Master's Thesis at IDLab, Ghent University - imec

A Trip to Sesame Street: Evaluation of BERT and Other Recent Embedding Techniques Within RDF2Vec

Author:
Terencio Agozzino

Advisor:
Prof. Dr. Femke Ongenae

Supervisors:
Dr. Ir. Gilles Vandewiele
Dr. Ir. Samuel Cremer
Ir. Bram Steenwinckel

Master's thesis submitted in fulfillment of the requirements for the degree of Master in Industrial Engineering

Academic year 2020-2021

Abstract

Over the past decade, various use cases have highlighted the benefits of converting a Knowledge Graph into a 2D feature matrix, called embedding matrix. This conversion can be done with RDF2Vec, an unsupervised task-agnostic algorithm for numerically representing Knowledge Graph nodes to be used for downstream Machine Learning tasks. Since 2016, this algorithm has provided good results using Word2Vec, an embedding technique initially used in the Natural Language Processing field. However, other techniques in this field have emerged, such as BERT, which since 2018, is the state-of-the-art algorithm. The goal of this Master's thesis mainly focused on evaluating BERT for Knowledge Graphs to determine its impact compared to Word2Vec and FastText. As a result, this Master's thesis proposed an implementation of BERT and FastText related to Knowledge Graphs and improving the node embedding's quality generated by Word2Vec. For the latter, it was suggested to both extract the root nodes' parents and centralize the position of these roots within their walk extraction. This Master's thesis also extended the use of RDF2Vec by introducing `SplitWalker` and `WideSampler` as new walking and sampling strategies. The study done reveals that the results obtained by BERT with RDF2Vec are not conclusive, contrary to the expectations. The main reason is the lack of optimization of the BERT's architecture towards Knowledge Graphs, which explains the creation of BERT variants in this direction. Finally, the model's accuracy generated by Word2Vec has increased considerably, and both `SplitWalker` and `WideSampler` have proven their effectiveness in certain use cases.

Keywords: BERT, Knowledge Graph, Machine Learning, RDF2Vec, Word2Vec

Acknowledgements

First of all, I would like to express my sincere appreciation to my supervisors, Dr. Ir. G. Vandewiele, Ing. B. Steenwinckel, and Dr. Ir. S. Cremer. My heartfelt thanks to Dr. Ir. G. Vandewiele, who, with his experience and knowledge, guided me throughout this Master's thesis. Dr. Ir. G. Vandewiele was the supervisor that any student would want to have. He was always there to help and give good advice. I would also like to thank Ir. B. Steenwickel, who also played an essential role in this work. Ir. B. Steenwickel also helped me get the job done. Finally, I would like to thank Dr. Ir. S. Cremer for his advice in the elaboration of this Master's thesis and for having accepted to supervise me.

I would also like to express my gratitude to Prof. Dr. F. Ongenae, who gave her approval to realize this Master's thesis and directly accepted me with welcome arms.

I wish to extend my special thanks to Dr. P. Colpaert for allowing me to get in touch with imec and the IDLab research center. Without his help, this experience would not have been possible.

I want to express my gratitude to the Haute École in Hainaut to achieve this Master's thesis and to thank every teacher for having passed on their knowledge to me. Special attention to Ing. L. Isidoro, MSc. Ir. J-S. Lerat, BSc. Y. Pietrzak, MSc. L. Remacle, and Ing. G. Tricarico for their kindness, their passion, and for having made me want to continue these studies.

I would also like to thank my friends, whom I have met during these years of study and who, through their words, their motivation, and the many memories I have shared with them, have made my life better. In particular, I would like to thank C. Bruyère, I. Delsarte, V. Denis, S. Eker, N. Luongo, G. Quittet, and T. Simon. A particular thought to my childhood friend, H. Koch, who followed my evolution since the beginning of my teen years and has always been of great help.

Finally, I would like to dedicate this Master's thesis to my family and, more particularly, to my great-grandfather and my grandmother. Throughout these years of study, they have always supported me. My life would have taken a different path without you. I hope that I can one day be as good a person as you are.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In an increasingly digital world, data generation applies different semantic and syntactic origins (Ceravolo et al., 2018). However, this data diversity must remain interpretable by the computer. In the absence of semantics, the evaluation of the validity of the data is more constraining. Without it, the same data point might not represent the same thing. For example, the value of a sensor may be correct in one context but an anomaly in another. Therefore, semantics makes it possible to make the context of these data precise and understand their relationships.

The usage of the Resource Description Framework (RDF) standard of the World Wide Web Consortium (W3C) enables the semantic encoding of data. Such a standard allows the management of this diversity of data through the Semantic Web and Linked Open Data by interconnecting the different data sources. Disregarding the *knowledge base*, which contains semantic information, one way to make this interconnection is to use graphs. A *graph* is an ordered pair (V, E), where V is a finite and non-empty set of elements called *vertices* (or *nodes*), and E is a set of unordered pairs of distinct nodes of V, called *edges*.

Based on this alternative representation of RDF data, the concept of Knowledge Graph (KG) was published (Singhal, 2012). Mathematically, a KG is a *directed heterogeneous multigraph*. This graph can store multiple directed labeled edges between two nodes whose edges and nodes can be of different types. Additionally, a KG can unite various sources and enhance conventional data formats such as Comma Separated Values (CSV) by explicitly encoding the relations between various nodes. Due to the richness of relations that these types of graphs bring, several Machine Learning (ML) techniques can benefit from them. From then on, there is a restricted usage of KGs due to their symbolic constructs as most ML techniques require converting these KGs into numerical input feature vectors.

During this decade, different techniques emerged (Ristoski and Paulheim, 2014) to create these numerical feature vectors, called *embeddings*. One of them is to use Resource Description Framework To Vector (RDF2Vec), an unsupervised and task-agnostic algorithm to numerically represent nodes of a KG (Ristoski et al., 2019b) in an *embedding matrix* used for downstream ML tasks. For this purpose, RDF2Vec uses a walking strategy to extract *walks* of a KG, where a walk is an n-tuple of nodes starting with a root node. In addition to this strategy, it is possible to use a sampling strategy to better deal with larger KGs (Cochez et al., 2017) by privileging some hops over others through edge weights. Once extracted, these walks are injected into an *embedding technique* to learn the embeddings of the provided root nodes of a KG and generate the corresponding embedding matrix.

Following the significant advances in Natural Language Processing (NLP), the community developed multiple embedding techniques. Among them, the release of Word2Vec in 2013 initially learns the vector representation of a word (Mikolov et al., 2013). However, KGs used this NLP technique where walks can be an analogy of sentences and nodes as words. Word2Vec

is the default embedding technique of RDF2Vec and has shown promising results (Ristoski et al., 2019a) in its use with KGs. However, other significant advances in the NLP field have taken place.

One of these advances was the creation of FastText, a Word2Vec extension to improve the obtained embeddings. Published in 2016, FastText has improved these embeddings in many use-cases at the expense of Random Access Memory (RAM) consumption and training time, but more recent techniques have emerged. Since 2018, Bidirectional Encoder Representations from Transformers (BERT) is the state-of-the-art NLP technique (Devlin et al., 2019) whose objective is to generate an unsupervised language representation model. This new technique outperformed Word2Vec in everyday NLP tasks (Saha et al., 2020; Beseiso and Alzahrani, 2020; Hendrycks et al., 2020). Due to its efficiency, the community published many variants of BERT, each bringing its specificity. However, BERT's benefits with KGs are still debatable. Few research papers have used this technique in KGs to compare it with other embedding techniques.

As a result of this finding, the main purpose of this Master's thesis is to provide a research work to focus on evaluating BERT with KGs based on Word2Vec and FastText. Such an evaluation would help determine its impact on the created embeddings and improve the generation of a 2D feature matrix from a KG.

To achieve this evaluation, an implementation of BERT and FastText will have to be made to the `pyRDF2Vec` library, a central Python implementation of the Java-based version of the RDF2Vec algorithm (Vandewiele et al., 2020a). In addition, since BERT works differently from Word2Vec, and FastText, it will be necessary to adapt the extraction and formatting of the walks of a KG to improve its training. Finally, the choice of hyperparameters will be a determining criterion on the accuracy of the BERT model, so it may be wise to find a compromise between training time and the model's accuracy.

Besides this primary objective, this Master's thesis proposes one more walking strategy and sampling strategy to those already provided by `pyRDF2Vec` (Cochez et al., 2017). Specifically, a walking strategy called `SplitWalker` uses a splitting function to pre-format the extracted walks before being injected into an embedding technique. In addition to this strategy, the `WideSampler` sampling strategy focuses on features shared between several entities by maximizing common relationships. Such additions could improve a model's accuracy and the extraction time for specific use cases according to a user's needs.

The result of this Master's thesis work is structured as follows. Chapter 2 introduces more context on the work related to BERT and RDF2Vec with KGs. Chapter 3 focuses on providing the specifications for this Master's thesis and stating the problems encountered. Chapter 4 provides background information on the fundamental concepts of graphs, ML, and NLP. This chapter also introduces advanced concepts such as the Attention mechanism and the Transformer architecture, both used by BERT. In addition, Chapter 5 focuses on covering in detail the functioning of each embedding technique, dedicating a section to the adaptation of BERT with a graph. From then on, Chapter 6 refers to RDF2Vec, including the walking strategies and the sampling strategies proposed by this Master's thesis. Chapter 7 describes the work done behind this Master's thesis, explaining the different implementations that were put in place. Chapter 8 focuses on providing the setup and results obtained from the different benchmarks related to the embedding techniques, walking and sampling strategies. Following these results, Chapter 9 discusses the correlation of the results with those of other research papers and provides some leads for future research with BERT. Finally, Chapter 10 is dedicated to the conclusion of this Master's thesis, summarizing the issues and solutions proposed.

Chapter 2

Related Work

This chapter presents the related work around BERT and RDF2Vec with KGs to understand better the study done by this Master's thesis.

2.1 BERT

Over the last few years after the publication of BERT (Devlin et al., 2019), several variants of this model emerged and were evaluated with its original version. One of these variants is DistilBERT, a smaller, cheaper, and lighter version of BERT. The comparison of DistilBERT with BERT and ELMo, another embedding technique, showed promising results across several data sets. Specifically, DistilBERT (Sanh et al., 2019) had almost as excellent performance as BERT surpassing ELMo's score in most data sets.

In the NLP related to biomedical and clinical texts, a benchmark of five tasks with ten datasets of different sizes and difficulties also compared BERT and ELMo (Peng et al., 2019). In this benchmark, BERT showed better overall results compared to ELMo. Another evaluation provides lower accuracy for BERT than GloVe when using a dataset related to tweets with or without sarcasm (Khatri and P, 2020). According to the authors, these lower BERT scores are due to the lack of context provided by the sarcastic tweets. Finally, another benchmark is interested in comparing BERT with ELMo, GloVe, and FastText by performing principal component static embeddings (Ethayarajh, 2019). In this use case, BERT shows the best results in most data sets.

Meanwhile, the community has released other KG-oriented variants of BERT, such as KG-BERT, released in 2019. This BERT-based framework achieves state-of-the-art performance in triple classification and link and relationship prediction tasks (Yao et al., 2019). Shortly after the publication of KG-BERT, K-BERT is also released, which has the particularity of not using pre-training. K-BERT uses BERT to infer relevant knowledge in a domain text to solve the lack of domain-specific knowledge within a general language representation from large-scale corpora (Liu et al., 2020). In this paper, K-BERT significantly outperforms BERT in NLP tasks from different fields such as finance, law, and medicine.

After the release of KG-BERT and K-BERT, others variants followed, such as Graph-BERT published in 2020. Graph-BERT relies solely on the Attention mechanism without using graph convolution or aggregation operations. This KG-variant also shows promising results overcoming Graph Neural Networks (GNNs) in the learning efficiency for node classification and graph clustering tasks (Zhang et al., 2020).

BERT-INT is also another variation of BERT, which predicts the identity of two entities across multiple KGs. BERT-INT works on proximity information to predict such identity without relying on the KG structure itself (Tang et al., 2020). Precisely, BERT-INT mimics entity comparison as humans would by comparing their name, description, and attributes.

When these two entities share the same information, a second comparison consists of assessing the similarity of neighbors between entities, but this time based on their name and description only.

Finally, the last paper uses BERT with Transfer Learning to answer questions based on a domain-specific KG (Vegupatti et al., 2020). Specifically in the field of medical biology using a dataset of 600 questions. In this paper, BERT succeeded in answering these questions with more than acceptable results.

The discoveries made around BERT suggest that the usage of this embedding technique with KGs is possible. However, too few research papers compare BERT with other embedding techniques. This lack of articles makes it unclear what the context of BERT's application is. The work of Khatri et al. gives a lead, but it is not enough.

2.2 Graph-Based Machine Learning

From a data modeling perspective, the use of KG since its publication has become a standard in the Semantic Web. This specific type of graph has its use in many fields to model a knowledge domain. However, despite the valuable information such a graph can provide, an ML model cannot directly learn from them (Ristoski et al., 2019b). Therefore, this decade proposed several techniques to convert KGs into embeddings for downstream ML tasks. In the field of KG embeddings, three categories of embedding creations are distinguished: *direct encoding* based, Deep Learning (DL) based, and *path/walk* based.

The first category includes algorithms such as RESCAL and Translating Embeddings (TransE) that mainly stand out in tasks related to graph completion as well as link prediction and entity classification, also called *node classification* (Bordes et al., 2013). It is still possible to perform mathematical operations with direct encoding while remaining the *embedded space* node classification result. This embedded space ensures the data embedding after dimensionality reduction. However, two of the drawbacks of direct encoding are their limited use with dynamic graphs and support for literals.

Modeling Relational Data with Graph Convolutional Networks (R-GCNs) mainly dominates the second category. R-GCNs is a supervised algorithm published in 2017 working directly on graphs. This algorithm illustrates DL's power in the Statistical Relational Learning tasks as link prediction and entity classification (Schlichtkrull et al., 2018). However, a significant disadvantage of R-GCNs is their memory consumption due to loading the KG upstream, limiting its use to KGs of reasonable size. Another drawback concerns their dependency on manual labeling of the training datasets due to the *supervised learning*, which can be time-consuming depending on its size. Besides that, the support of literals is theoretically possible, but only a few studies have investigated the subject.

The last category includes RDF2Vec, which is the state-of-the-art unsupervised algorithm since 2016. Unlike the other two categories, in this category, the creation of embeddings is done by traversing a KG using a walking and sampling strategy. Therefore, an essential drawback of RDF2Vec is that without caching and other optimization mechanisms, the walk extraction time can be significant for large KGs.

Each of these three categories of algorithms ensures the generation of embeddings. However, according to the use case, one category may be preferred to another. For example, since R-GCNs explicitly remove edges in a KG before training a model, RDF2Vec could be preferred. Furthermore, RDF2Vec can decide whether or not to prioritize a hop due to its walking and sampling strategy, which avoids training a model over the whole KG. Finally, these categories can support an online learning implementation and guarantee that a model remains up to date.

Chapter 3

Objectives

This chapter covers the specifications expected by this Master's thesis and the problems faced in its elaboration. In addition, each specification is followed by a summary of what was done to get a better idea of the overall report.

3.1 Specifications

In 2018, BERT was released and outperformed all other existing techniques on various ML tasks related to various fields. This Master's thesis aims to integrate BERT into `pyRDF2Vec`[1], but more specifically to evaluate its impact in terms of runtime, predictive performance, and memory usage using different combinations of walking and sampling strategies. Moreover, by reading literature from general graph-based ML and NLP research, inspiration can be found to design new sampling and walking strategies that result in improved predictive performances by exploiting the properties of BERT. The specifications of this Master's thesis are composed of the following four points:

1. **Support BERT and other embedding techniques for comparison purposes**: at least the BERT embedding technique should be integrated into `pyRDF2Vec` and be compared to the Word2Vec technique, which is currently used. Additionally, the extension of Word2Vec, namely FastText, will also have to be implemented and be compared to know its impact.

   For this Master's thesis, BERT and FastText have been implemented and integrated within the `pyRDF2Vec` library. Added to this implementation is a new architecture for `pyRDF2Vec` that easily allows other embedding techniques. Since Word2Vec already exists in the library, this Master thesis proposed a better walk extraction to improve the model's accuracy.

2. **Evaluate the impact of BERT**: A thorough benchmarking study to evaluate the impact of the BERT embedding technique on different dimensions will have to be conducted. For this, data sets having different properties and stemming from different domains will have to be selected. Moreover, a rigorous framework that performs the required experiments and logs all of these results will have to be developed.

   For this Master's thesis, BERT, Word2Vec, and FastText are compared on `MUTAG`, a small KG. In addition to these benchmarks of embedding techniques, the `pyRDF2Vec` library was modernized, allowing to get more information with the `verbose` parameter of the `RDF2VecTransformer` class.

[1] https://github.com/IBCNServices/pyRDF2Vec/

3. **Support of new walking strategies for RDF2Vec**: while an initial set of five simple walking strategies are already implemented in `pyRDF2Vec`, many other alternatives exist. Implementing more walking strategies would allow more extensive and detailed comparisons for embedding techniques.

   For this Master's thesis, the `SplitWalker` walking strategy is proposed and implicitly introduced some of its variants. Finally, this Master's thesis offers some benchmarks related to `SplitWalker` and other existing walking strategies.

4. **Support of new sampling strategies for RDF2Vec**: similarly, other sampling strategies can be created and implemented in `pyRDF2Vec`. Therefore, it will be possible to see the impact of the choice of a walking strategy and a sampling strategy with the performances related to BERT and other embeddings techniques.

   For this Master's thesis, the `WideSampler` walking strategy is proposed, leading to other variants of the latter. In addition, this Master's thesis evaluates `WideSampler` with other existing sampling strategies to knows its impact.

3.2 Problems Encountered

This section discusses the problems that have been encountered and why some benchmarks are missing. Initially, benchmarks related to bigger KGs such as `AM` and `DBP:Cities` should have been done. Therefore, it is helpful to understand the reasons for this to prevent these errors from recurring.

3.2.1 Garbage Collector

This Master's thesis being coupled with the internship, the same servers were used. Due to an internal problem with IDLab's server infrastructure, a loss of time equivalent to one and a half months of work was made. Therefore, this loss of time had repercussions on this Master's thesis. Specifically, the benchmarks related to BERT are not presented for this version of the report. However, they will be added for the next version. Moreover, some benchmarks had to be shortened due to time constraints.

For more information related to this issue, IDLab servers use the Stardog[2] infrastructure, which is implemented in Java. Therefore, the different benchmarks for the internship and Master's thesis recorded high variants. After several weeks of debugging, thinking that this was a `pyRDF2Vec` issue, a Stardog engineer confirmed that the issue was related to their infrastructure. In more detail, the concern was the lack of RAM release from the servers due to garbage collection. As a result, new data was being saved directly to a physical disk, which resulted in much higher latencies than RAM. From then on, this was reflected in the variance of the results.

3.2.2 Exceptions Raised

In addition to the problems related to the garbage collector, exceptions are encountered from time to time during SPARQL Protocol and RDF Query Language (SPARQL) queries. These exceptions are due to the inaccessible locally hosted SPARQL endpoint during updates or internal network problems. As a result, after more than three attempts to retrieve the walks of a provided entity, an exception is thrown, which makes the benchmarks stop. Since these benchmarks can take several days, it also delayed other benchmarks.

[2] https://www.stardog.com/

Chapter 4

Background

This chapter ensures that the reader understands the vocabulary used in this document. Specifically, this chapter contains the following five parts:

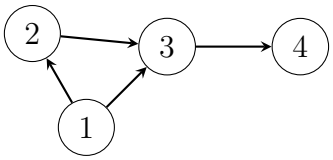
1. **Graphs**: covers the basic graph types and defines the KG.
2. **ML**: describes the three basic ML paradigms as well as some activation functions.
3. **NLP Techniques**: defined some notions to understand the embedding techniques better.
4. **Attention**: reviews the basics of the attention mechanism helpful in understanding the Transformer architecture.
5. **Transformer**: introduces the Transformer architecture, used used by many recent embedding techniques such as BERT.

4.1 Graphs

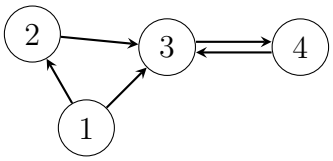
Graphs have been used long before the introduction of ML and cover a wide range of applications. To better understand the precise vocabulary of graphs, this section provides some definitions.

Definition 4.1.1 (Graph)**.** Ordered pair (V, E), where V is a finite and non-empty set of elements called *vertices* (or *nodes*), and E is a set of unordered pairs of distinct nodes of V, called *edges*.

Basic types of graphs include:

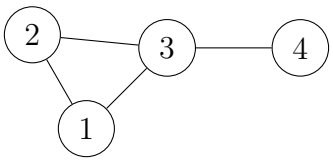


Definition 4.1.2 (Oriented Graph)**.** Directed Graph where bidirected edges connect no pair of vertices.

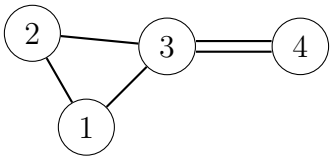


Definition 4.1.3 (Directed Graph)**.** Named *digraph*, it is defined as a graph whose edges have an orientation, also known as *directed edges*, *directed links*, *arrows*, or *arcs*.

Figure 4.1: Basic Graph Types (Part I).



Definition 4.1.4 (Undirected Graph)**.** Graph whose edges are bidirectional.



Definition 4.1.5 (Multigraph)**.** Undirected graph that can store multiple edges between two nodes.

Figure 4.2: Basic Graph Types (Part II).

Based on these definitions of graph types, the KG is defined.

Definition 4.1.6 (Knowledge Graph)**.** Directed *heterogeneous* multigraph whose node and relation types have domain-specific semantics (Kamakoti, 2020). These nodes can be of different types. From a terminology point of view, the nodes/vertices of a KG are often called *entities*, and the directed edges refer to as predicates. Moreover, this multigraph has *triple*[1] designing a 3-tuple, where each triple defines a (`subject`, `predicate`, `object`) tuple[2]. For ease of processing, the multigraph aspect of the KG can be removed by representing each triple as two 2-tuple: (`subject` → `predicate`) and (`predicate` → `object`).

4.2 Machine Learning

Machine Learning (ML) is a branch of AI focusing on improving the automatic learning of models without having been explicitly programmed for it. ML comes with three basic paradigms: supervised, unsupervised, and reinforcement learning. Each of them is defined below.

Definition 4.2.1 (Supervised Learning)**.** Type of ML with human supervision, where a model looks for patterns in a data set with labels (Wood, 2020).

Definition 4.2.2 (Unsupervised Learning)**.** Type of ML with minimal human supervision, where a model looks for patterns in a data set without labels.

