TripLan2vec: Leveraging Pre-Trained Language Models for Inductive Triple Embeddings

Adriaan Thomas Kisjes
(4279093)

April 2023

Abstract

Many organizations and data dependent applications deal with the fact that data is often incomplete and siloed across multiple knowledge bases. The semantic web and knowledge graphs are powerful tools that mitigate this by allowing rule-based systems to complete and connect different knowledge bases. To enable the use of more advanced machine-learning algorithms such as logistic regression or neural networks, knowledge graphs need to be transformed into some kind of numeric input. In the field of neural language processing this has been solved with vector embeddings, where for each word a vector is learned that captures its semantic meaning. There exist many knowledge graph embedding techniques inspired by natural language processing, one of which is Triple2vec where triples (two entities and the relation connecting them) are embedded as a whole. Triple2vec is innovative because it captures both the graph topology as well as the heterogeneity of knowledge graphs, where other methods often focus on just one of those aspects. This thesis proposes to build on triple embeddings by developing TripLan2vec: a triple embedding technique that uses a pre-trained language model to generate embeddings based on textual descriptions. This enriches the embeddings by both capturing graph structure as well as natural language semantics. Moreover, it also enables the triple embeddings to be generated inductively with just a description as input, this means that triples that are not part of the training process can still be embedded, unlike with Triple2vec. During evaluation it was shown that TripLan2vec performs well at discriminating between true and false triples, and at predicting whether two triples are neighbours. In inductive evaluation, where just part of the training data was available, TripLan2vec outperforms most other methods.

Keywords— Knowledge Graph, Natural Language Model, Triple Embedding

Contents

1 Introduction **4**
1.1 General . . . 4
1.2 Relevance of the Research . . . 5
1.3 Research Questions . . . 5

2 Core Concepts **7**
2.1 Feedforward Neural Networks . . . 7
2.1.1 Practice of Training a Feedforward Neural Network . . . 7
2.2 Entity Embedding Techniques . . . 7
2.2.1 Entity Vector Embeddings . . . 7
2.2.2 Skip Gram Architecture . . . 8
2.2.3 Negative Sampling . . . 9
2.3 BERT . . . 10
2.3.1 Using BERT . . . 10
2.3.2 BERT Sentence Embeddings . . . 10
2.4 Knowledge Graphs . . . 11

3 Overview of Knowledge Graph Embedding Techniques **13**
3.1 TransE and Derivatives . . . 13
3.2 BERT Powered Approaches . . . 13
3.2.1 KG-BERT . . . 13
3.2.2 BERT for Link Prediction . . . 14
3.3 Walk Based Approaches . . . 14
3.3.1 Triple Embeddings . . . 14
3.3.2 Random Walks . . . 15

4 Methodology: Training Pipeline **19**
4.1 Pipeline Overview . . . 19
4.2 Parameter Analysis . . . 20
4.3 Triple Description to Language Embedding . . . 20
4.4 Walk Extraction . . . 21
4.4.1 Line graph Transformation . . . 21
4.4.2 Determining the Edge Weights . . . 21
4.4.3 The complexity of Node2vec walk extraction . . . 21
4.4.4 FastWalk: Extracting Walks from a Line Graph . . . 22
4.4.5 Variable Speed vs Variable Accuracy . . . 26
4.4.6 Walk Parameters . . . 27
4.5 Mapping from BERT Sentence Embeddings to Triple Embeddings . . . 27
4.5.1 Software and Hardware used . . . 27
4.5.2 Network Architecture . . . 27
4.5.3 Network Hyper Parameters . . . 29
4.5.4 Training the Network . . . 32

5 Methodology: Evaluation **33**
5.1 Evaluation Requirements . . . 33
5.2 Evaluation Benchmarks . . . 34
5.2.1 Triple Labeling . . . 34
5.2.2 Recommendation Engine . . . 34
5.2.3 Triple Classification . . . 35

5.2.4 Link Prediction and Relation Prediction . . . 35
5.2.5 Conclusion . . . 36
5.3 Evaluation of FastWalk and Negative Sample Batching . . . 36
5.4 Exploratory Evaluation . . . 36
5.4.1 Datasets . . . 37
5.4.2 Cosine Similarity . . . 37
5.4.3 Neighbour Prediction . . . 37
5.4.4 Entity extraction . . . 37
5.4.5 Evaluation of Individual Pipeline Processes . . . 38

