Q \mid} \sum_{q \in Q} \text{rank}(q)$$

Here, $\mid Q \mid$ denotes the total number of queries, which equals the size of the test or validation set. During evaluation, we perform both head and tail entity predictions for each triple. For example, we predict both $\langle ?, \text{relation}, \text{tail} \rangle$ and $\langle \text{head}, \text{relation}, ? \rangle$. The variable q denotes a query, and $\text{rank}(q)$ indicates the rank position of the correct entity. The final MR score is the average rank over all head and tail predictions.

Clearly, this metric ranges from [1, |number of entities|], as a node can connect to at most $n - 1$ other nodes plus a self-loop. However, this metric is highly sensitive to outliers, as certain relations may yield extremely low rankings for correct entities. To

address this issue, our method—as well as other recent works—also adopts the Mean Reciprocal Rank (MRR) metric.

5.2.0.3 Mean Reciprocal Rank (MMR)

This is the Mean Reciprocal Rank (MRR), calculated as the reciprocal of the average rank obtained for a correct prediction. Higher values indicate better model performance. Since this metric takes the reciprocal of each rank, it helps mitigate the noise sensitivity encountered in the Mean Rank (MR) metric:

$$MRR = \frac{1}{\mid Q \mid} \sum_{q \in Q} \frac{1}{\text{rank}(q)}$$

5.3 Training Methodology

5.3.1 Training with the KBGAT Model

We first initialize the entity and relation embeddings using the TransE model [5]. To construct negative (invalid) triples, we randomly replace either the head or tail entity in a valid triple with another entity sampled from the entity set.

The training process is divided into two phases. The first phase serves as an encoder, transforming the initialized embeddings into new embeddings that aggregate neighborhood information using the KBGAT model. This produces updated embeddings for both entities and relations. The second phase serves as a decoder, performing link prediction by incorporating n-hop neighborhood information. This enables the model to better aggregate context from neighboring entities. Furthermore, we incorporate auxiliary relations to enrich the local structure in sparse graphs.

We use the Adam optimizer with a learning rate of $\mu = 0.001$. The final embedding dimension for both entities and relations is set to 200. The optimal hyperparameters are determined via grid search, as detailed in Appendix A.

5.4 Experimental Results

As previously mentioned, our rule-based model can be fully executed on a standard laptop. In our experiment, the machine configuration was as follows: T480, Core i5 8th Gen, 16GB RAM, 4 cores and 8 threads. The source code was implemented in

Python version 3.6, utilizing only built-in Python functions without any third-party libraries. The experiments were conducted on four widely-used datasets: FB15k, FB15-237, WN18, and WN18RR. Detailed information about these datasets is provided in Section 5.1 under the training datasets section.

As described in Algorithm 2, the AnyBURL algorithm learns rules generated within a user-configurable time interval. Here, we set the training time to 1000 seconds (approximately 17 minutes), with saturation (SAT) set to 0.85, confidence threshold Q set to 0.05, and sample size S configured as ($\frac{1}{10}$ of the training set). With this setup, our Python version of the model produced results comparable to the Java version developed by Meilicke, Christian et al. [27], which was configured similarly but trained for only 100 seconds. The difference in training time is primarily due to performance differences between Python and Java. We chose Python because it is the primary language used in many recent artificial intelligence models and provides convenience for performance comparison and evaluation with other deep learning methods, most of which are also implemented in Python.

	FB15k				**FB15k-237**			
	H@1	**H@10**	**MR**	**MRR**	**H@1**	**H@10**	**MR**	**MRR**
ComplEx	81.56	90.53	34	0.848	25.72	52.97	202	0.349
TuckER	72.89	88.88	39	0.788	25.90	53.61	162	0.352
TransE	49.36	84.73	45	0.628	21.72	49.65	209	0.31
RoteE	73.93	88.10	42	0.791	23.83	53.06	178	0.336
ConvKB	59.46	84.94	51	0.688	21.90	47.62	281	0.305
KBGAT	70.08	91.64	38	0.784	36.06	58.32	211	0.4353
AnyBURL	79.13	82.30	285	<u>0.824</u>	20.85	42.40	490	0.311