Visually, supervised learning and unsupervised learning are defined as follows:

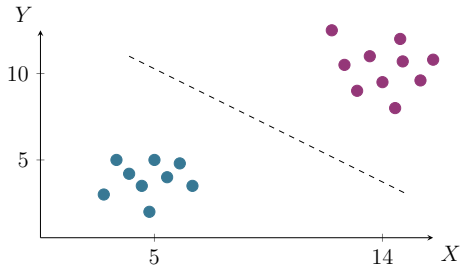


Figure 4.3: Supervised Learning

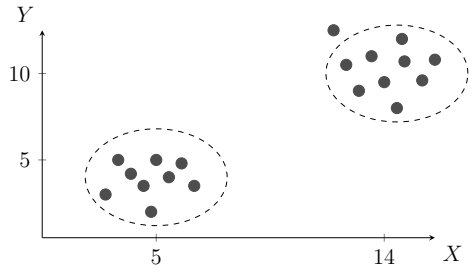


Figure 4.4: Unsupervised Learning

Figure 4.5: Learning a Model Using Raspberry and Blueberry Data.

In Figure 4.3, supervised learning contains raspberry and blueberry data already classified by their label. A model learns to predict the values based on a *cost function*, which allows it to

[1] Also called *triplets*.

[2] Some authors define this tuple as `(h, r, t)`. In this definition, `h` is the head entity, `t` the tail entity, and `r` the relation associating the head with the tail entities.

check and correct its predictions according to the actual values. With these labeled data, the classification and regression problems can use this type of learning.

In Figure 4.4, unsupervised learning still contains raspberry and blueberry data. However, this time these data are not labeled, which means that a model must find the patterns and structure independently. The clustering and association problems use this type of learning. For example, this clustering example creates two clusters, with one raspberry not belonging to any cluster.

Definition 4.2.3 (Reinforcement Learning)**.** Type of ML where a model learns from the experience and feedback of an autonomous agent.

In ML, each neuron inside an Artificial Neural Networks (ANN) outputs a number between $-\infty$ and ∞ propagating to its predecessor via an activation function. Once the ANN has completed its processing through these neurons, it generates *logits*, a non-normalized vector of predictions generated by a classification model. For ease of processing, it is usually appropriate to convert these logits into probabilities to have a unitary sum. According to the required type of classification, activation functions such as *softmax* and *sigmoid* are helpful. These functions can help with various tasks, such as predicting the most likely word to be the missing word in a sentence in the Natural Language Processing (NLP) field.

Definition 4.2.4 (Softmax Function)**.** Also called *softargmax*, a logistic regression model uses this multi-classification function for transformation purposes. Specifically, this model transforms a vector of K real values into a vector of K elements that range between 0 and 1 with the particularity of having a unitary sum (Wood, 2019). Let $\mathbf{z}$ be an input vector and K be some classes in a multi-class classifier. Mathematically, the softmax function of these classes is defined as follows:

$$\sigma(\mathbf{z})_i = \frac{\mathrm{e}^{z_i}}{\sum_{j=1}^{K} \mathrm{e}^{z_j}} \tag{4.1}$$

Definition 4.2.5 (Sigmoid Function)**.** Binary classification function recognizable with an "S" shaped curve used by a logistic regression model for transformation purposes. Unlike the softmax function, this model transforms a vector of K real values into a vector of K elements range this time between -1 and 1 with still the particularity of having a unitary sum. Let $\mathbf{x}$ be an input vector. Mathematically, the sigmoid function is defined as follows:

$$\sigma(\mathbf{x}) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1} \tag{4.2}$$

Definition 4.2.6 (Rectified Linear Unit)**.** Commonly called *ReLU*, the community considers this function as one of the most straightforward functions. Let $\mathbf{x}$ be an input vector. Mathematically, the ReLU function is defined as follows:

$$\mathrm{f}(\mathbf{x}) = \max(0, \mathbf{x}) \tag{4.3}$$

These functions can help with various tasks, such as predicting the most likely word to be the missing word in a sentence in the NLP field. Subsequently, in ML, it is essential to know the similarity of two vectors. For example, it is helpful in NLP to tell if two words share semantic similarities. As a result, the use of cosine similarity is relevant.

Definition 4.2.7 (Cosine Similarity)**.** Measures the cosine of the angle between two non-zero vectors of an inner product space (DeepAI, 2019a). Let $\mathbf{u}, \mathbf{v} \in \mathbb{R}^d$ be two non-zero d-dimensional vectors. Mathematically, the cosine similarity between $\mathbf{u}$ and $\mathbf{v}$ is defined as follows:

$$\cos(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \, \|\mathbf{v}\|} = \frac{\sum_{i=1}^{n} u_i v_i}{\sqrt{\sum_{i=1}^{n} u_i^2} \sqrt{\sum_{i=1}^{n} v_i^2}} \tag{4.4}$$

where $\cos(\mathbf{u}, \mathbf{v})$ produces a value ranging from -1 to 1. Specifically, the cosine similarity returns -1 if two vectors do not have any similarities, 0 if they are unrelated, and 1 if they share every similarity.

4.3 Natural Language Processing Techniques

There are many NLP techniques. The objective is not to cover every one of them but to define certain notions used by more advanced concepts such as Word2Vec.

Definition 4.3.1 (Distributed Representation)**.** Describes the same data features across multiple scalable and interdependent layers (DeepAI, 2019b). In a distributed word representation, there is a distribution of the word information across vector dimensions.

Definition 4.3.2 (Bag-of-Words)**.** Vectorizes a text by counting the number of unique words (also called *tokens*) in a text. Let "Suikoden is my favorite game, it is a wonderful game!" be a text. The representation of this text with Bag-of-Words (BoW) is defined as follows:

$$\{\ \text{Suikoden: 1, is: 2, my: 1, favorite: 1, game: 2, it: 1, a: 1, wonderful: 1}\ \} \tag{4.5}$$

for which this text can be characterized (e.g., `[1, 2, 1, 1, 2, 1, 1, 1]`) by differences measures (e.g., word frequency).

Definition 4.3.3 (One-Hot Encoding)**.** Quantifies categorical data as binary vectors. Specifically, the belonging of a data point to the ith category implies the acquisition of a zero value for the components of this vector, except for the ith component, which receives a unitary value. Let K be several categories in a data set, and $\mathbf{y}^{(i)}$ be a data point in the ith class. Mathematically, the following vectorial representation defines such one-hot encoding:

$$\mathbf{y}^{(i)} = \underbrace{\begin{bmatrix} 0 & \dots & 0 & \underbrace{1}_{\text{index } i} & 0 & \dots & 0 \end{bmatrix}^{\top}}_{K \times 1} \tag{4.6}$$

where one-hot encoding vectors allow ML algorithms to make better predictions. Such an encoding does not capture words' semantic and syntactic information. Therefore, it does not detect semantic and order difference between the sentences the "I like cats more than dogs" and "I like dogs more than cats" As a result, *word embeddings* are privileged to detect these differences and allow a better numerical representation of words.

Definition 4.3.4 (Word Embeddings)**.** Unsupervised model that captures words' semantic and syntactic information using an *embedding matrix*, where the embeddings of a w word are a vector $\mathbf{v}_w$.

Definition 4.3.5 (Embedding Matrix)**.** Randomized matrix of dimensions $\mathcal{W} \times \mathcal{F}$, where $\mathcal{W}$ is the number of unique words in a document and $\mathcal{F}$ is the number of features that each unique word in this vocabulary has. The *gradient descent* uses these matrix values to find the minima of a function for several *epochs*, the number of complete cycles on a training data set. From then on, closer words in vector space are assumed to have a similar meaning.

Definition 4.3.6 (Window Size)**.** Determines the context words, also called *training samples*, of a target word from a sliding window along with a sentence. Therefore, a window size of two means 2-triple context words, including the related target word and each of the two words on its left and two on its right. Let "I will always remember her" be a sentence. The following table defines the context words for a window size of 2:

Table 4.1: Context Words Determination for a Window Size of 2.

Input Text	Target Word	Context Words
I will always remember her	i	will always
I will always remember her	will	i always remember
I will always remember her	always	i will remember her
I will always remember her	remember	will always her
I will always remember her	her	always remember

In Table 4.1, the target word highlighted in blue is modified at each iteration starting from left to right, considering two words forward and backward (highlighted in a lighter blue). As such, the context for a given sentence are known.

Definition 4.3.7 (Stop Words)**.** Commonly used words (e.g., "an", "the", and "is") having little value for training a model and are therefore considered noise in a training data set.

4.4 Attention

The *Attention* is a Deep Learning mechanism published in 2014 by Bahdanau et al. to solve the bottleneck issue of Recurrent Neural Networks (RNNs) sequential models widely used for neural machine translation (Chorowski et al., 2015). To fully understand its mechanism, it is helpful to start by introducing the functioning of RNNs.

4.4.1 Recurrent Neural Networks

Before using such a mechanism, RNNs caused accuracy losses for a model when processing long input sequences mainly due to the way RNN encoders generated the context vector. Added to that, RNNs are difficult to parallelize.

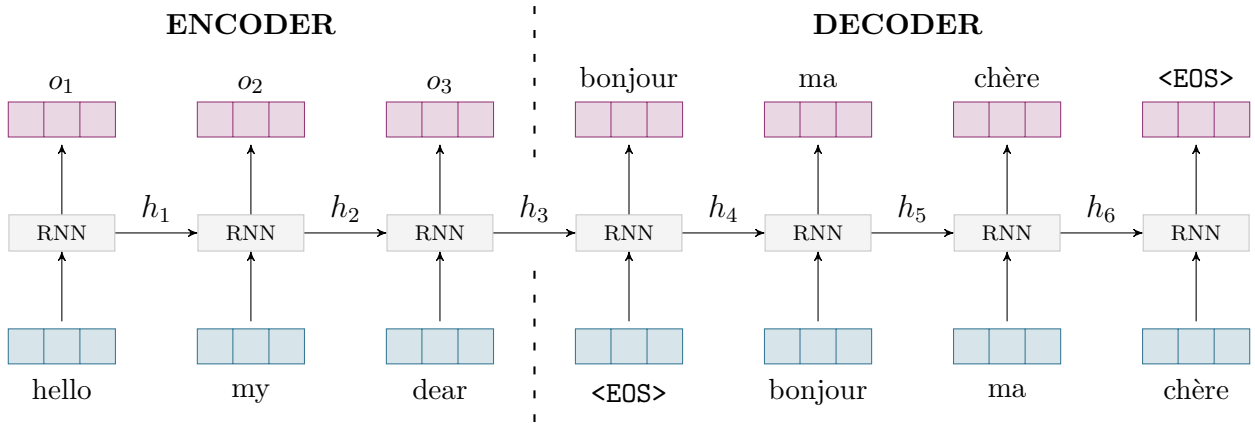


Figure 4.6: Sequence-to-Sequence Learning With RNNs.

In Figure 4.6, a Sequence to Sequence (Seq2Seq) model translates an English sentence into French using RNN encoders and decoders. Each time step has an RNN unit containing an activation function that takes a word embedding and a hidden state as input for both encoding and decoding. This hidden state serves as a memory to save the entire previous context. Specifically, the hidden state at the t time step is computed based on the hidden state at the $t-1$ time step and the current word embedding. Once one RNN encoder reaches the first End-Of-Sentence (`<EOS>`), the RNN decoder receives a context vector containing the last generated hidden state, namely h_3. Finally, each RNN decoder unit translates a word based on this context until to reach once more the `<EOS>` token.

Although RNNs are effective for small sequences, this is not the case for more extensive sequences. This mechanism uses a fixed size context vector and generates the encoder's hidden states based on the previous hidden state. Consequently, the last words of an input sequence have a greater weight than the first ones. From this unbalanced weight, processing a long sequence by a model comes at the cost of forgetting the earlier parts of that sequence, resulting in a loss model's accuracy. RNN variants emerged to reduce this waste of information, such as Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU). These variants helped to improve the model's accuracy, but the Attention mechanism came up with an interesting idea.

4.4.2 Mechanism

The basic idea of the Attention mechanism is not only to pay attention to each input word in the context vector but also to give a relative importance to each of them (Chorowski et al., 2015). In other words, the Attention mechanism focuses on matching input and output elements. After its publication, this mechanism became one of the design choices in many NLP and Computer Vision tasks. Computer Vision is a field of Artificial Intelligence where the computer learns digital images or video content. This mechanism has received other variants (Luong et al., 2015), which increased the model's accuracy in most of the benchmarks that have been performed. Finally, due to the popularity of the Attention mechanism, the use of RNNs has been questioned many times. However, RNNs are still present in everyday life through various voice assistance applications such as Apple's Siri, Amazon Alexa, and Google Home.

With the Attention mechanism, the context vector includes each encoder's hidden state. In addition, each decoder's hidden state processes some additional calculation to achieve a better model's accuracy compared to the use of RNNs without Attention (Alammar, 2018b). This mechanism mainly solved the previous issues related to the lack of parallelization and forgetting previous word contexts for long sequences.

Visually, the mechanism works as follows:

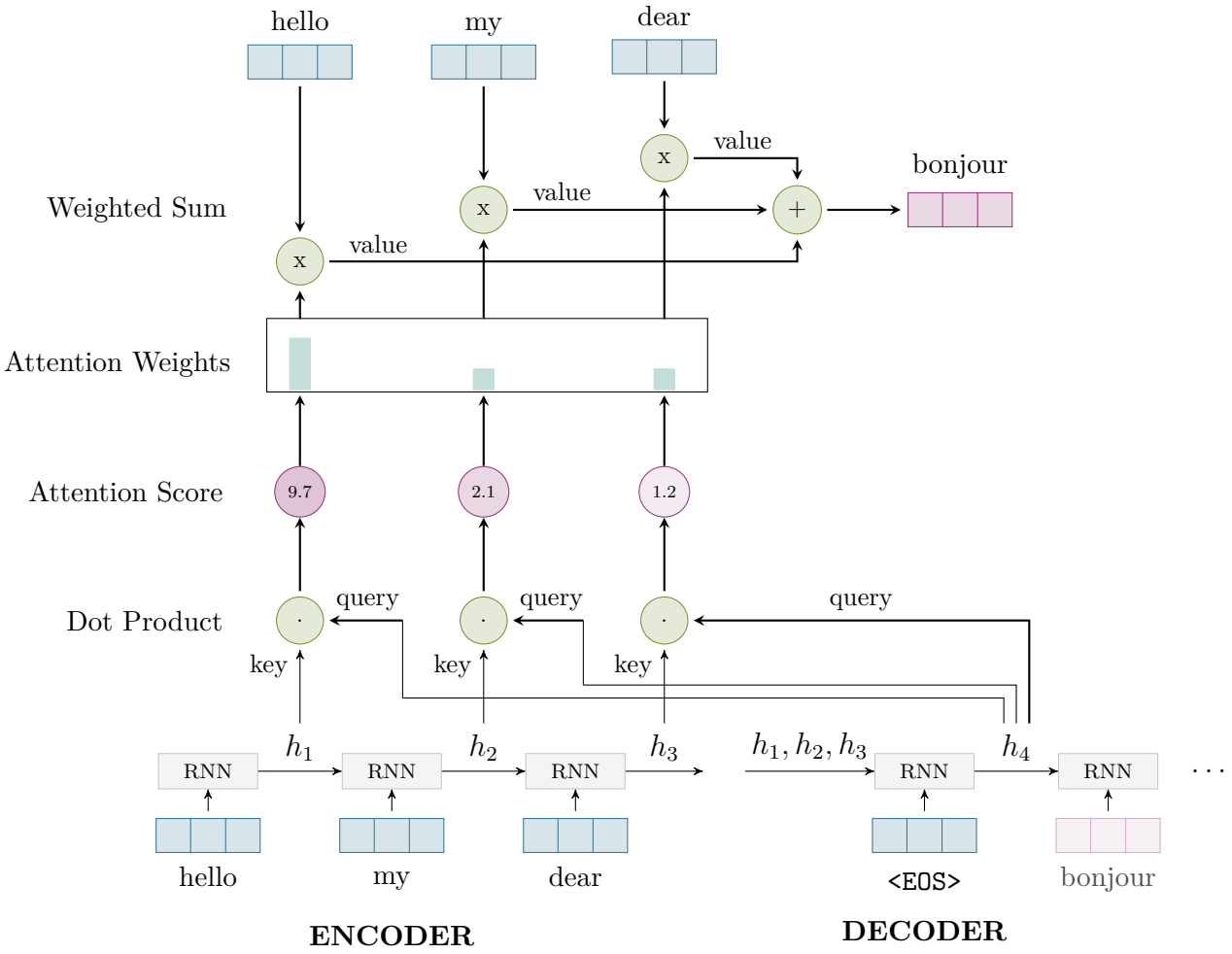


Figure 4.7: Seq2Seq Learning With Attention Mechanism.

In Figure 4.7, the Attention mechanism starts using the encoder RNNs to generate their hidden states, similar to Figure 4.6. Once these hidden states have been generated, the context including these hidden states is provided to each time step of the decoder RNNs. For each RNN decoder block, the hidden state of decoding is calculated in three significant steps:

1. **Computation of the Attention/Alignment Score for each word**: to know the most likely word to pay more attention to the translation of the current word, a dot product is made between the previous decoding hidden state with every encoding hidden state.

   The original research paper defines *query*, *key*, and *value* by making an analogy with a database. The query allows performing a search (e.g., book) associated with a set of keys (e.g., book title and abstract) for which each key is associated with a value. From then on, the *Dot-Product Attention* consists of computing the weighted matching between m queries and n keys in an m by n matrix.

2. **Computation of the Attention Weights**: the Attention scores are normalized with a softmax function in such a way as to ease their processing using a probability distribution whose sum is unitary.

   Definition 4.4.1 (Attention Score)**.** Let $h_1, \ldots, h_N \in \mathbb{R}^h$ be encoder hidden states, $s_t \in \mathbb{R}^h$ be the decoder hidden state, and t be the time step. Mathematically, the a_t

Attention score is defined as follows:

$$\alpha^t = \text{softmax}\left(\left[s_t^T h_1, \ldots, s_t^T h_N\right]\right) \in \mathbb{R}^N \tag{4.7}$$

$$\mathrm{a}_t = \sum_{i=1}^{N} \alpha_i^t h_i \in \mathbb{R}^N \tag{4.8}$$

3. **Computation of the weighted sum**: After normalization, it is necessary to multiply each embedded word with its weight to compute the hidden state.

After the calculation of the hidden states, Attention picks the most likely word for translation. This mechanism gives importance to these words by mapping them to a higher or lower weight depending on the word's relevance. Therefore, this improves the accuracy of the output predictions (Chorowski et al., 2015). However, the Attention mechanism in Figure 4.7 cannot be used in a neural network since there are no weights to train by a model. In practice, the Attention mechanism uses three weight matrices to multiply each key, value, and query by their respective matrices called the key, query, and value matrices.

Following the publication of Attention, another attention mechanism emerged which gives attention within input elements. *Self-Attention* also called *intra-attention*, is a Attention mechanism defines in 2016 and relates different positions of a single sequence to compute a representation of the same sequence (Cheng et al., 2016). Unlike the primary Attention mechanism, where the query, key, and value matrices may differ, Self-Attention has these three matrices identically as generated from the same input sequence. Based on a gradient signal, each self-attention block can use this signal to propagate information to the weight matrices, namely the key, query, and values matrices. It is largely thanks to Self-Attention that the *Transformer* architecture was created.

4.5 Transformer

The *Transformer* architecture is an alternative to RNNs published in 2017, questioning the need for RNNs following the discovery of the Attention mechanism (Vaswani et al., 2017). The original paper presents this architecture for Machine Translation and introduces two new Attention mechanisms: *Scaled Dot-Product Attention* and *Multi-Head Attention*.

4.5.1 Scaled Dot-Product Attention

Scaled Dot-Product Attention provides an Attention score based on a Dot Product Attention scaled by the square root inverse of the dimension of the query and key matrices. This scaled version prevents the Dot-Product Attention mechanism from growing large in magnitude when the dimensionality of the queries has a high value. Specifically, this high value is due to the multiplication and the addition of inputs, which push the softmax function into regions where its gradients are minimal.

Definition 4.5.1 (Scaled Dot-Product Attention)**.** Let $Q \in \mathbb{R}^{m \times d_k}$ be the matrix of queries and $K \in \mathbb{R}^{n \times d_k}, V \in \mathbb{R}^{m \times d_v}$ be matrices of a set of key-value pairs. Mathematically, the the Scaled Dot-Product Attention mechanism is defined as follows:

$$\mathrm{A}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \tag{4.9}$$

where d_k is the dimensionality of queries as well as keys and d_v the dimensionality of values.

4.5.2 Multi-Head Attention

Multi-Head Attention (MHA) is an improved version of the Self-Attention mechanism where each word not only brings excessive importance to itself, but pays attention to the interactions with other words. As a result, MHA achieves better results than Self-Attention, where the latter leads to a loss of efficiency in the Attention embeddings. Finally, MHA still performs a weighted average to generate the Attention vector for each token. However, the heads are this time computed in parallel.

Definition 4.5.2 (Multi-Head Attention)**.** Let $f_o : \mathbb{R}^{kd} \to \mathbb{R}^D$ be a linear map, $f_{i,q}, f_{i,k}, f_{i,v} : \mathbb{R}^D \to \mathbb{R}^d$ be a three sets of linear maps, $\hat{Q} \in \mathbb{R}^{M \times D}$ be a queries vector, and $\hat{K}, \hat{V} \in \mathbb{R}^{N \times D}$ be vectors of keys and values. Mathematically, the MHA is defined as follows:

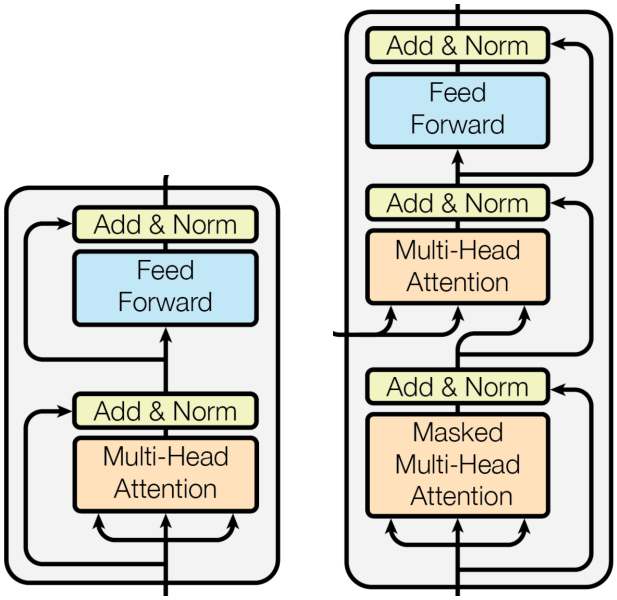
$$\text{MHA}(\hat{Q}, \hat{K}, \hat{V}) = f_o \left(\text{concat} \left(\left[A \left(f_{i,q}(\hat{Q}), f_{i,k}(\hat{K}), f_{i,v}(\hat{V}) \right) \ \forall\, i \in 1, \ldots, k \right] \right) \right)$$

where $\text{MHA}(\hat{Q}, \hat{K}, \hat{V}) \in \mathbb{R}^{M \times D}$ is the output of the MHA, also known as a *neural function* with k-headed attention block.

4.5.3 Architecture

The Transformer architecture mainly consists of an encoder and a decoder block, including the Scaled Dot-Product and MHA mechanisms. Specifically, the encoder and decoder are neural components composed of a stack of $N = 6$ identical layers.

Visually, one layer of the Transformer works as follows:



(a) Encoder (b) Decoder

Figure 4.8: One Layer of the Architecture of the Transformer.

Source: Vaswani et al. – Attention Is All You Need.

In Figure 4.8a, one layer of the encoder contains two sub-layers: a MHA mechanism and a position-wise fully connected Feed-Forward Neural Network (FFN).

Definition 4.5.3 (Feed-Forward Neural Network)**.** Let $W_1 + b_1$ and $W_2 + b_2$ be two linear transformations. Mathematically, the FFN is defined as follows:

$$\text{FFN}(x) = \max\left(0, xW_1 + b_1\right) W_2 + b_2 \tag{4.10}$$

where a ReLU activation function is used.

Moreover, a residual connection followed by a normalization of the layers wraps each of the two sublayers. Mathematically, the output of every sub-layer is defined as follows:

$$\text{LayerNorm}(x + \text{Sublayer}(x)) \tag{4.11}$$

where $\text{Sublayer}(x)$ refers to the implemented function by the sub-layer itself.

In Figure 4.8b, the decoder has layers that differ from the encoder with two sub-layer changes. The first sublayer change is defined using *Masked MHA* as the first sub-layer to prevent positions from attending subsequent positions. The last change is adding a new sub-layer, called *Encoder-Decoder Attention* which performs MHA over the Attention vectors of the decoder and those of the encoder. The use of this sub-layer makes it possible to determine the different relationships between these vectors. Finally, the generated embeddings are sent to a position-wise fully connected FFN. Like the encoder, each sub-layer is wrapped by a residual connection followed by a layer normalization.

From then one, the complete architecture of the Transformer is illustrated as follows:

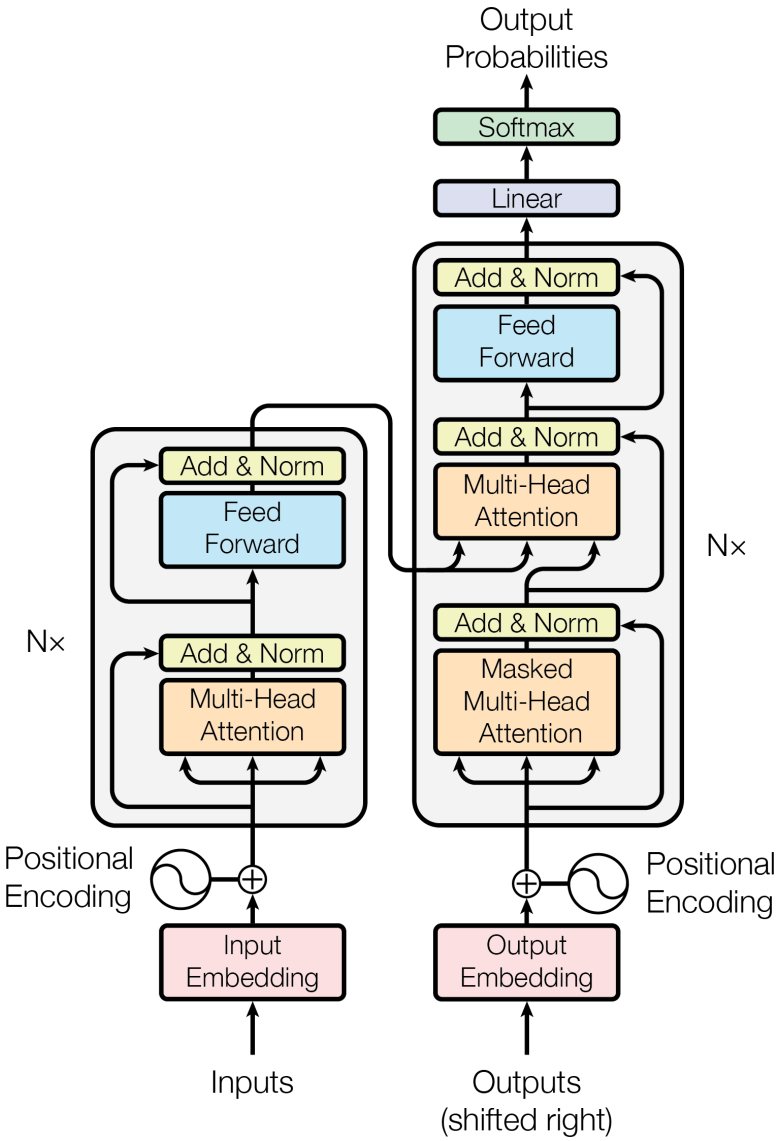


Figure 4.9: Model Architecture of the Transformer.

Source: Vaswani et al. – Attention Is All You Need

In Figure 4.9, both encoding and decoding take the sum of the input/output embeddings with positional embedding as input. Unlike RNNs, Transformer do not have a time step so that input sequences can be injected simultaneously and converted into input embeddings. Each token is represented in the embedding space through these input embeddings where semantically similar tokens are closer than others. However, the decoder has the particularity of having its input shifted. This shift avoids that a model only learns to copy the decoder input by allowing a model to predict the target word/character for position i according to the previous one from 1 to $i-1$.

At last, the output of the decoding layer is sent to a linear layer. This layer is nothing more than an FFN layer whose objective is to extend the dimension of this vector to the number of words of the language concerned. Once expanded, this vector is submitted to a softmax activation function to transform it into a probability distribution and predict the next word according to the word with the highest probability. Afterward, this decoding process is iterated several times until the end-of-sentence token is generated.

4.5.4 Positional Encoding

Considering that the transformer has no recurrence or convolution mechanism, the architecture cannot know the order in an input sequence. From then on, the transformer encodes the same meaning for different sentences. The easiest way to take this context into account is to encode these positions as one-hot features.

Definition 4.5.4 (Positional Encoding as One-Hot Features)**.** Let $x \in \mathbb{R}^{n \times d}$ be a matrice of sequentially ordered data along the n-dimensional axis, and e_k be a k'th standard basis vector in $\mathbb{R}^n$. Mathematically, the $z \in \mathbb{R}^{n \times d}$ learned combined representation is defined as follows:

$$\mathrm{z}_k = W_z^T ReLU\left(W_x^T x_k + W_e^T e_k\right), W_x \in \mathbb{R}^{dim(x) \times m}, W_e \in \mathbb{R}^{n \times m}, W_z \in \mathbb{R}^{m \times d} \tag{4.12}$$

Another approach for positional encoding is to build distinct representations of inputs and positions (Gehring et al., 2017). Despite these existing approaches to differentiate meanings, the original Transformer paper uses a sinusoid-wave-based positional encoding to inject absolute positional information of tokens into the sequence.

Definition 4.5.5 (Positional Encoding by Vaswani et al.)**.** Let d_{model} be the embedding dimension of words, and $pos \in [0, L-1]$ be the position of a w word in the $w = (w_0, \ldots, w_{L-1})$ input sequence. Mathematically, the positional encoding of w is defined as follows:

$$\mathrm{PE}(pos, i) = \begin{cases} \sin\left(\dfrac{pos}{10000^{2i/d_{model}}}\right), & i = 2k \\ \cos\left(\dfrac{pos}{10000^{2i/d_{model}}}\right), & i = 2k+1 \end{cases} \qquad k \in \mathbb{N} \tag{4.13}$$

where the positional encoding follows a specific, learned pattern to identify word position or the distance between words in the sequence (Alammar, 2018a). In Equation 4.13, the sinusoidal representation works as well as a learned representation and better generalizes sequences that are longer than the training sequences (Vaswani et al., 2017).

Chapter 5

Embedding Techniques

This chapter explains the Word2Vec, FastText, and Bidirectional Encoder Representations from Transformers (BERT) embedding techniques.

5.1 Word2Vec

Word2Vec is an unsupervised learning algorithm published in 2013, which, based on a corpus of text, provides a vector representation for each word of that text, using a two-layer neural network. This algorithm was initially implemented to learn word representations from large data sets efficiently but emerged as a standard embedding technique. According to the use case, two models are available: Continuous Bag-Of-Words (CBOW) and Skip-Gram (SG).

5.1.1 Continuous Bag-of-Words Model

CBOW is an unsupervised model that predicts a target word based on context words and window size. From a semantic point of view, the model is named "Bag-of-Words" since the order of the context words does not influence its training.

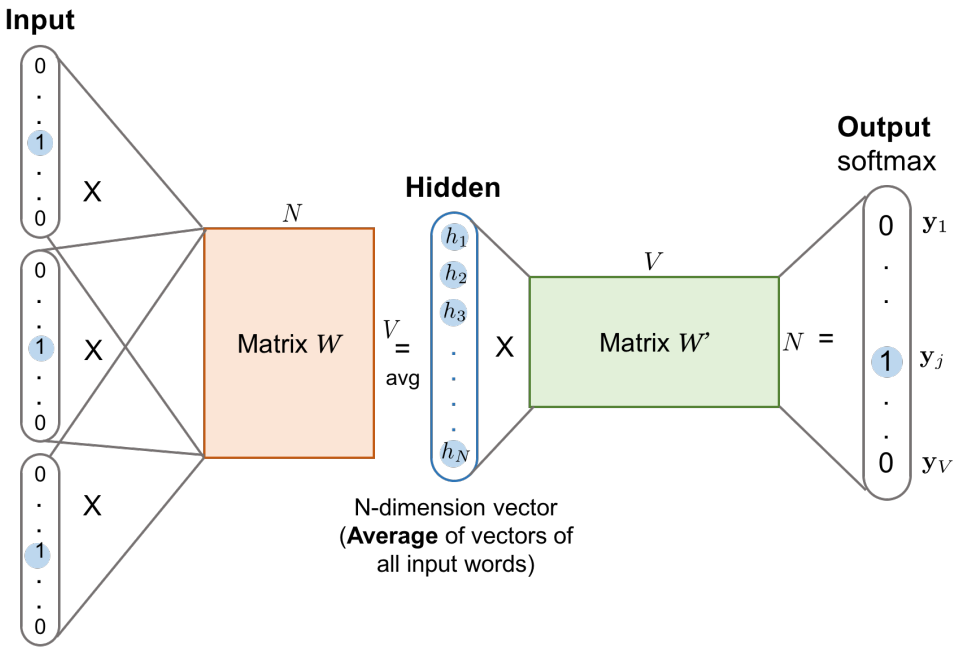


Figure 5.1: CBOW Model Architecture.

Source: Lilian Weng – Learning Word Embedding

In Figure 5.1, the CBOW model architecture starts by taking one or several targets word as input, represented as a V-dimensional one-hot encoding vector. V is the dictionary's size of unique words present in a text or in a training data set. Using a dot product, this/these input vector(s) is then multiplied by a first W weight matrix, named embedding matrix. This matrix is of dimension $V \times N$, where N is the number of features that each unique word in this vocabulary has defining the *embedding size*. Therefore, these dot products produce an N-dimensional hidden vector for each input vector. However, if there multiple input context vectors, these hidden vectors are then averaged element-wise into a single hidden vector, where this vector chooses the size of the vectors that will be used later.

According to the semantics of the words of the dictionary, a new dot product is made, but this time, between the calculated hidden vector and another W' weight matrix, named *context matrix*. This matrix being now of dimension $N \times V$. Then, their product generates an output vector of dimension V subjected to a softmax activation function. Therefore, this function ends by calculating a probability distribution and returns the probability that a word is a target word for the given context words. Finally, a cross-entropy loss function is applied to compute the loss between the true probability distribution of the given target word and the calculated model probability.

Definition 5.1.1 (Cross-Entropy)**.** Measures the difference between two probability distributions for a given random variable. Let p be a true probability distribution, q be a computed model probability, and c be the predicted result by an ML algorithm. Mathematically, the cross-entropy loss can be defined as follows:

$$H(p, q) = -\sum_{c=1}^{C} p(c) \log\left(q(c)\right) \tag{5.1}$$

After computation with the cross-entropy loss, the back-propagation updates the weights of the embedding matrix according to this loss. These steps are then repeated with other context words and another target word for a specified number of epochs. Once the training is done, the embedding matrix is used to generate the word embeddings from the one-hot encodings. Formally, given a sequence of training words $w_1, w_2, \ldots, w_t$, and a context window c, the objective of the CBOW model is to maximize the average log probability:

$$\frac{1}{T} \sum_{t=1}^{T} \log\left(p\left(w_t | w_{t-c}, \ldots, w_{t+c}\right)\right) \tag{5.2}$$

where T is the number of training samples, and the probability $p\left(w_t | w_{t-c}, \ldots, w_{t+c}\right)$ is computed using the softmax function (cf. Definition 4.2.4):

$$p\left(w_t | w_{t-c}, \ldots, w_{t+c}\right) = \frac{\exp\left(v^{-T} v'_{w_t}\right)}{\sum_{w=1}^{W} \exp\left(v^{-T} v'_w\right)} \tag{5.3}$$

where W is the vocabulary size, v'_w is the output vector of the w word, and v is the averaged input vector of all the context words:

$$\overline{w}_t = \frac{1}{2c} \sum_{-c \leq j \leq c, j \neq 0} w_{t+j} \tag{5.4}$$

Finally, the whole network is not used regarding the generation of embeddings, but only its first layer.

5.1.2 Skip-Gram Model

SG is an unsupervised model that predicts context words from a target word, according to window size and a sentence. Unlike CBOW, these context words are defined as multiple pairs of words (cf. Table 4.3.6), serving as training samples for this model. Such training allows this model to learn word vector representations suitable for predicting nearby words within a text, except for common words and stop words (cf. Section 5.1.3). Finally, SG compresses the information in a low dimensional space, such that this model learns a continuous low dimensional representation for the words.

The window size for each target word is arbitrarily chosen between one and a small positive number. This choice allows to add randomness during training and improves future predictions. In the case of a window size greater than two, the context words for a target word likely are more than two. If such a situation occurs, SG randomly picks a word from these word contexts to form a pair with the context word. Then, this model counts the number of times this pair of words appear, and the model starts to be trained.

Similar to CBOW, given a sequence of target words $w_1, w_2, \ldots, w_t$, and a context window c, the objective of the SG model is to maximize the average log probability:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log\left(p\left(w_{t+j}|w_t\right)\right) \tag{5.5}$$

where T is the number of training samples, and the probability $p\left(w_{t+j}|w_t\right)$ is also computed using the softmax function (cf. Definition 4.2.4):

$$p\left(w_o|w_i\right) = \frac{\exp\left(u_{w_o}^T v_{w_i}\right)}{\sum_{w=1}^{W}\exp\left(u_w^T v_{w_i}\right)} \tag{5.6}$$

where W is the vocabulary size, v the input representation of a word, and u the output representation of a word.

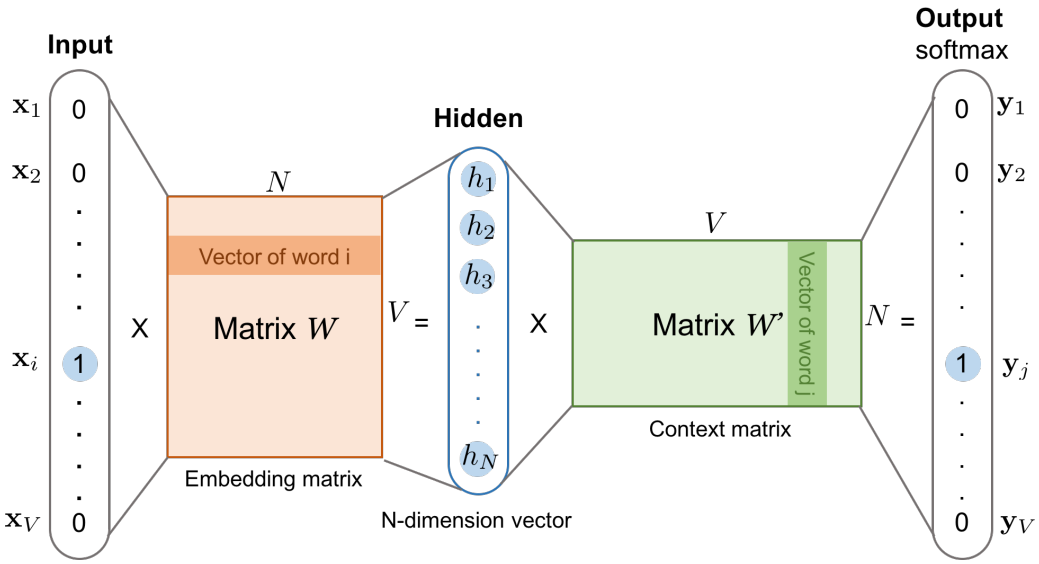


Figure 5.2: SG Model Architecture.

Source: Lilian Weng – Learning Word Embedding

In Figure 5.2, the SG model architecture can be seen as the inverse of the CBOW architecture. With this architecture, instead of having context words and predicting a target word,

it takes two words. From then on, a model will predict a target word according to a context word. In addition, since there is only one input word, the embedding matrix directly generates a hidden layer with no need to do an average. Finally, the cross-entropy loss function is applied this time to compute the loss between the true probability distribution of the given context word and the calculated model probability.

5.1.3 Subsampling Frequent Words

Subsampling frequent words is a technique whose objective is to reduce the number of training examples for a Word2Vec model. To accomplish this, it assumes that predicting context words that are too frequent (e.g., "the") provides little semantic value to differentiate a context (Mikolov et al., 2013). In contrast, infrequent words are more likely to convey specific information. Therefore, to improve the balance between infrequent and frequents words, this technique randomly eliminates words from a more frequent corpus than a certain threshold. This suppression occurs before a corpus is processed in word-context pairs.

Let w_i be a word, f_{w_i} be a word frequency in a corpus, and t be a chosen threshold, typically around 10^{-5}. Mathematically, the discarding of a word from a corpus is done as follows:

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}} \tag{5.7}$$

Therefore, each word w_i present in a text for a window of size c has a probability of being deleted, where each deleted word reduces the training samples by most c times. Therefore, words whose frequency is above this threshold, will be aggressively subsampled, preserving the frequency ranking. However, even though this formula speeds up learning and even significantly improves the accuracy of learned vectors of infrequent words (Mikolov et al., 2013), the Word2Vec implementation uses another more elaborate formula to discard a word:

$$P(w_i) = \left(\sqrt{\frac{z(w_i)}{t}} + 1\right) \frac{t}{z(w_i)} \tag{5.8}$$

where t has a default threshold of 10^{-3} and $z(w_i)$ is the fraction of the total words in the corpus that corresponds to this word. For example, if the "cat" word occurs 1000 times in a billion words corpus, then $z(\text{"cat"}) = 10^{-6}$. Therefore, $P(w_i) = 1$ when $z(w_i) \leq 0.0032 z(w_i) \leq 0.0032$ means to keep every instance of a word that represented 0.32 % or less and subsampled those that have a higher percentage. Please note that $P(w_i)$ does not correspond to a probability since it is no longer bounded between 0 and 1. Finally, from the accuracy point of view, subsampling frequent words can improve some accuracy and decrease one of the others.

5.1.4 Hierarchical Softmax

Hierarchical Softmax (HSM) is an efficient softmax approximation technique that uses a multilayer binary tree structure to reduce the computational cost of training a softmax neural network (Morin and Bengio, 2005). In this data structure, each node without child nodes, called *leaf nodes*, corresponds to a word and each internal node stands for relative probabilities of the children nodes.

For better accuracy, HSM structures the Word2Vec vocabulary using a Huffman tree, a binary tree data structure where data is stored in leaf nodes, without any particular order. To structure this vocabulary, HSM uses this tree so that frequent words are closer to the root node of the tree while infrequent words are deeper in this tree. Therefore, more frequent words have a greater weight than infrequent words, where the *path length* is proportional to the frequency

of a word. The path length of a tree is defined as the number of nodes that must be traversed to reach a specific word. Consequently, a HUFFMAN tree always has n leaf nodes for $n-1$ internal nodes, where each node is characterized by a weight defined by a numerical identifier. As a result, the weight of each parent node is equal to the sum of the lower weights of its children.

To compute the probability distribution of reaching a word, HSM uses a sigmoid activation function and two matrices. One for the inputs and one for the outputs differ on the values stored in each row.

Target Word	Context Words
...	..., ...
sleep	sleep, cat

(a) Input Matrix

Node	Sigmoid Value	Label
...	...	...
14	0.84	1
20	0.23	0

(b) Output Matrix

Table 5.1: HSM Matrices for the SG Model.

In Table 5.1, the input matrix of a SG model contains training samples for different target words. On the other hand, the output matrix has rows of node weights related to labels and a probability determined by the sigmoid function. This label, which can only have two values (zero or one), helps informs navigation in the tree based on a node. By convention, zero means to browse the left branch of this tree and one, the other branch. Therefore, the output matrix reports that the "cat" context word is found for the "sleep" target word by turning left at node 20 and turning right at node 14.

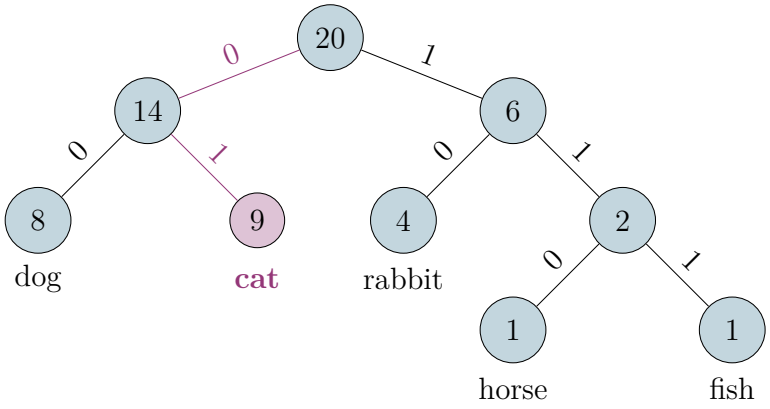


Figure 5.3: HUFFMAN Tree for Word2Vec.