Table 5.2: Experimental results on the FB15k and FB15k-237 datasets

	WN18				WN18RR			
	H@1	**H@10**	**MR**	**MRR**	**H@1**	**H@10**	**MR**	**MRR**
ComplEx	94.53	95.50	3623	0.349	42.55	52.12	4909	0.458
TuckER	94.64	95.80	510	0.951	42.95	51.40	6239	0.459
TransE	40.56	94.87	279	0.646	2.79	94.87	279	0.646
RoteE	94.30	96.02	274	0.949	42.60	57.35	3318	0.475
ConvKB	93.89	95.68	413	0.945	38.99	50.75	4944	0.427
KBGAT					35.12	57.01	1974	0.4301
AnyBURL	93.96	95.07	**230**	**0.955**	**44.22**	54.40	2533	0.497

Table 5.3: Experimental results on the WN18 and WN18RR datasets

Table 5.2 and Table 5.3 describe our experimental results with the $H@K$ metrics along with the experimental results of other methods mentioned in the survey [38].

		AnyBURL	**Batch edge AnyBURL**	**Edge AnyBURL**
FB-15k	hit@10	82.22	82.48	83.08
	MR	285	250	220
	MRR	0.824	0.853	0.866
FB15k-237	hit@10	42.40	43.40	43.51
	MR	490	472	441
	MRR	0.311	0.353	0.377
WN18	hit@10	95.07	95.09	95.19
	MR	230	229	228
	MRR	0.955	0.955	0.956
WN18RR	hit@10	54.40	54.63	54.70
	MR	2533	2346	2215
	MRR	0.497	0.553	0.581

Table 5.4: Accuracy results of the two new knowledge addition strategies

Table 5.4 describes our experimental results for the two strategies of adding new knowledge into the graph. We evaluate the total number of generated rules, as well as the number of rules with confidence $>= 50\%$ and $>= 80\%$.

		Batch edge AnyBURL	**Edge AnyBURL**
FB15k	num rule	1011	1367
	confidence 50%	416 (41,14%)	1185 (86,69%)
	confidence 80%	284 (28, 09%)	481 (35,18%)
FB15k-237	num rule	1120	756
	confidence 50%	244 (21,79%)	660 (87,30%)
	confidence 80%	95 (8,48%)	162 (21,43%)
WN18	num rule	533	260
	confidence 50%	270 (38, 46 %)	252 (96,92%)
	confidence 80%	240 (34,19%)	225 (86,54%)
WN18RR	num rule	439	106
	confidence 50%	110 (25,05%)	102 (96,22%)
	confidence 80%	83 (18,91%)	85 (81,19%)

Table 5.5: Evaluation results on the number of rules of the two new knowledge addition strategies

Chapter 6. CONCLUSION

In this section, we presented the results achieved by the proposed model, along with detailed analyses on different datasets to clarify both the strengths and the remaining limitations. From this, we identified potential research directions for improving the model in the future.

Although our rule-based method demonstrates performance comparable to modern deep learning models (state-of-the-art), and clearly outperforms them in terms of training time—only about 17 minutes compared to several hours for deep learning models—this does not imply that deep learning models are not worth studying. On the contrary, through performance analysis across different datasets, we observed that for datasets with diverse relations like FreeBase, the KBGAT model using attention mechanisms yielded significantly better results than on datasets like WordNet, which contain fewer relation types. This highlights the potential of leveraging deep learning mechanisms tailored to the specific characteristics of each dataset.

This shows that the attention mechanism, by incorporating relational embedding information, helps to better capture graph structures in datasets with a wide variety of relations. For datasets with many similar and inverse triples such as FB15k and WN18RR, the rule-based model AnyBURL achieved superior results, whereas deep learning methods only achieved average performance compared to other methods. The rule-based AnyBURL model performs better on datasets like FB15k and WN18RR; however, for datasets that have removed similar or inverse information, such as FB15k-237 and WN18RR, the rule-based method is less effective, since it relies on previously observed paths or links. In contrast, deep learning models represent relations and entities in a vector space to learn their interactions, allowing them to perform better on datasets like FB15k-237 and WN18RR than on FB15k and WN18.

One of the main advantages of the rule-based approach is that the generated rules are interpretable during training and require significantly less training time compared to other methods. However, after the training phase, the rule-based method must

iterate through all learned rules to make predictions. This is an area where deep learning models show better performance, as models like KBGAT can use the learned weights and computational layers to transform inputs into probabilistic predictions much faster. The drawback of deep learning approaches is their lack of interpretability during training, as well as the high computational cost. Regarding our two proposed algorithms for adding new knowledge to the graph, we found them to significantly outperform deep learning methods.