In Figure 5.3, a vocabulary of five words is structured in a HUFFMAN tree containing four internal nodes, where each edge indicates a label. This training aims to teach the model to navigate in the tree by finding a context word based on a target word. For this search, the tree is traversed from top to bottom, where HSM does the dot product between the target word vector and the current node. Afterward, the result is sent to a sigmoid activation function which returns a probability value. According to this value, HSM checks that this probability is correct based on a label and updates the neurons' weights if needed. This process is repeated until the word context for a target word is found. Since training is not done on all words, there is no need to traverse every tree node. As a result, the complexity of the gradient goes from $\mathcal{O}(W)$ to $\mathcal{O}(\log_2(w))$. Once the training is completed, the SG model (the same reasoning is used for CBOW) can use this tree to predict a context word for a target word, where this time the label is not specified in the output matrix.

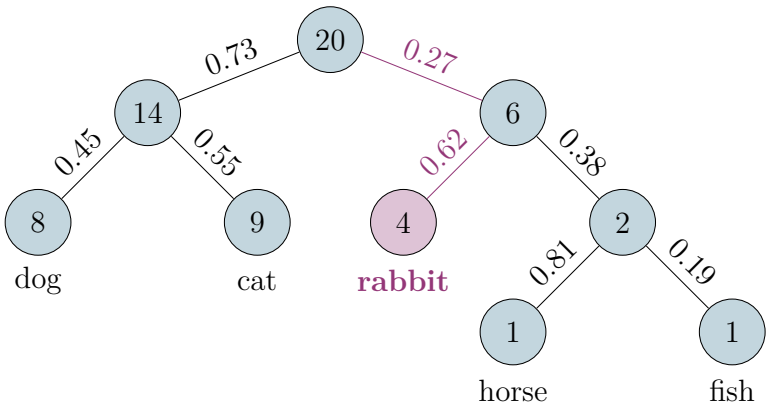


Figure 5.4: HUFFMAN Tree for Word2Vec.

In Figure 5.4, the tree edges provide the number of occurrences that a context word is to the left or right of a node, according to a given target word. After training this SG model for training data, the output matrix indicates that 27 % of the time, the context word for the same target word was in the right branch of the root node, and 73 % of the time, this context word was left branch. Based on this principle, the other nodes also return a probability distribution for their left and right branches that varies according to the input vector of a target word.

Therefore, the probability of having "rabbit" as a context word for the "sleep" target word is equal to the product of the probabilities ($\simeq 17\,\%$). Added to the probability distribution, each word in the vocabulary guarantees that its sum is unitary. Finally, through HSM, Word2Vec can retrieve word input vectors where the probability distribution of two synonymous words will have a cosine similarity close to one.

5.1.5 Negative Sampling

Negative sampling is an alternative technique to HSM, which also reduces the training speed but combined with Subsampling Frequent Words, it can improve the word embeddings' quality. To achieve this, MIKOLOV et al. assume that updating all the neurons of a ANN for each training sampling is computationally expensive. From then on, negative sampling focuses on making each training sample change only a tiny percentage of the weights rather than all of them (McCormick, 2017). However, with or without negative sampling, the hidden layer only updates the input word weights.

The way negative sampling works can be considered a simplified version of the Noise Contrastive Estimation (NCE) metric, whose purpose of NCE is to learn a data distribution by comparing it against a noise distribution. In the context of negative sampling, the latter differentiates a target word from noise samples using a logistic regression classifier (Gutmann and Hyvärinen, 2010).

The words being stored in one-hot vectors, this technique updates the weights of the "positive" and "negative" words. A word is considered positive if it was initially in the training sample, while a word is negative if it is considered noise. Specifically, this noise results from the return of the zero value in these one-hot vectors by the ANN. A small number makes selecting these negative words of random words using a "unigram distribution", where the most frequent words are more likely to be chosen as negative samples (McCormick, 2017).

Let w_i be a word from a corpus, f_{w_i} be a word frequency, and w_j be a total number of negative words in the corpus. The probability that a w_i word is selected as a negative word is defined as follows:

$$P(w_i) = \frac{f(w_i)}{\sum_{j=0}^{n} f(w_j)} \tag{5.9}$$

However, the published version increases the number of words to 3/4 power to provide better results. The following formula also increases the probability of infrequent words and decreases the likelihood of frequent words:

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=0}^{n} f(w_j)^{3/4}} \tag{5.10}$$

In practice, Word2Vec implements this negative sampling selection by filling a table called *unigram table*, including the index of each word present in a vocabulary (McCormick, 2017). Then, the selection of a negative sample is made by generating a random index, where the index of a word appears $P(w_i) * table_{\text{size}}$ times in a table. Therefore, the more frequent words are more likely to be negative words. Finally, for the choice of the number of negative words, it is recommended to take five to twenty words for small data sets instead of large data sets where it is preferable to take two to five words. (Mikolov et al., 2013).

5.1.6 Advantages and Disadvantages

In a non-exhaustive way, the advantages of the Word2Vec are:

— The use of unsupervised learning and can therefore work on any plain text.

— Requires less RAM compared to other words/vectors representations.

As for its main disadvantages, they are the following:

— No embeddings are available for Out of Vocabulary (OOV) words. Therefore, if Word2Vec has trained with the "cat" and "fish" words, it will not generate the embedding of the "catfish" compound

— No suffixes/prefixes meaning capture for given words in a corpus.

— Generation of low quality word embeddings for rare words.

— Difficulty in determining the best value for the dimensionality of the word vectors and the window size.

— The use of the softmax activation function is computationally expensive.

For the choice of the model, it is recommended to use SG when it is essential to predict infrequent words with a small amount of training data (Mikolov et al., 2013). However, CBOW is preferable for the prediction of frequent words. Added to that, SG performs significantly on semantic accuracy tests (e.g., "Athens" → "Greece" and "Oslo" → "Norway"). At the same time, CBOW offers better results in syntactic accuracy tests (e.g., "apparent" → "apparently" and "rapid" → "rapidly") (He, 2018).

5.2 FastText

FastText is an extension to Word2Vec created by Facebook in 2016, based on the decomposition of a word into character n-grams to improve the embeddings obtained by an SG model. Unlike Word2Vec, which treats each word in a corpus as an atomic entity, the decomposition made by FastText mainly solves the embeddings creation for OOV words and parameter sharing between words of the same radical. Finally, FastText can be used for unsupervised learning and supervised learning with text classification for tasks such as spam filtering.

5.2.1 Sub-Word Generation

The generation of sub-words called n-grams is an integral part of the success of FastText. Therefore, it is good to know the details of such a generation.

Definition 5.2.1 (Sub-word Generation)**.** Consists of adding angular brackets on either side of a word used as a delimiter and generating character n-grams of length n. In practice, picking this length allows extraction of n-grams ≥ 3 and ≥ 6 (Bojanowski et al., 2017). Let "flying" be a word. The following table illustrates the sub-word generation for character n-grams of length 3, 4, 5, and 6.

Table 5.2: Sub-word Generation for Character N-Grams of Length 3, 4, 5, and 6.

Word	**Length**	**Character n-grams**
flying	3	<fl, fly, lyi, yin, ing, ng>
	4	<fly, flyi, lyin, ying, ing>
	5	<flyi, lying, ying>
	6	<flyin, flying, lying>

In Table 5.2, each character n-grams of length n is generated by sliding a 2-characters window from the beginning of the bracket to its end.

As a result, the vector of a word corresponds to the sum of these n-grams of characters. This word decomposition into character n-grams comes at a storage cost since storing all the unique n-grams can quickly be significant. The original paper uses a variant of the Fowler-Noll-Vo (FNV) hash function, called *FNV-1a*, to hash character sequences into integer values to reduce this memory cost. The use of FNV is helpful for this use case since FNV was not designed for cryptography but for the fast use of hash tables and *checksums*. The checksum results of performing a cryptographic hash function on a piece of data, usually a file, to ensure that the data is authentic and error-free. Moreover, it is necessary to learn a certain amount of embeddings to hash character sequences. Specifically, this amount designates the hash bucket's size to better distribute these character n-grams for sorting and retrieval purposes. From then on, the hashing of each n-gram to a number between one and N, reduces the vocabulary size in the counterpart of potential *collisions*. This hash collision means that several character $n-grams$ stored as a key can result in the same hash and checksum. Therefore no longer ensure the authenticity of a value.

In some circumstances, the model size may be excessive. In this case, it is still possible to reduce the hash size where the appropriate value is near 20000. However, it is also possible to reduce the size of the vectors at the expense of smaller model accuracy.

5.2.2 Training

Like Word2Vec, the training of FastText's SG model can use HSM or Negative sampling. To better understand its application with negative sampling, one can take the following "I always remember her" sentence is taken. Let "remember" be the target word, using 3-grams and a window size of 3. This example predicts the "always" and "her" context words. First of all, this model computes the embeddings of the target word by summing the embeddings for the character n-grams and the whole word itself:

$$\mathrm{E}_{remember} = e_{<re} + e_{rem} + \cdots + e_{er>} + e_{remember} \tag{5.11}$$

Afterward, each context word is taken directly from their embeddings without adding the character n-grams. Then, several random negative samples are selected with a probability proportional to the square root of the unigram frequency. From then on, a context word is related to five random negative words. In the absence of n-gram embeddings, the Word2Vec model is used, specifying a maximum n-gram length of zero.

After selecting samples, the dot product between the target word and the actual context words computes the probability distribution. Unlike Word2Vec, there is this time use of the sigmoid function instead of the softmax function. Finally, the embeddings are updated with an SGD optimizer according to the calculated loss to bring the actual context words closer to the target word. Afterward, there is an incrementation in the distance of the negative samples.

This update of hyper-parameters for embeddings of n-grams adds a significant amount of extra computation to the training process. Moreover, CBOW generates the word embeddings by summing and averaging n-grams embeddings, which adds cost compared to SG. However, these additional calculations benefit from a set of embeddings containing the sub-words embeddings. From then on, they allow a better accuracy of a model in most cases.

5.2.3 Advantages and Disadvantages

In a non-exhaustive way, the advantages of the FastText:

- The use of unsupervised learning and supervised learning with text classification for tasks such as spam filtering.
- Captures the meaning of suffixes/prefixes for the words given in a corpus.
- Generation of better word embeddings for rare words as their character n-grams are shared with other more frequent words, adding more neighboring contextual words and improving its probability of being selected.
- Generation of word embeddings for OOV words.
- Better semantic similarity between two words (e.g., king and kingdom) by using extra information about the sub-words.
- No compromise of accuracy when stopwords are present.
- Significant improvement in model accuracy when used to perform syntactic word analogy tasks for morphologically rich languages (e.g., French and German).

As for its main disadvantages, they are the following:

- Requires much more RAM than Word2Vec due to the sub-words generation for each word.
- Lower accuracy compared to Word2Vec when a model is used on semantic analogy tasks.
- FastText is 50 % slower to train than the regular Skip-Gram model due to the added overhead of n-grams compared to Word2Vec.
- Difficulty in determining the best value for the n-grams generation.

5.3 BERT

Bidirectional Encoder Representations from Transformers (BERT) is a state-of-the-art NLP embedding technique published in 2018. BERT relies on the encoder part of the Transformer and whose objective is to generate an unsupervised language representation model. Unlike Word2Vec, the embeddings generated by BERT are contextualized using bidirectional representations. In the original paper, two pre-trained BERT models trained on the English Wikipedia (2,500 M words) and the Toronto BookCorpus (800 M words) are available:

1. **BERT-Base**: 12-layer Transformer, 768-hidden, 12-heads, 110 M parameters.
2. **BERT-Large**: 24-layer Transformer, 1024-hidden, 16-heads, 340 M parameters.

Since these models are pre-trained based on a corpus, their use fixes the learning vocabulary. From then on, as Word2Vec and FastText, BERT could face the presence of OOV words. As a result of these unknown words, BERT uses an adaptive tokenization algorithm.

5.3.1 Tokenization

BERT uses a WordPiece tokenization algorithm for the pre-training of a model. This tokenization proposed in 2015 by Google takes its main advantage to not replace unknown words with a `[UNK]` special token, which implies a loss of information about the input sequence. As an alternative to this conversion, WordPiece uses a segmentation method which breaks down a word into several word pieces units prefixed by `##`[1] and converted into unique identifiers. From then on, this algorithm allows BERT to learn common sub-words and does not considers OOV and rares words. As for the vocabulary ($\simeq$ 30,000 words), the latter is initializing itself by taking the individual characters of a language, and adding the most frequent combinations[2] of a corpus.

Table 5.3: Example of Tokenization With BERT.

	The	machine	loves	embeddings.					
`[CLS]`	the	machine	loves	em	##bed	##ding	##s	.	`[SEP]`
101	1996	3698	7459	7861	8270	4667	2015	1012	102

In Table 5.3, the input sequence of BERT allows receiving either one sentence (e.g., for text classification) or two sentences (e.g., for question answering) where each of them is limited to 512 characters. Beyond this limit, it is necessary to truncate the sentence. Finally, WordPiece uses three main special tokens:

1. `[CLS]`: classification token inserted at the beginning of the input used for prediction.
2. `[SEP]`: separator token used to indicates the end of each sentence;
3. `[PAD]`: padding token used when batching sequences of different lengths.

5.3.2 Input Embeddings

Unlike Transformer, BERT accepts one or two sentences as input sentences. From then on, BERT encodes these inputs to use them for the model pre-training.

[1] Except for Chinese characters, which are surrounded by spaces before any tokenization.

[2] For example, “u”, followed by “g”, would have only been merged if the probability of “ug” divided by “u”, “g” would have been greater than for any other symbol pair.

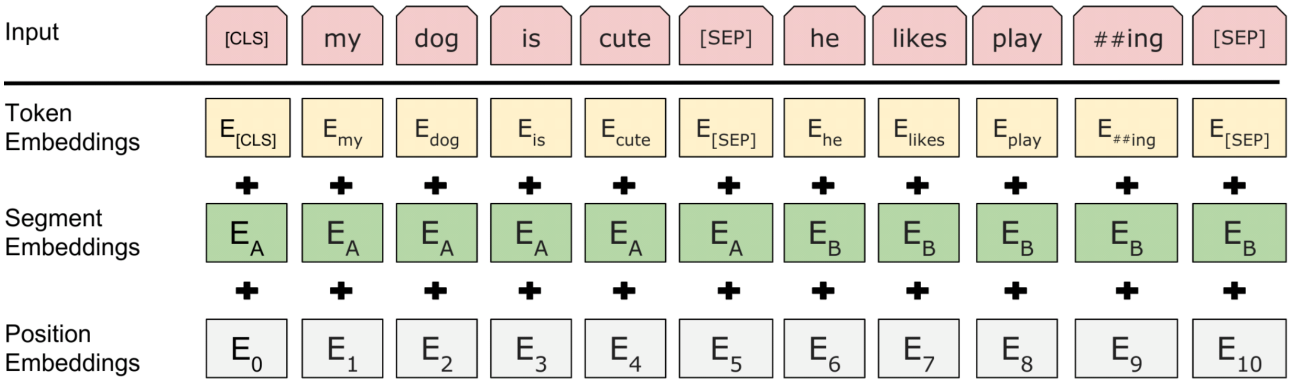


Figure 5.5: Example of Sentence Pair Encodding.

Source: Devlin et al. – BERT

In Figure 5.5, the encoding of the BERT's input requires the sum of three vector embeddings:

1. **Token embeddings**: the word pieces embeddings (cf. Section 5.3.1).
2. **Segment embeddings**: the embeddings that associate a token with its belonging sentence. In the case of a single sentence input sequence, the vector contains only unitary values. Otherwise, the `[SEP]` token and the tokens related to the first sentence are assigned a zero value. Those tokens of the second sentence are assigned a unitary value.
3. **Position embeddings**: embeddings that preserve the contextual information of the tokens by injecting their positioning into the input embeddings, as the Transformer architecture does.

5.3.3 Pre-training

The BERT model's pre-training gives it the ability to learn language by training simultaneously on two unsupervised tasks: the *Masked Language Model* (MLM) and *Next Sentence Prediction* (NSP). These two tasks mainly help overcome the lack of training data and allow the model to understand better a bidirectional representation of an input at the sentence and token level.

Masked Language Model

MLM is the first task use by the pre-training BERT model. This task helps BERT learn bidirectional contextual representations of tokens through the encoder layers by predicting masked tokens from the input sequence. For this purpose, MLM masks and predicts tokens from the input sequence. Mathematically, the following equation defines this prediction of masked tokens.

$$p\left(x_1, \ldots, x_n\right) = \sum_{i=1}^{n} p\left(x_i | x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n\right) \tag{5.12}$$

where each input sequence generally has 15 % of its tokens hidden according to the following subrules:

— a token is replaced by a `[MASK]` token in 80 % of the cases;

— a token is replaced randomly in 10 % of the cases;

— a token remains unchanged in 10 % of the cases.

These subrules prevent the Transformer encoder from being forced to maintain a distributive contextual representation of each input token. From then on, it is not recommended to modify the default token masking. Too little masking would imply a too expensive model to train, and too much masking would mean a lack of context for a token.

Next Sentence Prediction

The NSP is the second task use by the pre-training BERT model. This task helps BERT learn sentence relationships by solving a binary classification problem using Attention information shared between sentences.

Table 5.4: Example of NSP With BERT.

Sentence $\mathcal{A}$	**Sentence $\mathcal{B}$**	**Label**
GNU Emacs is the best text editor.	I should use it.	`isNextSentence`
GNU Emacs is the best text editor.	I have a cat.	`NotNextSentence`

In Table 5.4, NSP allows the model to train itself to predict whether sentence $\mathcal{B}$ is a continuation of sentence $\mathcal{A}$. For this purpose, the generation of the input sequences ensures continuity of $\mathcal{A}$ 50 % of the time, labeled as `isNext`. The rest of the time, $\mathcal{B}$ is related to a random sentence from the provided training data set, labeled as `NotNextSentence`.

Output

Once the BERT model understands the language representation, this model is suitable for most NLP tasks. These tasks include Neural Machine Translation, question answering, sentiment analysis, and text summarization.

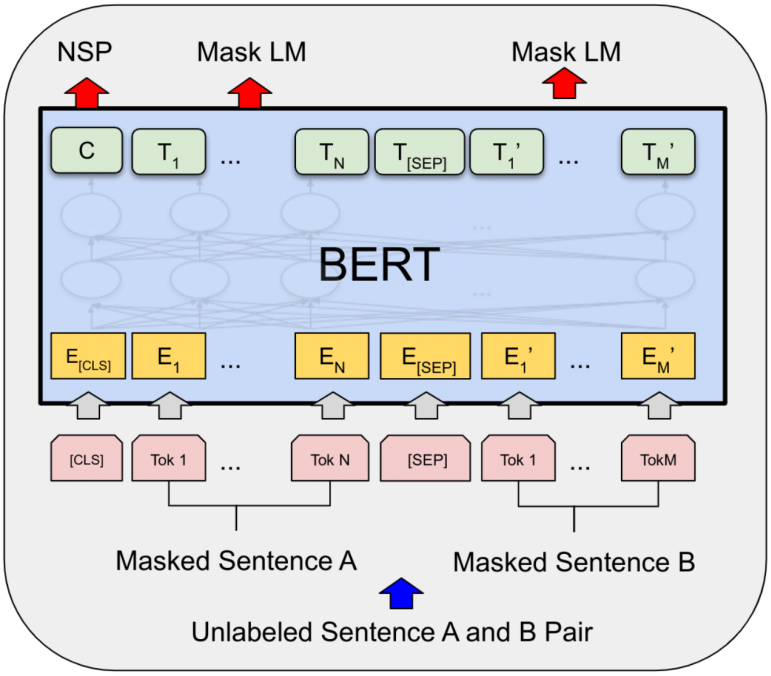


Figure 5.6: Pre-Training With BERT.

Source: Devlin et al. – BERT

In Figure 5.6, BERT produces the hidden states of each input token where each hidden state consists of a vector of the same size (e.g., 768 for BERT-Base) as the others, containing float

numbers. Among these hidden states, the first position is related to the hidden state of the token `[CLS]`. This hidden state is interesting as it determines the continuity of two sentences, which can later be used for fine-tuning tasks.

Furthermore, the hidden states pass through a last FFN containing as many neurons as tokens in the vocabulary used. The pre-training phase ends by obtaining probability distribution on the hidden states using a softmax activation function at the output of this FFN. Finally, BERT compares the distribution of the current one-hot encoded vector token with the predicted word and train the network using cross-entropy. It is important to note that the loss only considers the prediction of masked tokens produced by the network to raise awareness of the context during the network's training.

5.3.4 Fine-Tuning

The fine-tuning consists of training the BERT model on a specific NLP task. Depending on the use case, either the final hidden state of the `[CLS]` token or the hidden of the other tokens will be taken. The former is use for classification-related tasks and the latter for more complex tasks such as the Stanford Question Answering Dataset (SQuAD), Named Entity Recognition (NER), and Multi-Genre Natural Language Inference (MNLI).

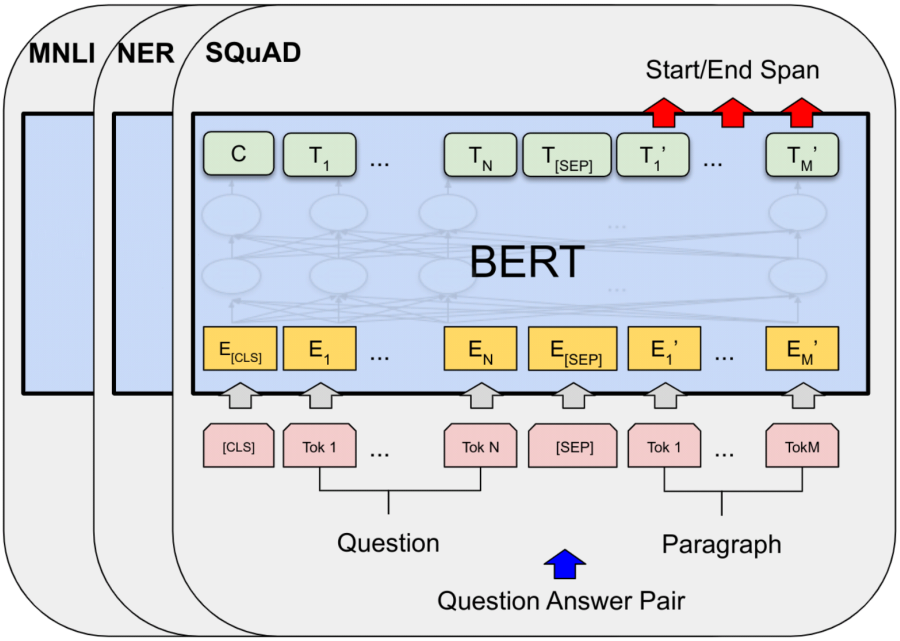


Figure 5.7: Fine-Tuning for SQuAD.

Source: Devlin et al. – BERT

In Figure 5.7, the fine-tuning of the model for SQuAD takes as input sequence a question and a paragraph containing the answer to this question. Regarding the model's output for this NLP task, BERT returns the answer to a submitted question. For this purpose, BERT highlights the starting and ending word of a given paragraph that includes the answer to that question only if that answer is in the paragraph. Therefore, it is interesting to look at the probability that a word is the start/end of the answer span.

Definition 5.3.1 (Word Probability – Start/End of the Answer Span)**.** Let $S \in \mathbb{R}^H$ be the start vector, $E \in \mathbb{R}^H$ be the end vector, $T_i \in \mathbb{R}^H$ be the final hidden vector for the i$^{\text{th}}$ input token, and j be the position of the ending word. Mathematically, the following two equations

represent the probability that the word i is the start/end of the answer span.

$$\mathrm{PS_i} = \frac{e^{ST_i}}{\sum_j e^{ST_j}} \qquad \mathrm{PE_i} = \frac{e^{ET_i}}{\sum_j e^{ET_j}} \tag{5.13}$$

where the sum of ST_i and ET_j defines the score of a candidate span from position i to position j. Afterward, the maximum scoring span where $j \geq i$ is used as a prediction.

The starting and ending tokens containing the answer are determined using the softmax activation function on the dot product between the output embeddings and the set of weights. From then on, the word with the highest probability is assigned as the start word, and the process continues to iterate to determine the end word. As most hyperparameters are similar to those of the pre-training, the only new hyperparameters added during the fine-tuning concern a classification layer. From then on, an exhaustive search of values can be done to choose the best model according to these hyperparameters.

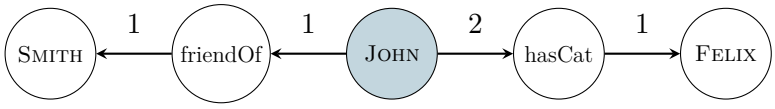
Chapter 6

RDF2Vec

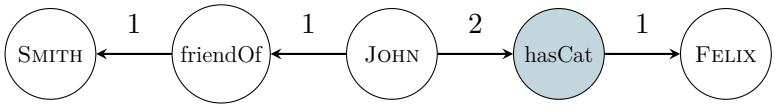
Resource Description Framework To Vector (RDF2Vec) is an unsupervised and task-agnostic algorithm to numerically represent nodes of a KG in an embedding matrix that downstream ML tasks can use. RDF2Vec is unsupervised as it only relies on the neighborhood of an entity to create embeddings and therefore does not require any information about the node labels. Precisely, the dimensions (e.g., $K \times 500$) of this embedding matrix result from the walk extraction (e.g., K) and vector size (e.g., 500).

6.1 Walk Extraction

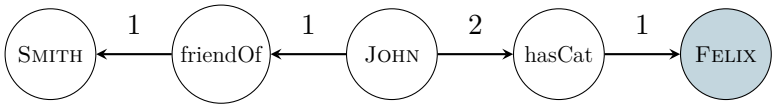
This algorithm starts using a *walking strategy* on a provided KG to extract these walks, collecting an n-tuple of nodes starting with a root node following a sequence of predicates and objects. Using the similarities shared by natural language and graphs, these walks then serve as sentences for existing NLP techniques, such as Word2Vec, to learn the embeddings of the root nodes of this KG provided by a user.



(a) Step I



(b) Step II



(c) Step III

Figure 6.1: Walk Extraction for an Oriented Graph.

In Figure 6.1, a KG is composed of five nodes and four edges. Each edge is related to a weight determined by a *sampling strategy* that can assign these weights either randomly or guided by a particular metric, called *biased walks*. These edge weights are helpful to the walking

strategy to identify the following neighboring entity to extract in a walk. The walk extraction starts with `John` as the root node, including `friendOf` and `hasCat` as possible candidates for the next hop. However, as the `hasCat` edge has a higher weight than the `friendOf` edge, the hop to `hasCat` is preferred. After this hop, the walking strategy updates its list of candidates with the neighbors of the current node. Finally, the process continues to iterates until this walking strategy returns an exhaustive list of walks or reaches a specified predefined depth covering the number of successive tuples within a walk. From then on, there is the extraction of the 3-tuple (`John`, `hasCat`, `Felix`) walk.

The original RDF2Vec implementation uses a random walking strategy to extract a limited number. The particularity of this walking strategy implies the extraction of a random hop[1] when neighboring hops have the same weight. The random walking strategy applies two algorithms to extract walks: the Depth-first search (DFS) and Breadth-first search (BFS). The former traverses a graph as deep as possible before retracing its steps. In contrast, the latter crosses every neighboring node of the same depth before crossing those of a different depth. Therefore, if there is a necessity to extract a specific number of walks, DFS is used by this strategy. Otherwise, the random walker strategy picks BFS to extract every walks of a KG.

6.1.1 Walking Strategies

After the publication of RDF2Vec, several walking strategies became available (Cochez et al., 2017), where every walking strategy can be an extraction (`type 1`) or transformation (`type 2`) technique (Vandewiele et al., 2020b). Each of these strategies brings its particularity, making it preferable to choose them in at least one use case.

As the name implies, extraction techniques focus on extracting walks, usually so that these walks provide richer information to produce a model resulting in the highest possible accuracy. This category includes random walking strategy and *Community Hops*. Based on relationships not explicitly modeled in a KG, the latter groups' nodes with similar properties through community detection. However, both walking strategies rely on the BFS and DFS algorithms, including or not some variations.

The transformation techniques categorize the walking strategies that transform the extracted walks provided by a `type 1` walking strategy. As they are easier to implement, this type includes more walking strategies than `type 1`. This technique's primary purpose is to define *one-to-many* or *many-to-one* cardinality between the old node's labels and the new ones. If there is a `one-to-one` cardinality, no additional information is gained and the original walking strategy could be used.

In a non-exhaustive way, the following walking strategies of `type 2` rely on the transformation of randomly extracted walks:

— **Anonymous Walk**: transforms each vertex name other than the root node into positional information to anonymize the randomly extracted walks.

— **Hierarchical Random Walks** (**HALK**): removes rare hops from randomly extracted walks, increasing the quality of the generated embeddings while reducing memory usage.

With this strategy, the suppression of a walk occurs when this walk only contains the root node following one or more infrequent hop(s), as it will not provide additional information.

— **N-Gram**: transforms the n-grams in random walks to define a mapping from *one-to-many*. The intuition behind this strategy is that the predecessors of a node that two different walks have in common can be different.

[1] Originally RDF2Vec preferred a random walk.

— **Walklets**: transforms randomly extracted walks into walklets which are walks of size one or two, including the root node and potentially another node that can be a predicate or an object.

— **Weisfeiler-Lehman**: transforms the nodes of the extracted random walks, providing additional information about the entity representations only when a maximum number of walks is not specified.

However, the implementations of these sampling strategies in `pyRDF2Vec` relied on extracting child nodes, not parent nodes, which lost context for a root node.

6.1.2 Sampling Strategies

The sampling strategies essentially allow to better deal with larger KGs. A naive implementation randomly samples a fixed number of walks for each entity to keep the total number of walks limited. Since then, the community (Cochez et al., 2017; Mukherjee et al., 2019; Taweel and Paulheim, 2020) has suggested several metrics to compute the sampling weights while walking.

Although sampling strategies allow the extraction of large KGs, most of these strategies require working on the entire KG to assign weights to edges. However, some KGs are so large that they need to be stored in a *triplestore*[2] and made available through a SPARQL endpoint. Therefore, one way to use these sampling strategies is to load parts of large KGs, assign weights, and start the walk extraction. Finally, the different walks extracted for these parts of KGs would be concatenated and returned.

6.2 Shortcomings

The node representation made by RDF2Vec has already achieved great predictive performances on several data sets in various fields. However, RDF2Vec still has a few shortcomings. In a non-exhaustive way:

— **RDF2Vec does not scale to large KGs**: the walk extraction grows exponentially with the predefined depth. This behavior is unacceptable with KGs containing many nodes, mainly when these KGs contain many highly connected nodes.

— **RDF2Vec cannot deal with literals in the KG**: this algorithm extracts node as *non-ordinal categorical* data, which discard a considerable amount of rich information that *literals* can provide. In other words, nodes can be from different types, but RDF2Vec extracts these nodes as a name without any classification instead of conserving their types.

— **RDF2Vec cannot deal with dynamic graphs**: adding a new entity in a KG implies re-training the model generated by the embedding technique. This re-training is undesirable, especially when the training time of a model is substantial.

— **RDF2Vec uses a simple data structure for storing walks**: Extracting more complex data structures, such as trees, or modifying the walking algorithm to introduce different inductive *biases* could result in higher quality embeddings. Consequently, these quality embeddings would improve the model's accuracy.

[2]Database designed for the storage and retrieval of RDF.

— **RDF2Vec uses an embedding technique that is no longer state of the art in NLP**: currently, RDF2Vec uses Word2Vec as an embedding technique. However, more recent NLP techniques such as BERT could be a better alternative to Word2Vec and improve the model's accuracy.

The community has proposed solutions to address the shortcomings mentioned above to improve this representation of nodes. Among these, optimization mechanisms (e.g., caching and multiprocessing) to better handle large KGs and an *online learning* implementation to update the vocabulary of nodes learned by RDF2Vec. In addition, a user could extract interesting literals by specifying a sequence of predicates followed by a walking strategy. Finally, other embedding techniques than Word2Vec were also proposed, such as *KGloVe*[3], which uses the Global Vectors for Word Representation (GloVe) embedding technique.

[3] https://datalab.rwth-aachen.de/embedding/KGloVe/

Chapter 7

Work Performed

This chapter is dedicated to the solutions provided by this Master's thesis. These solutions include:

— **Improving the accuracy of the Word2Vec model**: using a user-defined depth to extract the child and parent nodes of a current node for each walk. In addition, the centralization of the root node in the extracted walks maximizes the number of training samples containing this root node and therefore generates better quality embeddings.

— **A BERT implementation**: builds its vocabulary only based on special tokens and single extracted nodes, not allowing WordPiece splitting in the tokenization of nodes. Finally, this version of BERT only considers the MLM as a pre-training step, with training that tends to find a trade-off between its time and the accuracy that this model will generate.

— **A FastText implementation**: unlike its original version, its implementation does not allow splitting n-grams on a defined minimum and maximum length, but only according to a splitting function provided by a user. In addition, the reimplementation of the Cython n-grams computation function in Python to facilitate the use of `pyRDF2Vec` on Google Colaboratory[1].

— **WideSampler**: sampling strategy that maximizes the extraction of shared features between entities.

— **SplitWalker**: walking strategy that extracts walks according to a splitting function.

— **A better architecture**: the `pyRDF2Vec` library now allows to easily add walking strategies, sampling strategies, and embedding techniques by reimplementing only a few functions.

Finally, this Master's thesis implicitly proposes some research for future work related to BERT. Among these, the injection of `(subject, object)` 2-tuples in BERT instead of two walks. Such an injection would allow BERT to focus to predict predicates instead of seeing correlations between two different walks.

7.1 Improving the Accuracy of the Word2Vec Model

To better understand how it is possible to improve the model's accuracy generated by Word2Vec, it is helpful to consider the initial problem with the walk extraction. With RDF2Vec, the transformation for a n-tuple without any walks limitation and concerning `"URL#Alice"` as the root node can be achieved with each walking strategy as follows:

[1]Website that allows to run code on Google's cloud servers.

Walking Strategy	Initial/Transformed n-tuple
Anonymous Walk	`("URL#Alice", "URL#knows","URL#Bob")` `("URL#Alice", "1", "2")`
HALK	`("URL#Alice", "URL#knows", "URL#Bob")`, `("URL#Alice", "URL#loves", "URL#Bob")`, ... `("URL#Alice", "URL#knows","URL#Bob")`[1]
N-Gram	`("URL#Alice", "URL#knows", "URL#Bob")`, `("URL#Alice", "URL#loves", "URL#Dean")` `("URL#Alice", "URL#knows", "0")`, `("URL#Alice", "URL#loves", "1")`[2]
Walklets	`("URL#Alice", "URL#knows","URL#Bob")` `("URL#Alice", "URL#knows")`, `("URL#Alice", "URL#Bob")`
Weisfeiler-Lehman	`("URL#Alice", "URL#knows","URL#Bob")` `("URL#Alice", "URL#knows", "URL#Bob")`, `("URL#Alice", "URL#knows", "URL#Bob-URL#knows")`[3]

[1] Assuming a minor threshold frequency and an infrequent `"URL#loves"` hop compared to other hops.
[2] Assuming at least one gram to relabel. If grams > 2, every object names will remain identical for this example.
[3] Assuming a Weisfeiler Lehman iteration of 1.

Table 7.1: Example of n-tuple Transformation for Type 2 Walking Strategies.

The walking strategies in Table 7.1 already give good results. However, the `"URL#Alice"` root node is always positioned at the beginning of each walk, reducing the richness of these walks for embedding techniques like Word2Vec, which works with window size. As the solicitation of nodes placed in the middle of a walk is higher due to selecting context words, including the root node as close as possible to the middle for each walk could generate better quality embeddings. Indeed, much more training sampling would contain this root node.

In addition to this positioning issue of the root node, `type 1` walking strategies have the main disadvantage of continuously extracting child nodes of a root node. In other words, the embedding techniques never know the root node's parents. An alternative would be to extract the whole child and parent nodes of a root node. Such extraction maximizes the extracted walk information and, therefore, improves a model's accuracy after being trained with an embedding technique. However, this alternative is not suitable when extracted walks are limited, mainly used to process large KGs. Furthermore, when it comes to positioning, the extraction of parent nodes from a root node also implies a wrong positioning. Specifically, this root node would be this time placing as the last node of walks, preceded by a sequence of predicates and objects, which is not desirable.

Based on these constraints, this Master's thesis proposes to apply a user-defined depth to extract the child and parent nodes of a current node for each walk. Assuming that the root node has parent nodes, each extracted walk will have a better position for this root node. Indeed, each root node will be preceded by a part of its parent nodes and succeeded by a part of its child nodes.

In Table 4.1, assuming the sentence "I will always remember her" and considering the word "I" as the root node of a walk, the latter is only included twice in the context words. However, suppose now this word is positioned in the center instead of the word "always". In that case, its frequency of occurrence can be doubled, i.e., from two to four times for this example. As a result, the embeddings generated for the different root nodes are of better quality by the number of occurrence of root nodes and by the context provided.

7.2 BERT Implementation

The recommended library to implement BERT is `huggingface/transformers`[2] which provides many pre-trained models[3] and essential functions to create a model. As no pre-training model exists for BERT with KGs, it is necessary to create this model from scratch, which requires three main steps:

1. **Build the vocabulary**: based on the nodes, including one line per special token and one line per unique node, as part of a KG.
2. **Fits the BERT model**: based on the corpus of walks provided, including three main goals:
   (a) **Node tokenization**: in such a way that special tokens are inserted on both sides of the nodes, taking care not to split them. Which unlike words, splitting a URI is not desired.
   (b) **Pre-training**: only done with MLM, as the NSP pre-training task is not helpful since the walks do not share any continuity.
   (c) **Training**: is done by providing a training set of formatted walks, a data collator (e.g., `DataCollatorForLanguageModeling`), and the training parameters. The walk formatting is necessary to ensure added padding, truncating walks that are too long ($\geq$ 512 characters).
3. **Transforms the provided entities into embeddings**: returns entity embeddings.

Listing 7.1: Creation of the Walk Data Set.

```
class WalkDataset(Dataset):
  def __init__(self, corpus, tokenizer):
    self.walks = [
      tokenizer(
        " ".join(walk), padding=True, truncation=True, max_length=512
      )
      for walk in corpus
    ]

  def __len__(self):
    return len(self.walks)

  def __getitem__(self, i):
    return self.walks[i]
```

In Algorithm 7.1, creating the walks data set for BERT is done by creating a dedicated class. Within this class, tokenization is necessary. Each walk is truncated to 512 characters, followed by padding to handle walks of the same size. Finally, it is also useful to implement the Dunder[4] `__len__` and `__getitem__` methods to define the size of a sample of training data and for the fetching of this sample.

Listing 7.2: Creation of Node Vocabulary.

```
def _build_vocabulary(self, nodes, is_update = False):
  with open(self.vocab_filename, "w") as f:
    if not is_update:
      for token in ["[PAD]", "[UNK]", "[CLS]", "[SEP]", "[MASK]"]:
        f.write(f"{token}\n")
        self._vocabulary_size += 1
```

[2] https://huggingface.co/transformers/
[3] https://huggingface.co/models
[4] Also called *magic* methods.

```
    for node in nodes:
      f.write(f"{node}\n")
      self._vocabulary_size += 1
```

In Algorithm 7.2, each walk node is stored in a file with special tokens at the beginning of the file, namely: `[PAD]`, `[UNK]`, `[CLS]`, `[SEP]`, and `[MASK]`. Finally, a boolean is also sent to update or not an already existing vocabulary and avoid the complete re-training of the model.

Listing 7.3: Fitting the BERT Model According to the Provided Walks.

```
def fit(self, walks, is_update = False):
  walks = [walk for entity_walks in walks for walk in entity_walks]
  nodes = list({node for walk in walks for node in walk})
  self._build_vocabulary(nodes, is_update)
  self.tokenizer = BertTokenizer(
    vocab_file=self.vocab_filename,
    do_lower_case=False,
    never_split=nodes,
  )
  self.model_ = BertForMaskedLM(
    BertConfig(
      vocab_size=self._vocabulary_size,
      max_position_embeddings=512,
      type_vocab_size=1,
      )
   )
   Trainer(
     model=self.model_,
     args=self.training_args,
     data_collator=DataCollatorForLanguageModeling(
       tokenizer=self.tokenizer
     ),
     train_dataset=WalkDataset(walks, self.tokenizer),
   ).train()
```

In Algorithm 7.3, the training of the BERT model consists of retrieving each unique node from the provided walks, tokenizing these walks, choosing a suitable hyperparameter config, and running the training based on a set of walks data.

BERT needs to be trained on a large enough corpus of walks to provide good results. Having such a corpus is not always possible, depending on the size of some KGs. In addition, hyperparameters for training BERT impact the model's accuracy and training time. This training time can take hours, days, weeks, if not more. Therefore it is necessary to improve as much as possible the quantity and quality of the walks corpus with RDF2Vec to reduce this training time.

Listing 7.4: Getting the Entities' Embeddings with BERT.

```
def transform(self, entities):
  check_is_fitted(self, ["model_"])
  return [
    self.model_.bert.embeddings.word_embeddings.weight[
      self.tokenizer(entity)["input_ids"][1]
    ]
    .cpu()
    .detach()
    .numpy()
    for entity in entities
  ]
```

In Algorithm 7.4, the retrieving of embeddings for the requested entities, i.e., the root notes, consists of recovering the generated embeddings and ensuring their good formatting.

These methods are sufficient for a classical implementation of BERT. All the subtlety of generating a correct model is characterized by the extraction and injection of walks and the values of chosen hyperparameters.

7.3 FastText Implementation

To implement FastText, the `gensim`[5] library is recommended to be used. However, `pyRDF2Vec` had to reimplement much of the code in order:

— **To remove the `min_n` and `max_n` parameters for n-grams splitting**: the `object` nodes in `pyRDF2Vec` are encoded in MD5 to reduce their storage in RAM. Therefore, splitting them into n-grams is pointless.

— **To allow a user to compute n-grams for walks only by separating** (default split by their symbols) **the URIs of subjects and predicates**: a user will likely want to provide an alternative splitting strategy for computing entity n-grams on the KG. If this is the case, `pyRDF2Vec` allows a user to implement this function that FastText will use.

— **To avoid dependency on Cython**: Cython is a programming language between Python and C. Its main interest is to obtain performances of calculation time similar to C for specific functions in Python by reimplementing them in Cython. The gensim library uses Cython for the calculation of n-grams hashes. However, `pyRDF2Vec` has chosen to reimplement this function in Python to facilitate its use on Google Colaboratory.

Listing 7.5: Reimplementation of the Hash Calculation Functions in `gensim`.

```
  def compute_ngrams_bytes(entity):
    if "http" in entity:
      ngrams = " ".join(re.split("[#]", entity)).split()
      return [str.encode(ngram) for ngram in ngrams]
    return [str.encode(entity)]

    def ft_hash_bytes(self, bytez: bytes) -> int:
      h = 2166136261
      for b in bytez:
        h = h ^ b
        h = h * 16777619
      return h

    def ft_ngram_hashes(entity, num_buckets = 2000000):
      encoded_ngrams = func_computing_ngrams(entity)
      hashes = [ft_hash_bytes(n) % num_buckets for n in encoded_ngrams]
      return hashes

    def recalc_char_ngram_buckets(bucket, buckets_word, index_to_key) -> None:
      if bucket == 0:
        buckets_word = [np.array([], dtype=np.uint32)] * len(index_to_key)
        return
      self.buckets_word = [None] * len(index_to_key)
      for i, word in enumerate(index_to_key):
        buckets_word[i] = np.array(ft_ngram_hashes(word, 0, 0, self.bucket),
            dtype=np.uint32)
```

In Algorithm 7.5, the functions present in `gensim` are reimplemented in such a way to include the splitting function of `pyRDF2Vec`. Added to that this reimplementation does not consider the use of Cython anymore.

Listing 7.6: Fitting the FastText Model According to the Provided Walks.

```
def fit(self, walks, is_update = False):
  corpus = [walk for entity_walks in walks for walk in entity_walks]
  self._model.build_vocab(corpus, update=is_update)
  self._model.train(
    corpus,
    total_examples=self._model.corpus_count,
    epochs=self._model.epochs,
  )
  return self
```

[5] https://github.com/RaRe-Technologies/gensim

In Algorithm 7.6, training with FastText consists of extracting each node from each walk, building the vocabulary, and training the model based on the corpus of walks.

Listing 7.7: Getting the Entity Embeddings with FastText.

```
def transform(self, entities):
  if not all([entity in self._model.wv for entity in entities]):
    raise ValueError(
      "The entities must have been provided to fit() first "
      "before they can be transformed into a numerical vector."
    )
  return [self._model.wv.get_vector(entity) for entity in entities]
```

In Algorithm 7.7, even though FastText can generate entity embeddings that it has not learned, `pyRDF2Vec` throws an exception instead of avoiding any unpleasant surprises from the model's accuracy.

7.4 SplitWalker

Based on the idea of FastText, but directly applied to the extraction of walks, this Master's thesis proposes `SplitWalker` as a new `type 2`. Specifically, this strategy splits the vertices of the random walks for a based entity. To achieve this, each vertex, except the root node, is split according to symbols, capitalization, and numbers by removing any duplication.

	Initial Node	**Node After Splitting**
Walk 1	http://dl-learner.org/carcinogenesis#d19	http://dl-learner.org/carcinogenesis#d19
	http://dl-learner.org/carcinogenesis#hasBond	has
		bond
	http://dl-learner.org/carcinogenesis#bond3209	3209
Walk 2	http://dl-learner.org/carcinogenesis#d19	http://dl-learner.org/carcinogenesis#d19
	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	type
	http://dl-learner.org/carcinogenesis#Compound	compound

Table 7.2: Example of Use of `SplitWalker`.