The graph embedding process helps represent the features of entities, relations, or the characteristics of the knowledge graph as lower-dimensional vectors (Section 4.1). However, in practice, a piece of knowledge represented by entities and relations is entirely independent between these components; thus, they should be embedded into vectors with different dimensionalities. The ratio of dimensions between entities and relations is also an important issue that requires further investigation.

Additionally, in the real world, the temporal factor is a critical piece of information that can completely alter the meaning of a piece of knowledge. Therefore, integrating temporal information into the attention mechanism is one of the research directions we aim to pursue to ensure the semantic accuracy of the knowledge graph.

For the rule-based method AnyBURL, the reinforcement learning branch has recently seen significant progress, and the authors Meilicke, Christian and Chekol [26] have recently proposed a study to optimize the AnyBURL method using reinforcement learning. We also intend to explore this direction and aim to report our findings in the near future.

REFERENCES

[1] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1999.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[3] Shumeet Baluja, Rohan Seth, Dharshi Sivakumar, Yushi Jing, Jay Yagnik, Shankar Kumar, Deepak Ravichandran, and Mohamed Aly. Video suggestion and discovery for youtube: taking random walks through the view graph. In *Proceedings of the 17th international conference on World Wide Web*, pages 895–904, 2008.

[4] Aleksandar Bojchevski, Oleksandr Shchur, Daniel Zügner, and Stephan Günnemann. Netgan: Generating graphs via random walks. *arXiv preprint arXiv:1803.00816*, 2018.

[5] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Advances in neural information processing systems*, pages 2787–2795, 2013.

[6] Samuel R Buss. An introduction to proof theory. *Handbook of proof theory*, 137:1–78, 1998.

[7] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 30(9):1616–1637, 2018.

[8] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM interna-*

tional on conference on information and knowledge management, pages 891–900, 2015.

[9] Jean-Baptiste Cordonnier, Andreas Loukas, and Martin Jaggi. On the relationship between self-attention and convolutional layers. *arXiv preprint arXiv:1911.03584*, 2019.

[10] Jean-Baptiste Cordonnier, Andreas Loukas, and Martin Jaggi. Multi-head attention: Collaborate instead of concatenate. *arXiv preprint arXiv:2006.16362*, 2020.

[11] Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. Convolutional 2d knowledge graph embeddings. *arXiv preprint arXiv:1707.01476*, 2017.

[12] Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. Convolutional 2d knowledge graph embeddings. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[13] Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M. Suchanek. Amie: association rule mining under incomplete evidence in ontological knowledge bases. In *WWW '13*, 2013.

[14] Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M Suchanek. Fast rule mining in ontological knowledge bases with amie. *The VLDB Journal*, 24(6):707–730, 2015.

[15] Google. *Introducing the Knowledge Graph: things, not strings*, 2020 (accessed on August 27, 2020).

[16] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

[17] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[18] Lingbing Guo, Zequn Sun, and Wei Hu. Learning to exploit long-term relational dependencies in knowledge graphs. *arXiv preprint arXiv:1905.04914*, 2019.

[19] Wilfrid Hodges et al. *A shorter model theory*. Cambridge university press, 1997.

[20] John J Hopfield. Hopfield network. *Scholarpedia*, 2(5):1977, 2007.

[21] Sergey Ivanov and Evgeny Burnaev. Anonymous walk embeddings. *arXiv preprint arXiv:1805.11921*, 2018.

[22] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and Philip S Yu. A survey on knowledge graphs: Representation, acquisition and applications. *arXiv preprint arXiv:2002.00388*, 2020.

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

[24] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object recognition with gradient-based learning. In *Shape, contour and grouping in computer vision*, pages 319–345. Springer, 1999.

[25] Johann A. Makowsky. Why horn formulas matter in computer science: Initial structures and generic examples. *Journal of Computer and System Sciences*, 34(2-3):266–292, 1987.

[26] Christian Meilicke, Melisachew Wudage Chekol, Manuel Fink, and Heiner Stuckenschmidt. Reinforced anytime bottom up rule learning for knowledge graph completion. *arXiv preprint arXiv:2004.04412*, 2020.