In Table 7.2, two walks are transformed by `SplitWalker`. Both keep their root node intact. However, the first walk has the `.../hasBond` split by the letter `B` into two nodes: `has` and `bond`. In addition, its third node `.../bond3209` is also split into two nodes: `bond` and `3209`, but since `bond` is an existing node in the walk, it is removed from it. Finally, the second walk has the particularity to have a node with a capital letter. In this case, this node is rewritten in lowercase.

Listing 7.8: Splits Nodes of Random Walks with `SplitWalker`.

```
def basic_split(self, walks):
  canonical_walks = set()
  for walk in walks:
    canonical_walk = [walk[0].name]
    for i, _ in enumerate(walk[1::], 1):
      vertices = []
      if "http" in walk[i].name:
        vertices = " ".join(re.split("[\#]", walk[i].name)).split()
      if i % 2 == 1:
        name = vertices[1] if vertices else walk[i].name
        preds = [
          sub_name
          for sub_name in re.split(r"([A-Z][a-z]*)", name)
          if sub_name
        ]
```

```
        for pred in preds:
          canonical_walk += [pred.lower()]
      else:
        name = vertices[-1] if vertices else walk[i].name
        objs = []
        try:
          objs = [str(float(name))]
        except ValueError:
          objs = re.sub("[^A-Za-z0-9]+", " ", name).split()
          if len(objs) == 1:
            match = re.match(
              r"([a-z]+)([0-9]+)", objs[0], re.I
            )
            if match:
              objs = list(match.groups())
        for obj in objs:
          canonical_walk += [obj.lower()]
    canonical_walk = list(dict(zip(canonical_walk, canonical_walk)))
    canonical_walks.add(tuple(canonical_walk))
  return canonical_walks
```

Algorithm 7.8 starts by traversing each provided walk, making sure to save the root node characterized by the first vertex of the walk. Then, this function looks to see if that node has the prefix "http" for each node. If it does, then that node is split by the `#` symbol. Otherwise, if the current node is a predicate, this function does an uppercase split to create other nodes. Finally, this process is also done in the case where the node contains numbers. In the end, this function deletes all the duplicated nodes and returns the walks.

7.5 WideSampler

Based on the principle that humans tend to classify objects according to common features (e.g., color and shape), this Master's thesis proposes a new sampling strategy called `WideSampler`. This strategy addresses the assumption that single entity-specific features would have a negligible impact on the quality of the generated embeddings. `WideSampler` assigns higher weights to edges that lead to the most significant number of predicates and objects in the neighborhood and terms of occurrence in a graph to extract a maximum of shared features between entities.

After training the model, this walking strategy can retrieve the weight of a neighboring hop as shown below.

Algorithm 1 `get_weight(h, d, c)`

Require: a $\mathcal{H}$ 2-tuple that contains a predicate and an object.
Require: a $\mathcal{D}$ array of $n \geq 1$ degree indexed from 0 to $n-1$
Require: a $\mathcal{C}$ array of $n \geq 1$ counter of neighbors indexed from 0 to $n-1$
Ensure: The weight of the hop for this predicate

1: **if** $\mathcal{D}_{preds}$ and $\mathcal{D}_{objs}$ and $\mathcal{C}$ **then**
2: **return** $(\mathcal{C}[\mathcal{H}[0]_{name}] + \mathcal{C}[\mathcal{H}[1]_{name}]) \left(\frac{\mathcal{D}_{preds}[\mathcal{H}[0]_{name}] + \mathcal{D}_{objs}[\mathcal{H}[1]_{name}]}{2} \right)$
3: **end if**

Algorithm 1 assigns a weight to a `(predicate, object)` hop according to the multiplication of two sums. The first one considers the child nodes of a predicate and the parent nodes of an object node. The second one considers the number of occurrences of this predicate and this node in the whole KG. Finally, the second sum is divided by half to provide a slight preference to a hop that reaches multiples nodes.

Algorithm 2 `fit(v)`

Require: an $\mathcal{V}$ array of $n \geq 1$ edges indexed from 0 to $n-1$
Ensure: Fits the WideSampler sampling strategy

$\mathcal{C}_{objs} \leftarrow$ new array of n objects.
$\mathcal{C}_{preds} \leftarrow$ new array of n predicates.
for all $vertex \in \mathcal{V}$ **do**
  if $vertex$ is predicate **then**
    $\mathcal{C}_{neighbors}[vertex_{name}] = |\texttt{get\_children(vertex)}|$
    $\mathcal{C}_{tmp} \leftarrow \mathcal{C}_{preds}$
  else
    $\mathcal{C}_{neighbors}[vertex_{name}] = |\texttt{get\_parents(vertex)}|$
    $\mathcal{C}_{tmp} \leftarrow \mathcal{C}_{objs}$
  end if
  if $vertex_{name}$ in $\mathcal{C}_{tmp}$ **then**
    $\mathcal{C}_{tmp}[vertex_{name}] \leftarrow \mathcal{C}[vertex_{name}] + 1$
  else
    $\mathcal{C}_{tmp}[vertex_{name}] \leftarrow 1$
  end if
end for

Algorithm 2 trains the `WideSampler` strategy by iterating through a set of vertices. Depending on the vertex, the algorithm considers the number of children or parents of this vertex. If it is a predicate, then the number of its child nodes is stored in a counter. Otherwise, the algorithm stores the number of parent nodes in another counter. Finally, the number of occurrences of each identical vertex in the whole KG is also collected.

7.6 Library Architecture

`pyRDF2Vec`[6] is the Python library created by IDLab that allows to use RDF2Vec. Through the internship and this Master's thesis, this library has undergone many changes in its architecture to improve its use.

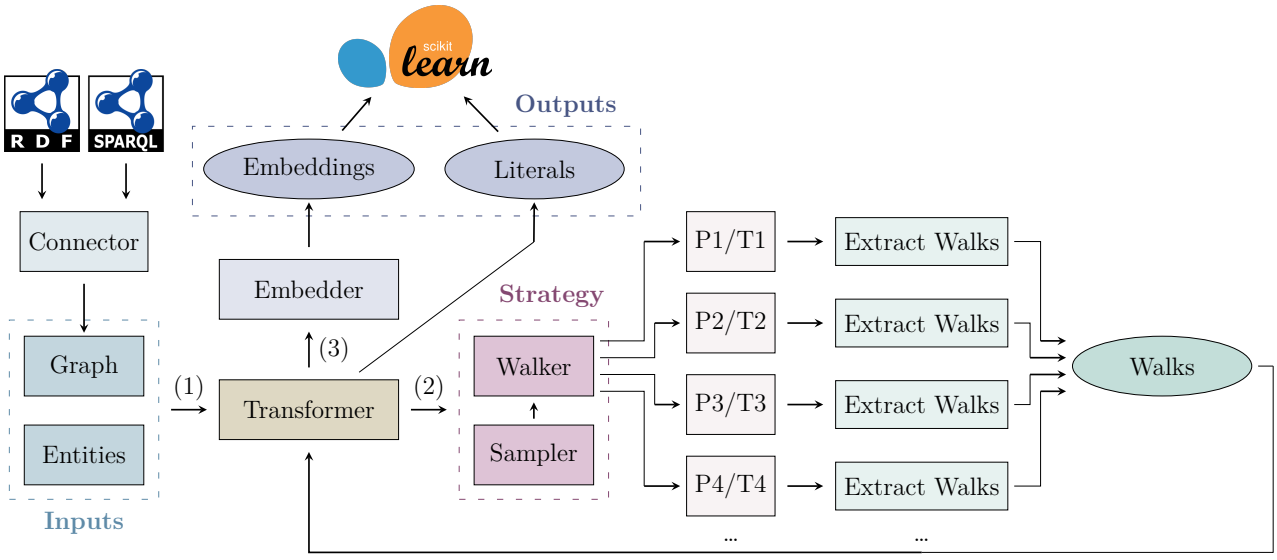


Figure 7.1: Workflow of `pyRDF2Vec`.

[6] https://github.com/IBCNServices/pyRDF2Vec

In Figure 7.1, the workflow of `pyRDF2Vec` includes three primary operations and is divided into seven main consecutive significant blocks:

1. **Connector**: in charge of interaction with a local or remote graph.
2. **Graph**: in charge of providing a graph encoding the knowledge-based.
3. **Entities**: in charge of provides the entities in a `rdflib.URI.term` or `str` type to generate the embeddings.
4. **Transformer**: in charge of converting graphs into embeddings for downstream ML tasks, using a walking strategy and sampling strategy and an embedder.
5. **Sampler**: in charge of prioritizing the use of some paths over others using a weight allocation strategy.
6. **Walker**: in charge of extracting walks in a KG from provided entities and optionally from a sampling strategy using multiple processors/threads.
7. **Embedder**: in charge of training a model with an embedding technique using extracted walks and therefore generate embeddings of entities provided by a user.

The design of such architecture allows having a long-term vision. Due to this architecture, a user can contribute to `pyRDF2Vec` and easily add new walking strategies and sampling and embedding techniques. For this Master's thesis, implementing this architecture was necessary to facilitate the comparison of embedding techniques. Each new embedding technique is added in an `Embedder` package and must reimplement the `fit` and `get_weight` functions of the `Embedder` class.

Chapter 8

Benchmarks

For these benchmarks, only the maximum number of walks per entity and the maximum depth per walk is varied. Furthermore, due to a time issue (cf. Section 3.2) these benchmarks are performed on `MUTAG`, a graph of moderate size composed of 74,567 triples[1] 22,534 entities[2], and 24 relations[3]. Finally, each value entered in these benchmarks is the result of the average of five values.

8.1 Setup

Benchmarks related to embeddings techniques and walking strategies are directly launched on IDLab's[4] servers with 4 CPUs, 64 GB RAM, and one GPU. Those related to sampling strategies are launched directly on a ThinkPad machine with 4 CPUs and 16 GB of RAM. This physical device change is made since, except for `UniformSampler`, the sampling strategies only work on locally stored KGs. As the IDLab servers interact with the KGs via SPARQL endpoints, they were not used for these benchmarks. Finally, the benchmarks use 340 training entities and attempt to predict 68 test entities, which is a standard for `MUTAG`.

8.2 Results

This section contains the results of the different embedding techniques, walking strategies, and sampling strategies for `MUTAG`.

8.2.1 Embedding Techniques

FastText, BERT and Word2Vec are trained on the basis of ten epochs. In addition, FastText and Word2Vec use twenty negative words with a vector size of 500. For its splitting function, FastText uses a primary splitting function where the `#` symbol splits each entity Finally, each embedding technique uses a `RandomWalker` and an `UniformSampler`.

About the training of BERT, the latter is only trains on `MUTAG`. However, in case of multiple KGs, it is important to re-train it for each different KGs. Similarly, if BERT is trained with too few walks, it will be necessary to retrain it with a larger number of walks. In this case, online learning is important to avoid to retrain the whole model which can be time-consuming. Unline BERT that can take hours, days for training the model, Word2Vec and FastText take a few minutes to tens of minutes to train, which is significant difference.

[1] **SELECT** (COUNT(*) AS ?triples) **WHERE** { ?s ?p ?o }
[2] **SELECT** (COUNT(**DISTINCT** ?s) AS ?entities) **WHERE** { ?s a }
[3] **SELECT** (COUNT(**DISTINCT** ?p) AS ?relations) **WHERE** { ?s ?p ?o }
[4] Research group of imec.

Hyperparameter	Value
Epochs	10
Warmup Steps	500
Weight Decay	0.01
Learning Rate	2e-5
Batch Size	16

Table 8.1: Basic Hyperparameters Used for Training the BERT Model.

In Table 8.1, The values of these basic hyperparameters were chosen after several tests using a Grid Search with Cross Validation and depending on the training time. More precisely, a training of the BERT model with these hyperparameters with MUTAG and the same hardware characteristics as those of the IDLab servers, is done between 25 minutes and few hours.

Embedding Technique	Max. Depth	Max. Walks	Accuracy (%)	Rank
`FastText(negative=20,vector_size=500)`	2	250	79.71 ± 2.35	1
`Word2Vec(negative=20,vector_size=500)`			76.76 ± 1.71	2
`BERT(learning_rate=2e-5,batch_size=16)`			70.59 ± 5.88	3
`FastText(negative=20,vector_size=500)`	4	250	77.06 ± 1.50	1
`Word2Vec(negative=20,vector_size=500)`			75.00 ± 1.61	2
`BERT(learning_rate=2e-5,batch_size=16)`			74.26 ± 2.21	3
`FastText(negative=20,vector_size=500)`	6	250	82.35 ± 1.86	1
`BERT(learning_rate=2e-5,batch_size=16)`			76.32 ± 3.24	2
`Word2Vec(negative=20,vector_size=500)`			74.71 ± 2.35	3

Table 8.2: Evaluation of the Embedding Techniques for `MUTAG` According to the Maximum Depth per Walk.

In Table 8.2, regardless of the maximum depth per walk chosen for the same number of walks per entity, FastText indicates a model's accuracy above Word2Vec. Specifically, FastText allows an average increase of the model's accuracy of 4.22 times the one given by Word2Vec. In addition, FastText provides an excellent model's accuracy with `MUTAG` for a maximum depth per walk of 6. For BERT, the latter shows better results for larger maximum depth per walk.

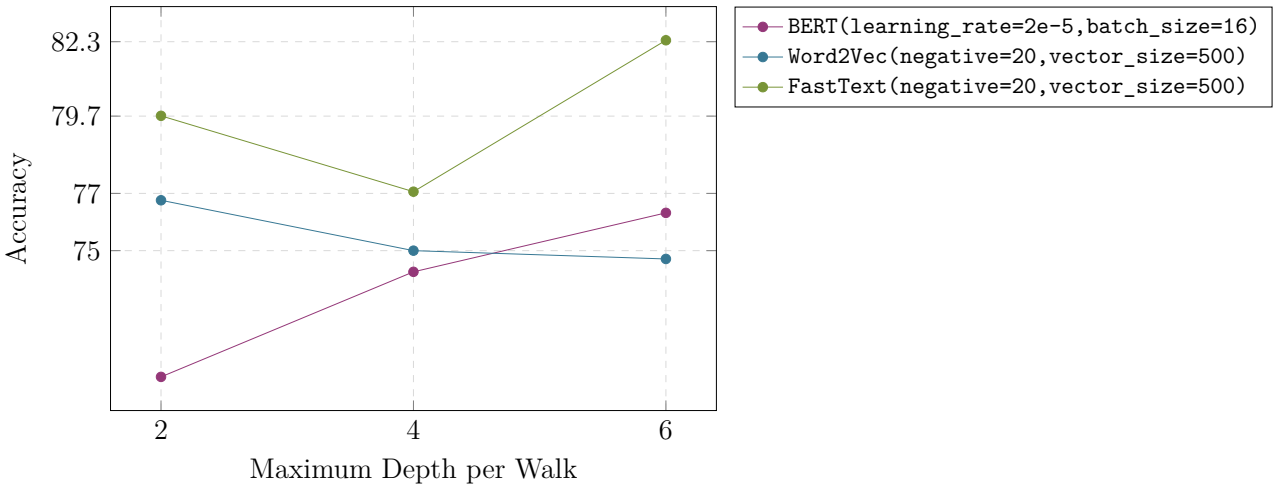


Figure 8.1: Evaluation of the Embedding Techniques for `MUTAG` According to the Maximum Depth per Walk.

In Figure 8.1, the curves of Word2Vec and FastText have an almost identical trajectory, except for a maximum depth per walk of 6. In this case, the accuracy model of FastText increases, while the accuracy model of Word2Vec decreases. Finally, BERT's accuracy is proportional to the maximum depth per walk. As well as the time needed to train the model.

Embedding Technique	Max. Depth	Max. Walks	Accuracy (%)	Rank
`FastText(negative=20,vector_size=500)`			77.94 ± 1.61	1
`Word2Vec(negative=20,vector_size=500)`	4	100	71.47 ± 2.20	2
`BERT(learning_rate=2e-5,batch_size=16)`			69.43 ± 1.73	3
`FastText(negative=20,vector_size=500)`			77.35 ± 3.90	1
`Word2Vec(negative=20,vector_size=500)`	4	250	74.71 ± 2.53	2
`BERT(learning_rate=2e-5,batch_size=16)`			73.54 ± 2.28	3
`FastText(negative=20,vector_size=500)`			76.18 ± 1.71	1
`BERT(learning_rate=2e-5,batch_size=16)`	4	500	75.24 ± 2.37	2
`Word2Vec(negative=20,vector_size=500)`			73.53 ± 1.86	3
`FastText(negative=20,vector_size=500)`			77.35 ± 2.73	1
`BERT(learning_rate=2e-5,batch_size=16)`	4	1000	76.58 ± 1.17	2
`Word2Vec(negative=20,vector_size=500)`			74.41 ± 3.03	3

Table 8.3: Evaluation of the Embedding Techniques for `MUTAG` According to the Maximum Number of Walks per Entity

In Table 8.3, regardless of the number of walks chosen for the same maximum depth per walk, FastText indicates a model's accuracy above Word2Vec. Specifically, FastText allows an average increase of the model's accuracy of 3.675 times the one given by Word2Vec. For BERT, the latter shows better results for larger maximum depth per walk, but performs less well for smaller maximum depth per walk. For BERT, the latter indicates an interesting model's accuracy for 500 and 1000 walks. However, the results are not as exceptional for a lower maximum of walks per entity.

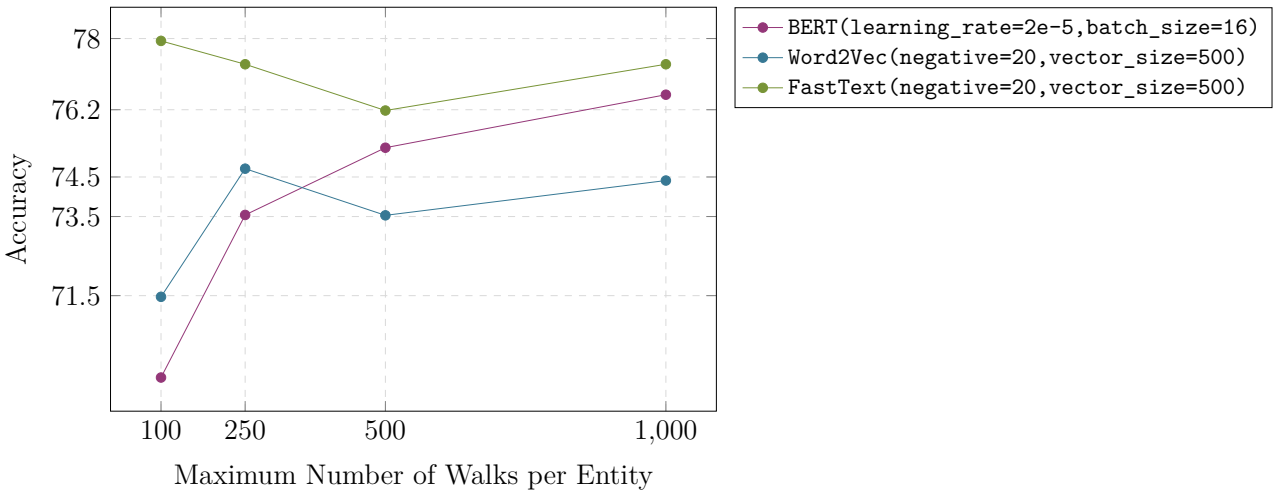


Figure 8.2: Evaluation of the Embedding Techniques for `MUTAG` According to the Maximum Number of Walks per Entity.

In Figure 8.2, the curves of Word2Vec and FastText still have an almost identical trajectory, except for a maximum number of walks per entity of 250. In this case, the accuracy model of Word2Vec increases, while the accuracy model of FastText decreases. Finally, BERT's accuracy is also proportional to the maximum number of walks per entity. As well as the time needed to train the model.

Embedding Technique	Average Rank
`FastText(negative=20,vector_size=500)`	1
`Word2Vec(negative=20,vector_size=500)`	2
`BERT(learning_rate=2e-5,batch_size=16)`	3

Table 8.4: Evaluation of the Average Rank of the Embedding Techniques for `MUTAG`.

In Figure 8.4, Word2Vec is the winning embedding techniques in these benchmarks for `MUTAG`, followed by FastText, and BERT.

8.2.2 Walking Strategies

Each walking strategy is evaluated using `UniformSampler` as sampling strategy and Word2Vec as embedding technique. For these benchmarks, Word2Vec keeps the same hyperparameters as given in Section 8.2.1, namely ten epochs, twenty negative words and a vector size of 500.

Walker	Max. Depth	Max. Walks	Accuracy (%)	Rank
`RandomWalker`	2	250	**77.94** ± 2.08	1
`NGramWalker(grams=3)`			76.47 ± 1.32	2
`HALKWalker(freq_threshold=0.01)`			75.59 ± 2.39	3
`SplitWalker`			74.71 ± 2.53	4
`WalkletWalker`			72.06 ± 1.32	5
`AnonymousWalker`			65.29 ± 1.76	6
`HALKWalker(freq_threshold=0.01)`	4	250	**78.82** ± 1.50	1
`SplitWalker`			77.35 ± 4.01	2
`RandomWalker`			76.76 ± 6.06	3
`NGramWalker(grams=3)`			75.88 ± 3.90	4
`WalkletWalker`			73.82 ± 1.95	5
`AnonymousWalker`			66.47 ± 1.44	6
`HALKWalker(freq_threshold=0.01)`	6	250	**81.18** ± 4.87	1
`SplitWalker`			79.71 ± 2.16	2
`NGramWalker(grams=3)`			77.65 ± 1.95	3
`RandomWalker`			75.29 ± 2.16	4
`WalkletWalker`			71.76 ± 1.10	5
`AnonymousWalker`			67.65 ± 1.86	6

Table 8.5: Evaluation of the Accuracy of Walking Strategies for `MUTAG` According to the Maximum Depth per Walk.

In Table 8.5, regardless of the maximum depth per walk chosen for the same number of walks per entity, `HALKWalker` indicates a model's accuracy above `SplitWalker`. However, these two walking strategies provide better results for more significant maximum depths per walk. While `RandomWalker` indicates a good model's accuracy for small maximum depth per walk.

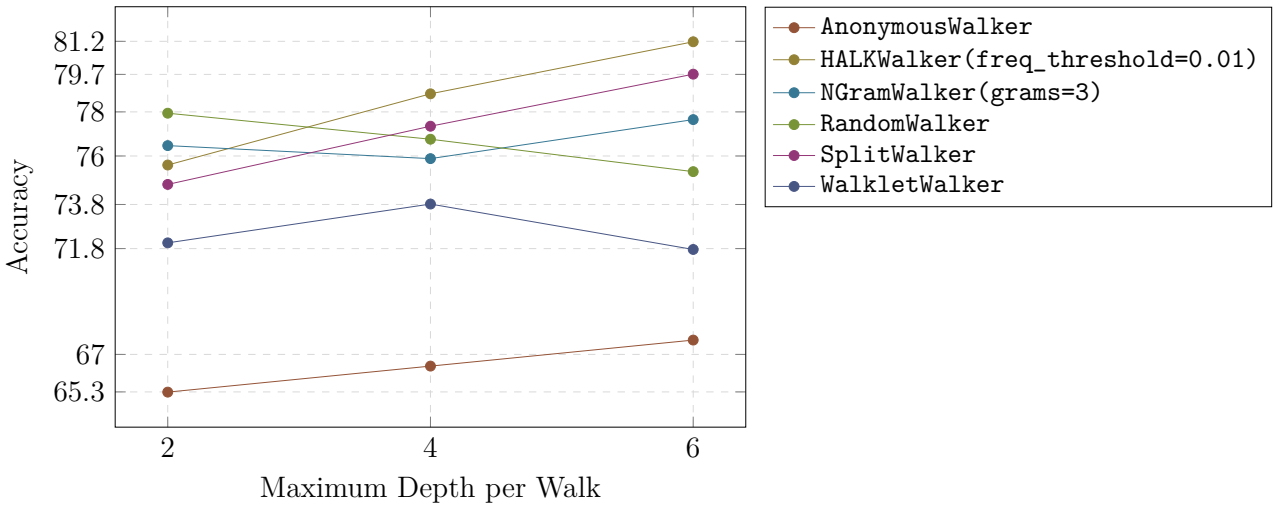


Figure 8.3: Evaluation of the Accuracy of Walking Strategies for MUTAG According to the Maximum Depth per Walk.

In Figure 8.3, the curve of `SplitWalker` and `HALKWalker` have an identical trajectory, except that `HALKWalker` indicates better model's accuracy than `SplitWalker`. In addition, they return better model's accuracy than the other walking strategies.

Walker	**Max. Depth**	**Max. Walks**	**Accuracy** (%)	**Rank**
`SplitWalker`	4	100	**79.12** ± 4.20	1
`HALKWalker(freq_threshold=0.01)`			77.65 ± 2.53	2
`WalkletWalker`			73.82 ± 1.95	3
`RandomWalker`			72.35 ± 3.77	4
`NGramWalker(grams=3)`			68.82 ± 3.99	5
`AnonymousWalker`			65.59 ± 1.18	6
`SplitWalker`	4	250	**77.35** ± 2.56	1
`HALKWalker(freq_threshold=0.01)`			76.76 ± 3.65	2
`RandomWalker`			76.18 ± 1.71	3
`NGramWalker(grams=3)`			73.24 ± 3.40	4
`WalkletWalker`			71.76 ± 1.10	5
`AnonymousWalker`			66.76 ± 1.18	6
`HALKWalker(freq_threshold=0.01)`	4	500	**79.12** ± 3.99	1
`SplitWalker`			77.35 ± 1.50	2
`WalkletWalker`			77.06 ± 3.55	3
`RandomWalker`			72.06 ± 1.32	4
`NGramWalker(grams=3)`			71.76 ± 1.10	5
`AnonymousWalker`			65.80 ± 1.44	6

Table 8.6: Evaluation of the Accuracy of Walking Strategies for `MUTAG` According to the Maximum Number of Walks per Entity

In Table 8.6, regardless of the number of walks chosen for the same maximum depth per walk, `SplitWalker` indicates a correct model's accuracy above the average of walking strategies. Specifically, `SplitWalker` allows an average model's accuracy of 78.53 %.

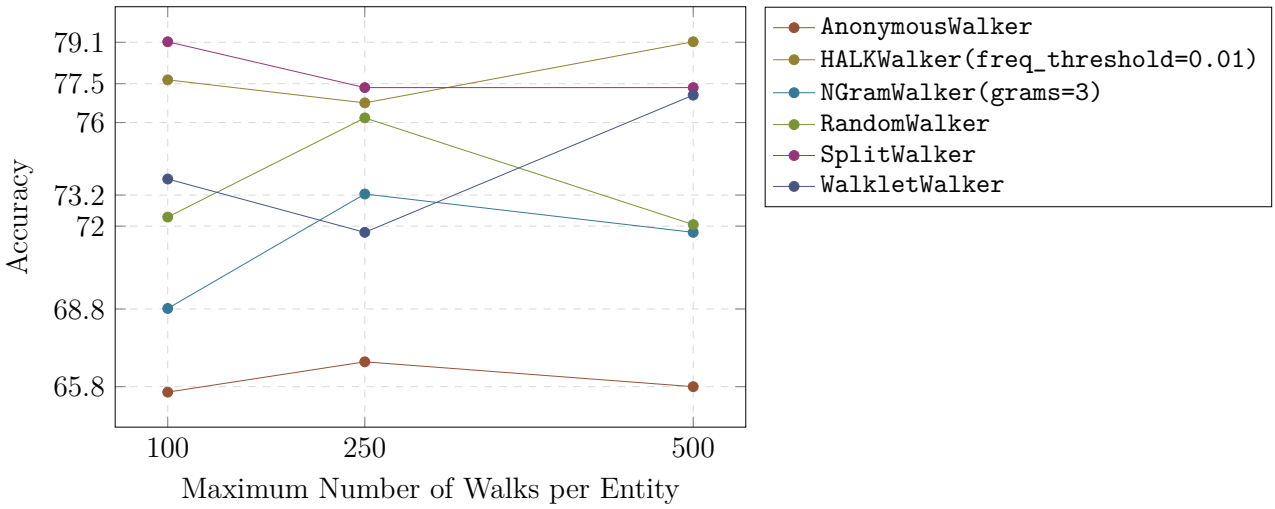


Figure 8.4: Evaluation of the Walking Strategies for Different Data Sets According to the Maximum Number of Walks per Entity.

In Figure 8.4, the model's accuracy with `SplitWalker` tends to stay around 77 % after a maximum number of walks of 250. In addition, `AnonymousWalker`, `NGramWalker`, and `RandomWalker` follow the same curve trajectory. Specifically, they have a peak of accuracy at a maximum number of walks per entity of 250. In contrast, the other walking strategies have a decrease of model's accuracy at this stage.

Walker	**Average Rank**
`HALKWalker(freq_threshold=0.01)`	1
`SplitWalker`	2
`RandomWalker`	3
`NGramWalker(grams=3)`	4
`WalkletWalker`	5
`AnonymousWalker`	6

Table 8.7: Evaluation of the Average Rank of the Walking Strategies.

In Figure 8.7, `HALKWalker(freq_threshold=0.01)` is the winning walking strategy in these benchmarks for `MUTAG`, followed by `SplitWalker`, and `RandomWalker`.

8.2.3 Sampling Strategies

To determine the accuracy impact of `WideSampler`, the latter is compared to other sampling strategies in a first time using `MUTAG` where only the number of walks and depth are variable. The number of standard entities for `MUTAG` being fixed at 340 trained entities, 68 of these serve as testing entities. As RDF2Vec is unsupervised, including the testing entities in the training set is not an issue.

Sampler	Max. Depth	Max. Walks	Accuracy (%)	Rank
ObjPredFreqSampler			**78.82** ± 3.17	1
ObjFreqSampler			77.94 ± 3.35	2
PredFreqSampler(inverse=True)			77.65 ± 2.35	3
WideSampler			76.76 ± 1.95	4
ObjPredFreqSampler(inverse=True)			76.76 ± 2.16	5
PredFreqSampler			76.76 ± 4.87	6
PageRankSampler(inverse=True,split=True,alpha=0.85)	2	100	76.18 ± 2.16	7
PageRankSampler(alpha=0.85)			76.47 ± 2.08	8
UniformSampler			76.18 ± 4.50	9
ObjFreqSampler(inverse=True,split=True)			75.88 ± 4.01	10
PageRankSampler(split=True,alpha=0.85)			74.71 ± 1.71	11
PageRankSampler(inverse=True,alpha=0.85)			74.41 ± 1.50	12
ObjFreqSampler(inverse=True)			73.53 ± 2.79	13
WideSampler			**77.35** ± 1.99	1
PredFreqSampler			76.18 ± 3.14	2
PageRankSampler(split=True,alpha=0.85)			75.88 ± 1.50	3
ObjFreqSampler(inverse=True,split=True)			75.88 ± 3.03	4
PageRankSampler(inverse=True,split=True,alpha=0.85)			75.88 ± 5.06	5
ObjFreqSampler(inverse=true)			75.59 ± 1.99	6
ObjFreqSampler	2	250	75.59 ± 4.71	7
ObjPredFreqSampler(inverse=True)			75.29 ± 1.10	8
UniformSampler			75.29 ± 1.44	9
ObjPredFreqSampler			75.29 ± 2.35	10
PageRankSampler(inverse=True,alpha=0.85)			75.00 ± 2.46	11
PageRankSampler(alpha=0.85)			74.12 ± 1.99	12
PredFreqSampler(inverse=True)			73.82 ± 2.53	13
ObjFreqSampler			**75.29** ± 2.16	1
PageRankSampler(alpha=0.85)			75.00 ± 0.00	2
WideSampler			74.71 ± 1.10	3
ObjFreqSampler(inverse=True,split=True)			74.71 ± 3.14	4
PageRankSampler(inverse=True,alpha=0.85)			74.12 ± 2.56	5
ObjPredFreqSampler(inverse=True)			73.82 ± 2.53	6
PageRankSampler(inverse=True,split=True,alpha=0.85)	2	500	73.82 ± 2.16	7
PageRankSampler(split=True,alpha=0.85)			73.82 ± 2.53	8
ObjPredFreqSampler			73.53 ± 1.32	9
UniformSampler			73.53 ± 2.63	10
PredFreqSampler			72.65 ± 2.73	11
ObjFreqSampler(inverse=True)			72.65 ± 2.73	12
PredFreqSampler(inverse=True)			72.35 ± 1.95	13

Table 8.8: Accuracy of Sampling Strategies for MUTAG (Part I).

In Table 8.8, for a maximum depth of walk of 2 with a maximum number of walks of 250, WideSampler indicates the best model's accuracy. In addition, the latter gives good precision models for a maximum number of walks of 100 and 500.

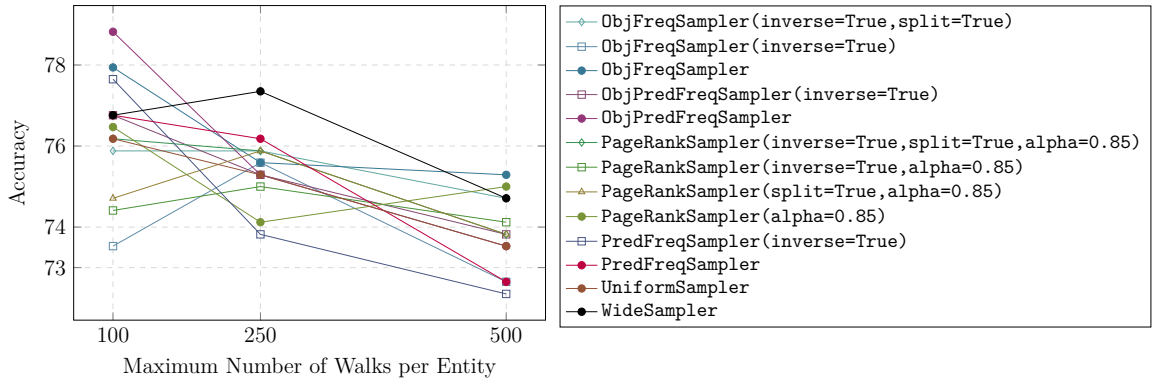


Figure 8.5: Sampling Strategies for MUTAG According to a Maximum Depth per Walk of 2.

In Figure 8.5, the model's accuracy for the different walking strategies is inversely proportional to the maximum number of walks per entity. Moreover, `WideSampler` and `PredFreqSampler` have two almost similar curves by their trajectory.

Sampler	Max. Depth	Max. Walks	Accuracy (%)	Rank
WideSampler			**78.82** ± 3.03	1
PredFreqSampler(inverse=True)			78.24 ± 2.53	2
ObjPredFreqSampler			75.29 ± 3.99	3
PageRankSampler(inverse=True,alpha=0.85)			75.00 ± 2.63	4
PageRankSampler(split=True,alpha=0.85)			74.12 ± 3.79	5
ObjFreqSampler(inverse=True)			73.82 ± 2.16	6
PredFreqSampler	4	100	73.82 ± 2.53	7
UniformSampler			73.24 ± 1.95	8
ObjPredFreqSampler(inverse=True)			72.94 ± 1.50	9
PageRankSampler(inverse=True,split=True,alpha=0.85)			72.65 ± 1.18	10
ObjFreqSampler(inverse=True,split=True)			72.65 ± 3.79	11
PageRankSampler(alpha=0.85)			72.65 ± 4.22	12
ObjFreqSampler			69.12 ± 5.73	13
ObjPredFreqSampler(inverse=True)			**79.41** ± 2.63	1
ObjFreqSampler			78.53 ± 2.56	2
PageRankSampler(inverse=True,alpha=0.85)			77.06 ± 2.56	3
ObjFreqSampler(inverse=True)			76.18 ± 3.14	4
PageRankSampler(inverse=True,split=True,alpha=0.85)			75.59 ± 2.88	5
ObjFreqSampler(inverse=True,split=True)			75.59 ± 1.76	6
WideSampler	4	250	75.00 ± 2.94	7
PredFreqSampler(inverse=True)			74.71 ± 2.85	8
PredFreqSampler			74.41 ± 2.73	9
UniformSampler			74.41 ± 2.56	10
PageRankSampler(alpha=0.85)			73.53 ± 2.28	11
PageRankSampler(split=True,alpha=0.85)			71.18 ± 2.39	12
ObjPredFreqSampler			68.82 ± 3.53	13
ObjPredFreqSampler(inverse=True)			**77.94** ± 0.93	1
ObjFreqSampler(inverse=True)			77.65 ± 1.95	2
UniformSampler			77.35 ± 1.18	3
PageRankSampler(split=True,alpha=0.85)			77.35 ± 2.73	4
ObjPredFreqSampler			77.06 ± 2.39	5
ObjFreqSampler(inverse=True,split=True)			76.18 ± 1.10	6
PredFreqSampler	4	500	75.59 ± 4.32	7
PageRankSampler(alpha=0.85)			75.59 ± 3.03	8
PageRankSampler(inverse=True,split=True,alpha=0.85)			75.29 ± 2.16	9
PageRankSampler(inverse=True,alpha=0.85)			75.29 ± 3.14	10
WideSampler			74.41 ± 2.20	11
ObjFreqSampler			74.41 ± 2.20	11
PredFreqSampler(inverse=True)			74.12 ± 4.12	12

Table 8.9: Accuracy of Sampling Strategies for `MUTAG` (Part II).

In Table 8.9, for a maximum depth per walk of 4 with a maximum number of walks of 100, `WideSampler` indicates the best model's accuracy. However, the model's accuracy of `WideSampler` is inversely proportional to the maximum number of walks per entity.

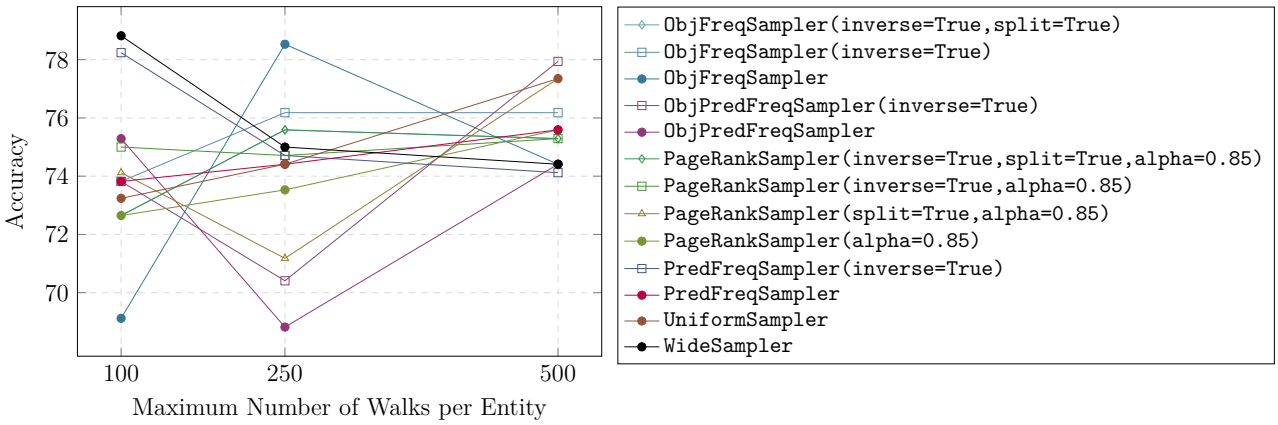


Figure 8.6: Sampling Strategies for `MUTAG` According to a Maximum Depth per Walk of 4.

In Figure 8.6, the model's accuracy for most of the walking strategies is proportional to the maximum number of walks per entity.

Sampler	Max. Depth	Max. Walks	Accuracy (%)	Rank
PredFreqSampler			**79.41** ± 2.46	1
PageRankSampler(split=True,alpha=0.85)			77.06 ± 3.03	2
ObjFreqSampler(inverse=True)			76.76 ± 3.40	3
WideSampler			76.76 ± 3.40	3
PageRankSampler(inverse=True,alpha=0.85)			76.76 ± 3.88	4
ObjFreqSampler			75.59 ± 5.06	5
ObjPredFreqSampler(inverse=True)	6	100	75.59 ± 2.73	6
PageRankSampler(inverse=True,split=True,alpha=0.85)			75.29 ± 1.71	7
PredFreqSampler(inverse=True)			75.29 ± 1.71	8
ObjFreqSampler(inverse=True,split=True)			74.12 ± 3.03	9
PageRankSampler(alpha=0.85)			73.53 ± 3.35	10
UniformSampler			73.24 ± 3.77	11
ObjPredFreqSampler			70.88 ± 6.47	12
PageRankSampler(inverse=True,alpha=0.85)			**80.00** ± 1.50	1
ObjFreqSampler(inverse=True,split=True)			78.53 ± 3.17	2
PredFreqSampler(inverse=True)			78.24 ± 4.30	3
ObjPredFreqSampler(inverse=True)			78.24 ± 6.20	4
PageRankSampler(alpha=0.85)			77.35 ± 2.39	5
PageRankSampler(inverse=True,split=True,alpha=0.85)			76.18 ± 2.53	6
ObjFreqSampler(inverse=True)	6	250	76.18 ± 4.30	7
UniformSampler			75.59 ± 1.76	8
PredFreqSampler			75.59 ± 3.03	9
ObjFreqSampler			75.00 ± 2.08	10
WideSampler			75.00 ± 4.83	11
ObjPredFreqSampler			73.24 ± 4.78	12
PageRankSampler(split=True,alpha=0.85)			72.65 ± 2.56	13
PredFreqSampler			**79.41** ± 3.08	1
ObjFreqSampler(inverse=True)			78.82 ± 1.50	2
WideSampler			78.53 ± 1.18	3
ObjFreqSampler			78.24 ± 1.95	4
PageRankSampler(inverse=True,split=True,alpha=0.85)			77.94 ± 1.32	5
UniformSampler			77.94 ± 1.86	6
PredFreqSampler(inverse=True)	6	500	77.35 ± 1.50	7
PageRankSampler(alpha=0.85)			77.35 ± 1.50	7
PageRankSampler(inverse=True,alpha=0.85)			77.35 ± 1.99	8
ObjPredFreqSampler			77.06 ± 1.99	9
PageRankSampler(split=True,alpha=0.85)			76.76 ± 2.16	10
ObjFreqSampler(inverse=True,split=True)			75.88 ± 1.76	11
ObjPredFreqSampler(inverse=True)			75.59 ± 3.55	12

Table 8.10: Accuracy of Sampling Strategies for MUTAG (Part III).

In Table 8.10, for a maximum depth of walk of 6 with a maximum number of walks of 500, `WideSampler` indicates a good model's accuracy. However, the latter indicates a poor model's accuracy for a maximum number of walks of 500.

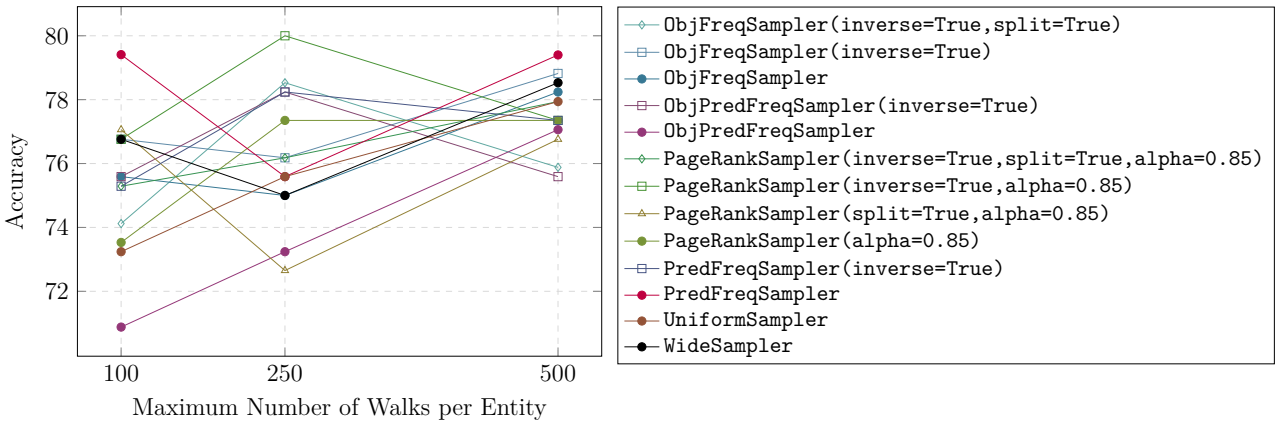


Figure 8.7: Sampling Strategies for `MUTAG` According to a Maximum Depth per Walk of 6.

In Figure 8.7, the curve of `WideSampler` shows the same trajectory as `PredFreqSampler` and `PageRankSampler(inverse=True,alpha=0.85)`. In addition, `ObjPreqFreqSampler` is proportional to the maximum number of walks per entity.

Sampler	**Average Rank**
`WideSampler`	1
`ObjPredFreqSampler(inverse=True)`	2
`PredFreqSampler`	3
`ObjFreqSampler`	4
`ObjFreqSampler(inverse=True)`	5
`PageRankSampler(inverse=True,alpha=0.85)`	6
`PageRankSampler(inverse=True,split=True,alpha=0.85)`	7
`ObjFreqSampler(inverse=True,split=True)`	8
`PageRankSampler(split=True,alpha=0.85)`	9
`PredFreqSampler(inverse=True)`	10
`ObjPredFreqSampler`	11
`UniformSampler`	11
`PageRankSampler(alpha=0.85)`	12

Table 8.11: Average Rank of the Sampling Strategies.