[27] Christian Meilicke, Melisachew Wudage Chekol, Daniel Ruffinelli, and Heiner Stuckenschmidt. Anytime Bottom-Up Rule Learning for Knowledge Graph Completion, 2019.

[28] Christian Meilicke, Melisachew Wudage Chekol, Daniel Ruffinelli, and Heiner Stuckenschmidt. Anytime bottom-up rule learning for knowledge graph completion. In *IJCAI*, pages 3137–3143, 2019.

[29] Christian Meilicke, Manuel Fink, Yanjie Wang, Daniel Ruffinelli, Rainer Gemulla, and Heiner Stuckenschmidt. Fine-grained evaluation of rule-and embedding-based systems for knowledge graph completion. In *International Semantic Web Conference*, pages 3–20. Springer, 2018.

[30] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[31] Seyedeh Fatemeh Mousavi, Mehran Safayani, Abdolreza Mirzaei, and Hoda Bahonar. Hierarchical graph embedding in vector space by graph pyramid. *Pattern Recognition*, 61:245–254, 2017.

[32] Deepak Nathani, Jatin Chauhan, Charu Sharma, and Manohar Kaul. Learning attention-based embeddings for relation prediction in knowledge graphs. *arXiv preprint arXiv:1906.01195*, 2019.

[33] Dai Quoc Nguyen, Tu Dinh Nguyen, Dat Quoc Nguyen, and Dinh Phung. A novel embedding model for knowledge base completion based on convolutional neural network. *arXiv preprint arXiv:1712.02121*, 2017.

[34] Ministry of Health of Vietnam. *Memory and Attention*, 2020. (Accessed on August 26, 2020).

[35] Stefano Ortona, Venkata Vamsikrishna Meduri, and Paolo Papotti. Robust discovery of positive and negative rules in knowledge bases. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1168–1179. IEEE, 2018.

[36] 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*, pages 701–710, 2014.

[37] Prajit Ramachandran, Niki Parmar, Ashish Vaswani, Irwan Bello, Anselm Levskaya, and Jonathon Shlens. Stand-alone self-attention in vision models. *arXiv preprint arXiv:1906.05909*, 2019.

[38] Andrea Rossi, Donatella Firmani, Antonio Matinata, Paolo Merialdo, and Denilson Barbosa. Knowledge graph embedding for link prediction: A comparative analysis. *arXiv preprint arXiv:2002.00819*, 2020.

[39] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic routing between capsules. In *Advances in neural information processing systems*, pages 3856–3866, 2017.

[40] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter

language models using gpu model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[41] 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*, pages 1067–1077, 2015.

[42] Andrew Tao, Karan Sapra, and Bryan Catanzaro. Hierarchical multi-scale attention for semantic segmentation. *arXiv preprint arXiv:2005.10821*, 2020.

[43] Kristina Toutanova and Danqi Chen. Observed versus latent features for knowledge base and text inference. In *Proceedings of the 3rd Workshop on Continuous Vector Space Models and their Compositionality*, pages 57–66, 2015.

[44] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. Complex embeddings for simple link prediction. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, 2016.

[45] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[47] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[48] Thanh Vu, Tu Dinh Nguyen, Dat Quoc Nguyen, Dinh Phung, et al. A capsule network-based embedding model for knowledge graph completion and search personalization. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2180–2189, 2019.

[49] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1225–1234, 2016.

[50] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*, pages 5753–5763, 2019.

Appendix A. OPTIMAL HYPERPARAMETERS

In this section, we present the set of optimal hyperparameters used for both models: the attention-based model and the ConvKB model. The hyperparameter optimization process was conducted using grid search based on the Hits@10 evaluation metric. For the attention-based model, we trained on the entire dataset without any data splitting. In contrast, for the ConvKB model, we applied the same hyperparameter configuration across all datasets. The detailed hyperparameters are shown in the following table:

	μ	Weight decay	Epochs	negative ratio	Dropouts	$\alpha_{\text{LeakyRELU}}$	N_{head}	D_{final}	γ
FB15k	1e-3	1e-5	3000	2	0.3	0.2	2	200	1
FB15k-237	1e-3	1e-5	3000	2	0.3	0.2	2	200	1
WN18	1e-3	5e-6	3600	2	0.3	0.2	2	200	5
WN18RR	1e-3	5e-6	3600	2	0.3	0.2	2	200	5