In Figure 8.11, `WideSampler` is the winning sampler strategy in these benchmarks for `MUTAG`, followed by `ObjPredFreqSampler(inverse=True)`, and `PredFreqSampler`.

Chapter 9

Discussion

The results obtained with BERT are less conclusive than those expected. Creating a BERT model requires considerable time, taking at least several hours or days of training. In addition, BERT has a lower accuracy than Word2Vec and FastText and needs to be re-trained for each KG, making its use not pragmatic. However, this evaluation does not mean that the use of BERT should be prohibited. Within this report, the research focused on three main fields:

1. the walks extraction from a KG;
2. the implementation of the BERT model for `pyRDF2Vec`;
3. the fine-tuning of BERT.

The research papers referred to in Section 2.1 are related to the implementation of BERT within KGs. However, most of them needed to create a new architecture that bears few or many changes compared to BERT. Therefore, it is interesting to examine the creation of these architectures.

9.1 BERT's Architecture

Understanding the implementation of the BERT variant architecture in the research papers would provide insight into the causes of the inconclusive results obtained.

9.1.1 KG-BERT

KG-BERT completes a KG by predicting missing tuples based on other existing tuples. The design of this architecture has the particularity that BERT can be trained based on extracted triples. For this training purpose, each triple injection consists of taking either the name or the description of their nodes. In this way, KG-BERT considers every node as a word. However, KG-BERT proposes two ways to inject these triples:

1. by varying the relevance of a relation using noise to train BERT to deduce the plausibility of this relation;
2. by using pairs of entities to train BERT to deduce the relation between these pairs of entities.

Based on this idea, this Master's thesis also tested BERT again on MUTAG and BGs. However, the results were still not more conclusive than those of Word2Vec and FastText.

Looking at the KG-BERT[1] implementation in more detail, the latter uses a sigmoid function to calculate the triple score instead of the original BERT softmax function. In addition, KG-BERT has adapted its cross-entropy loss computation by mainly considering the triple labels. These adjustments likely played an essential role in the success of KG-BERT. However, this BERT variant has not been compared with Word2Vec and uses other datasets than those used for this Master's thesis.

9.1.2 BERT-INT

BERT-INT predicts two identical entities across multiple KGs by using the name or description of the current entities, but this time, also of those of their neighbors. Since there is no propagation of information within neighboring entities, BERT-INT has the particularity to ignore the characteristic structure of the KGs.

Unlike KG-BERT, which has minor modifications compared to the BERT model, BERT-INT uses BERT as a basic representation unit. Specifically, BERT is used to generate embeddings of the entity name and description and their attributes, including values. From then on, the BERT-INT architecture combines several BERT units using a pairwise margin loss to fine-tune the resulting BERT interaction model. As a result, the BERT-INT architecture is more consistent. The resulting model consists of the name/description-view interaction plus the ones from the neighbor-view and the attribute-view.

9.1.3 Graph-BERT

Graph-BERT mainly helps with node classification and graph clustering tasks. The main feature of this variant BERT is that it relies solely on the Attention mechanism, without the necessity to use graph convolution or aggregation operations. To train the model, Graph-BERT uses unbound subgraphs sampled in their local contexts to avoid the use of KGs, which can be immense. For this purpose, there is an injection of sampled nodes and their local context into the model, which is then refined for the corresponding task. Finally, The Graph-BERT architecture is composed of five parts:

1. a linkless subgraph batching;
2. the node input vector embeddings;
3. a graph transformer-based encoder;
4. a representation fusion;
5. a functional component that generates different output according to the target application task.

Each of these parts plays an important role in the success of Graph-BERT. In addition, this variant also uses BERT as a small part of its architecture.

9.1.4 K-BERT

With Graph-BERT, K-BERT has one of the most advanced architectures compared to other BERT variants related to KG. The latter uses four modules:

1. Knowledge layer: injects relevant triples from a KG, converting the original sentence into a knowledge-rich sentence tree.
2. Embedding layer: convert a phrase tree into an embedding representation as the basic BERT architecture can do, except that the embedding layer is a sentence tree instead of a sequence of tokens.

[1] https://github.com/yao8839836/kg-bert

3. Seeing layer: uses a visible matrix to restrict the visible area of each token. This restriction ensures that an excess of knowledge does not lead to changes in the meaning of the original sentence. According to the authors, this layer is an integral part of the K-BERT success.

4. Mask-Transformer layer: allows BERT to receive the visible matrix as input using a multiple block stack of mask self-attention blocks.:

Even if there are similarities with BERT, K-BERT gets its good performance to a more advanced architecture than the one initially presented by BERT.

9.2 Future Works

From the set of architectures of the KG-oriented BERT variants stated above, KG-BERT is the one that keeps an almost similar architecture to the basic BERT architecture. Creating such a variant reinforces the idea that applying the BERT model to a KG without modifying its architecture is probably not interesting. This Master's thesis focused on only injecting pairs of walks to BERT. Probably injecting subject/object pairs as KG-BERT does to train BERT to predict predicates would lead to better results. However, given the lack of context and a non-suitable internal architecture, it is likely that the traditional BERT model would not be sufficient to have a clear improvement of model accuracy compared to Word2Vec. In particular, if Word2Vec uses the recommendations made by this Master's thesis to include the root node in more training samples.

As the objective of this Master's thesis was to evaluate BERT, no new variants have been implemented. Such an implementation would probably be more suitable for a PhD degree where more research time is allocated. However, this report has the privileges to closes several bad leads. It has allowed understanding better where its success comes from for its few evaluations within KGs. Added to that, none of these papers have evaluated their version of BERT with Word2Vec. Various possibilities for future work around this remain open such as evaluating the previously stated KG-oriented BERT models with Word2Vec and other embedding techniques. In addition, there is nothing to prevent the recreation of a new BERT architecture for KGs if this proves necessary in the long run. Finally, it may be interesting to evaluate `SplitWalker` and `WideSampler` on larger KGs and to know their impact compared to other strategies.

Chapter 10

Conclusion

RDF is a W3C standard that ensures that the data diversity remains machine-interpretable by encoding the semantics of these data. Semantic Web and Linked Open Data use this standard by interconnecting several sources using KGs, which are directed heterogeneous multigraphs composed of triples. Afterward, this type of graph is converted into a numerical vector and used for downstream ML tasks.

This conversion can be done with RDF2Vec, an unsupervised task-agnostic algorithm for numerically representing KG nodes. Since 2016, this algorithm has provided good results using Word2Vec, an embedding technique initially used in the NLP field. However, the NLP field has made other advances, notably related to the Attention mechanism published in 2014 by Bahdanau to solve the bottleneck problem of RNNs. Due to this mechanism, the Transformer architecture published in 2017 by Vaswani et al. was an alternative to RNNs, introducing two new Attention mechanisms: Scaled Dot-Product Attention and Multi-Head Attention. This architecture led to the creation of BERT, a state-of-the-art NLP embedding technique published in 2018.

Unlike Word2Vec, BERT allows generating contextualized embeddings using bidirectional representations. For this purpose, BERT can receive one or two sentences which the WordPiece algorithm will then tokenize. WordPiece allows BERT, on the one hand, to learn common subwords and, on the other hand, not to replace OOV words with a special token, being a rich source of information. Once the text corpus is tokenized, the BERT model is pre-trained using MLM and NSP as two unsupervised tasks. This pre-training is useful to mainly help overcome the lack of training data and allow the model to understand better a bidirectional representation of an input at the sentence and token level.

The objectives of this Master's thesis were to evaluate BERT with Word2Vec and FastText and extend RDF2Vec to a new walking strategy and sampling strategy. Such an evaluation is essential because few research papers have compared the classical BERT model with other embedding techniques. In addition, the minor use made of BERT with the KGs is done by creating one of its dedicated variances. Finally, creating new strategies allows improving the accuracy of a model for specific use-cases by focusing on extracting walks from a KG.

In order to evaluate BERT with FastText and Word2Vec on KGs, a dedicated implementation has been proposed. For BERT, it was not possible to use a pre-trained model and, therefore, necessary to create this model from scratch. For this purpose, the creation of the vocabulary of this new model included each special token and the nodes extracted from the walks by a walking strategy and a sampling strategy. For the training of BERT, it was first useful to tokenize the nodes by adding special tokens to the left and right of each node. Then, the pre-training focused only on the MLM since the NSP is of no interest since each walk has no semantic links with a second one. Finally, the training was done on a data collator dedicated to MLM, and a set of hyperparameters was provided, finding a compromise between training

time and model accuracy.

BERT's evaluation on `MUTAG` indicated a training time ranging from 25 minutes to several hours and a generally lower model accuracy than Word2Vec and FastText. These results showed that the model's accuracy is proportional to the maximum depth per walk and the maximum number of walks. Except for rare cases, the use of this BERT model for KGs is not significant enough. The main reason being its training time which can be excessively long with results similar to Word2Vec and FastText.

This Master's thesis proposed `SplitWalker` as a new walking strategy. Its principle is based on the decomposition of nodes according to a splitting function provided by the user. By default, `SplitWalker` splits each node according to its symbols, capital letters, and numbers. When comparing this strategy to others with `MUTAG`, the average rank of the walking strategies shows `SplitWalker` in second place, behind `HALKWalker`. However, a better splitting function could probably have improved these results even more.

In addition, a new sampling strategy has been introduced, namely `WideSampler`. This strategy mimics the classification of objects by humans, favoring as much as possible the common features between the objects. More precisely, `WideSampler` tends to favor the most frequent hops in a KG and those that allow access to more nodes. Moreover, its evaluation with `MUTAG` gives a good model accuracy, allowing it to finish first in the benchmarks' average. Added to that, it is likely that `WideSampler` has more impact on larger KGs.

Within this work, it was proposed to improve Word2Vec by extracting the parent and child nodes of a root node on the one hand and focusing the positioning of this root node within the walks. These recommendations can bring more contexts to the root node and improve its frequency of occurrence within the samples training. From then on, a better quality generation of root node embeddings.

This Master's thesis had some internal problems that resulted in missing benchmarks. These problems were related to the IDLab servers used. Their Stardog infrastructure generated significant variants during the benchmarks. More precisely, the version of Stardog used had a problem with the garbage collector, which prevented the correct memory deallocation. In addition, some benchmarks had exceptions thrown due to SPARQL endpoint not being available. This unavailability being due to updates and internal problems with the servers.

It was discussed the results obtained. The research papers using BERT pretended that the classical BERT model was not interesting with KGs. One of the reasons for this is that BERT is limited by its inputs and its architecture is not optimized to be used for KGs. Most of the research papers have developed their variant of BERT, which can be simple or complex. At most simple, some variants have replaced the softmax activation function with a sigmoid activation function. The traditional BERT model was only a tiny part of a more significant architecture at the most complex.

Research directions for future work have been proposed. Firstly it was recommended to evaluate `SplitWalker` and `WideSampler` for larger KGs to know more about their impact. Added to this, the traditional BERT model with larger KGs reinforces the idea that its use is limited. However, another test would be not to inject pairs of different walks, but rather pairs of `(subject, object)` 2-tuple to let BERT deduce the predicate. It was said that it would also be possible to compare Word2Vec and Fast to other BERT variants to know their impact better. In case of bad results, there is also the possibility to create a new BERT variant.

Bibliography

Jay Alammar. The illustrated transformer, Jun 2018a. URL http://jalammar.github.io/illustrated-transformer/. Accessed: May 2021.

Jay Alammar. Visualizing a neural machine translation model (mechanics of seq2seq models with attention), May 2018b. URL https://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/. Accessed: May 2021.

Majdi Beseiso and Saleh Alzahrani. An empirical analysis of bert embedding for automated essay scoring. *International Journal of Advanced Computer Science and Applications*, 11: 204–210, 11 2020. doi: 10.14569/IJACSA.2020.0111027.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. Enriching word vectors with subword information. *Trans. Assoc. Comput. Linguistics*, 5:135–146, 2017. URL https://transacl.org/ojs/index.php/tacl/article/view/999.

Antoine Bordes, Nicolas Usunier, Alberto García-Durán, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pages 2787–2795, 2013. URL https://proceedings.neurips.cc/paper/2013/hash/1cecc7a77928ca8133fa24680a88d2f9-Abstract.html.

Paolo Ceravolo, Antonia Azzini, Marco Angelini, Tiziana Catarci, Philippe Cudré-Mauroux, Ernesto Damiani, Alexandra Mazak, Maurice van Keulen, Mustafa Jarrar, Giuseppe Santucci, Kai-Uwe Sattler, Monica Scannapieco, Manuel Wimmer, Robert Wrembel, and Fadi A. Zaraket. Big data semantics. *J. Data Semant.*, 7(2):65–85, 2018. doi: 10.1007/s13740-018-0086-2. URL https://doi.org/10.1007/s13740-018-0086-2.

Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. In Jian Su, Xavier Carreras, and Kevin Duh, editors, *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, pages 551–561. The Association for Computational Linguistics, 2016. doi: 10.18653/v1/d16-1053. URL https://doi.org/10.18653/v1/d16-1053.

Jan Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 577–585, 2015. URL http://papers.nips.cc/paper/5847-attention-based-models-for-speech-recognition.

Michael Cochez, Petar Ristoski, Simone Paolo Ponzetto, and Heiko Paulheim. Biased graph walks for RDF graph embeddings. In Rajendra Akerkar, Alfredo Cuzzocrea, Jannong Cao, and Mohand-Said Hacid, editors, *Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics, WIMS 2017, Amantea, Italy, June 19-22, 2017*, pages 21:1–21:12. ACM, 2017. doi: 10.1145/3102254.3102279. URL https://doi.org/10.1145/3102254.3102279.

DeepAI. *Cosine Similarity*, May 2019a. URL https://deepai.org/machine-learning-glossary-and-terms/cosine-similarity. Accessed: March 2021.

DeepAI. *Distributed Representations*, May 2019b. URL https://deepai.org/machine-learning-glossary-and-terms/distributed-representation. Accessed: March 2021.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019. doi: 10.18653/v1/n19-1423. URL https://doi.org/10.18653/v1/n19-1423.

Kawin Ethayarajh. How contextual are contextualized word representations? comparing the geometry of bert, elmo, and GPT-2 embeddings. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 55–65. Association for Computational Linguistics, 2019. doi: 10.18653/v1/D19-1006. URL https://doi.org/10.18653/v1/D19-1006.

Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 1243–1252. PMLR, 2017. URL http://proceedings.mlr.press/v70/gehring17a.html.

Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In Yee Whye Teh and D. Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, volume 9 of *JMLR Proceedings*, pages 297–304. JMLR.org, 2010. URL http://proceedings.mlr.press/v9/gutmann10a.html.

Sijun He. *Word Embeddings*, Sep 2018. URL https://sijunhe.github.io/blog/2018/09/12/word-embeddings/. Accessed: April 2021.

Dan Hendrycks, Xiaoyuan Liu, Eric Wallace, Adam Dziedzic, Rishabh Krishnan, and Dawn Song. Pretrained transformers improve out-of-distribution robustness. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 2744–2751. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.acl-main.244. URL https://doi.org/10.18653/v1/2020.acl-main.244.

Balaji Kamakoti. *Introduction to Knowledge Graph Embedding with DGL-KE*, Jun 2020. URL https://towardsdatascience.com/introduction-to-knowledge-graph-embedding-with-dgl-ke-77ace6fb60ef. Accessed: April 2021.

Akshay Khatri and Pranav P. Sarcasm detection in tweets with BERT and glove embeddings. In Beata Beigman Klebanov, Ekaterina Shutova, Patricia Lichtenstein, Smaranda Muresan, Chee Wee Leong, Anna Feldman, and Debanjan Ghosh, editors, *Proceedings of the Second Workshop on Figurative Language Processing, Fig-Lang@ACL 2020, Online, July 9, 2020*, pages 56–60. Association for Computational Linguistics, 2020. URL https://www.aclweb.org/anthology/2020.figlang-1.7/.

Weijie Liu, Peng Zhou, Zhe Zhao, Zhiruo Wang, Qi Ju, Haotang Deng, and Ping Wang. K-BERT: enabling language representation with knowledge graph. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 2901–2908. AAAI Press, 2020. URL https://aaai.org/ojs/index.php/AAAI/article/view/5681.

Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In Lluís Màrquez, Chris Callison-Burch, Jian Su, Daniele Pighin, and Yuval Marton, editors, *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pages 1412–1421. The Association for Computational Linguistics, 2015. doi: 10.18653/v1/d15-1166. URL https://doi.org/10.18653/v1/d15-1166.

Chris McCormick. *Word2Vec Tutorial Part 2 - Negative Sampling*, Jan 2017. URL http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-negative-sampling/. Accessed: April 2021.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013. URL http://arxiv.org/abs/1301.3781.

Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In Robert G. Cowell and Zoubin Ghahramani, editors, *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics, AISTATS 2005, Bridgetown, Barbados, January 6-8, 2005*. Society for Artificial Intelligence and Statistics, 2005. URL http://www.gatsby.ucl.ac.uk/aistats/fullpapers/208.pdf.

Sourav Mukherjee, Tim Oates, and Ryan Wright. Graph Node Embeddings using Domain-Aware Biased Random Walks. *CoRR*, abs/1908.02947, 2019. URL http://arxiv.org/abs/1908.02947.

Yifan Peng, Shankai Yan, and Zhiyong Lu. Transfer learning in biomedical natural language processing: An evaluation of BERT and elmo on ten benchmarking datasets. In Dina Demner-Fushman, Kevin Bretonnel Cohen, Sophia Ananiadou, and Junichi Tsujii, editors, *Proceedings of the 18th BioNLP Workshop and Shared Task, BioNLP@ACL 2019, Florence, Italy, August 1, 2019*, pages 58–65. Association for Computational Linguistics, 2019. doi: 10.18653/v1/w19-5006. URL https://doi.org/10.18653/v1/w19-5006.

Petar Ristoski and Heiko Paulheim. A comparison of propositionalization strategies for creating features from linked open data. In Ilaria Tiddi, Mathieu d'Aquin, and Nicolas Jay, editors, *Proceedings of the 1st Workshop on Linked Data for Knowledge Discovery co-located with European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD 2014), Nancy, France, September 19th, 2014*, volume 1232 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2014. URL http://ceur-ws.org/Vol-1232/paper1.pdf.

Petar Ristoski, Jessica Rosati, Tommaso Di Noia, Renato De Leone, and Heiko Paulheim. Rdf2vec: RDF graph embeddings and their applications. *Semantic Web*, 10(4):721–752, 2019a. doi: 10.3233/SW-180317. URL https://doi.org/10.3233/SW-180317.

Petar Ristoski, Jessica Rosati, Tommaso Di Noia, Renato De Leone, and Heiko Paulheim. Rdf2vec: RDF graph embeddings and their applications. *Semantic Web*, 10(4):721–752, 2019b. doi: 10.3233/SW-180317. URL https://doi.org/10.3233/SW-180317.

Budhaditya Saha, Sanal Lisboa, and Shameek Ghosh. Understanding patient complaint characteristics using contextual clinical BERT embeddings. In *42nd Annual International Conference of the IEEE Engineering in Medicine & Biology Society, EMBC 2020, Montreal, QC, Canada, July 20-24, 2020*, pages 5847–5850. IEEE, 2020. doi: 10.1109/EMBC44109.2020.9175577. URL https://doi.org/10.1109/EMBC44109.2020.9175577.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108, 2019. URL http://arxiv.org/abs/1910.01108.

Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In Aldo Gangemi, Roberto Navigli, Maria-Esther Vidal, Pascal Hitzler, Raphaël Troncy, Laura Hollink, Anna Tordai, and Mehwish Alam, editors, *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings*, volume 10843 of *Lecture Notes in Computer Science*, pages 593–607. Springer, 2018. doi: 10.1007/978-3-319-93417-4_38. URL https://doi.org/10.1007/978-3-319-93417-4_38.

Amit Singhal. Introducing the knowledge graph: things, not strings, May 2012. URL https://www.blog.google/products/search/introducing-knowledge-graph-things-not. Accessed: May 2021.

Xiaobin Tang, Jing Zhang, Bo Chen, Yang Yang, Hong Chen, and Cuiping Li. BERT-INT: A bert-based interaction model for knowledge graph alignment. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 3174–3180. ijcai.org, 2020. doi: 10.24963/ijcai.2020/439. URL https://doi.org/10.24963/ijcai.2020/439.

Ahmad Al Taweel and Heiko Paulheim. Towards Exploiting Implicit Human Feedback for Improving RDF2vec Embeddings. *CoRR*, abs/2004.04423, 2020. URL https://arxiv.org/abs/2004.04423.

Gilles Vandewiele, Bram Steenwinckel, Terencio Agozzino, Michael Weyns, Pieter Bonte, Femke Ongenae, and Filip De Turck. pyRDF2Vec: Python Implementation and Extension of RDF2Vec. IDLab, 2020a. URL https://github.com/IBCNServices/pyRDF2Vec.

Gilles Vandewiele, Bram Steenwinckel, Pieter Bonte, Michael Weyns, Heiko Paulheim, Petar Ristoski, Filip De Turck, and Femke Ongenae. Walk Extraction Strategies for Node Embeddings with RDF2Vec in Knowledge Graphs. *CoRR*, abs/2009.04404, 2020b. URL https://arxiv.org/abs/2009.04404.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Mani Vegupatti, Matthias Nickles, and Bharathi Raja Chakravarthi. Simple question answering over a domain-specific knowledge graph using BERT by transfer learning. In Luca Longo, Lucas Rizzo, Elizabeth Hunter, and Arjun Pakrashi, editors, *Proceedings of The 28th Irish Conference on Artificial Intelligence and Cognitive Science, Dublin, Republic of Ireland, December 7-8, 2020*, volume 2771 of *CEUR Workshop Proceedings*, pages 289–300. CEUR-WS.org, 2020. URL http://ceur-ws.org/Vol-2771/AICS2020_paper_42.pdf.

Thomas Wood. *Softmax Function*. https://deepai.org/machine-learning-glossary-and-terms/softmax-layer, May 2019. Accessed: April 2021.

Thomas Wood. *Unsupervised Learning*. https://deepai.org/machine-learning-glossary-and-terms/unsupervised-learning, Aug 2020. Accessed: March 2021.

Liang Yao, Chengsheng Mao, and Yuan Luo. KG-BERT: BERT for knowledge graph completion. *CoRR*, abs/1909.03193, 2019. URL http://arxiv.org/abs/1909.03193.

Jiawei Zhang, Haopeng Zhang, Congying Xia, and Li Sun. Graph-bert: Only attention is needed for learning graph representations. *CoRR*, abs/2001.05140, 2020. URL https://arxiv.org/abs/2001.05140.