6 Results **39**
6.1 Triple Classification . . . 39
6.2 Exploratory Evaluation . . . 39
6.2.1 Cosine Similarity . . . 39
6.2.2 Neighbour Prediction . . . 40
6.2.3 Entity Extraction . . . 40
6.2.4 Evaluation of Individual Pipeline Processes . . . 40
6.3 FastWalk . . . 41

7 Discussion **43**
7.1 Limitations . . . 43
7.1.1 Evaluation . . . 43
7.1.2 Finetuning . . . 43
7.2 Strengths . . . 43
7.3 Future Work . . . 44
7.3.1 Evaluation . . . 44
7.3.2 Hybrid Models . . . 44
7.3.3 Beyond Language Models . . . 44

8 Conclusion **45**

1 Introduction

1.1 General

The power of the web is partially due to its decentralization, it allows everyone with the means, to host web-pages and publish whatever they want for the world to see. This allows for the existence of Wikipedia, IMDb, or the online edition of The New York Times to name a few. But the price for this abundance is that all this data is only loosely interconnected. The New York Times might mention the film Titanic (1996) in its review of Avatar 2 (2022) without any context, and although IMDb has extensive data on the film Titanic and Wikipedia has a long article on the historic event. Combining the information and data from these three different sources requires either manual labor or extensive indexing and advanced artificial intelligence. To make it easier for automated systems to make sense of the dispersed and decentralized knowledge bases on the web, the semantic web was developed. It strives to achieve this with multiple concepts and technologies of which two are relevant for this thesis: Firstly, universal identifiers that make it possible to determine the 'Titanic' mentioned in The New York Times is the same as the 'Titanic' page on IMDb. Secondly, a universal data structure that allows for the automatic merging of multiple distinct knowledge bases, once common concepts are determined [1]. This data structure is known as a knowledge graph, that as the name suggests utilizes a graph structure. In this graph nodes represent individual concepts and edges represent relationships between concepts [10], a more in-dept explanation is given in Section 2.4. Another way to look at this is as a list of triples consisting of a subject, predicate, and an object, whereby the subject and object are entities/concepts connected by a predicate that describes the relationship between them. Given two triples from two distinct knowledge bases (Leonardo DiCaprio, Played In, Titanic) and (Kate Winslet, Played In, Titanic) with the knowledge that both Titanics mentioned are in fact the same, simple rule-based logic can be used to determine that Kate Winslet and Leonardo DiCaprio played in the same movie. These technologies are now extensively used in practice, for example by Apple to develop Siri [46], or by Google for a host of applications [10].

With the technologies available to automatically combine different knowledge bases and extract new information from them using rule-based systems, the next logical evolution is to enable the use of more advanced machine learning techniques such as neural networks or logistic regression. The big challenge with this is transforming information from a knowledge graph into a suitable input for such systems, because they work best with numeric data. In the field of natural language processing this has been solved with word embeddings [24], these are numerical vector representations of individual words. The vectors are learned in such a way that they capture the semantic meaning of the target word. There have been multiple techniques developed to apply this to knowledge graphs, most of which generate embeddings for individual entities and predicates [40]. This has been shown to be effective at classifying entities [28] and knowledge graph completion [4] among other use cases.

The embedding technique this thesis will focus on is Triple2vec [12] where an entire triple is embedded in vector form. In the Titanic example this would mean that one vector represents the entire triple (Kate Winslet, Played In, Titanic). This has the dis-

advantage that it is harder to perform operations on individual entities, but is has been shown to outperform other embedding techniques in: triple classification [12], recommendation [12], and fact checking [31]. The reason for this is that triples all relate homogeneously with each other, they either share an entity or they do not, while entities in a knowledge graph can have many different types of relations (Played In, Is A, or Lives In). Homogeneity in relationship types simplifies the embedding method, and as shown by Triple2vec [12], enables more powerful embeddings in some cases.

This thesis thrives to build on triple embeddings by developing a technique that can generate inductive triple embeddings. In Triple2vec [12] embeddings are only available for triples that are part of the training process, so if new triples are encountered in a downstream application, either expensive retraining or a work around is required. With an inductive technique some external information describing the triple is used to generate an embedding. This means that triples not originally part of the training process can still be embedded. In the case of this thesis, language embeddings will be used for input as it has been shown by KG-BERT [42] that language models can effectively capture the meaning of triples and can then be used for multiple knowledge graph related tasks.

1.2 Relevance of the Research

The relevance of this research comes from the fact that triple embeddings are a new concept for which, to the best of my knowledge, only two methods exist of generating them. First Triple2vec [12] that uses a method inspired by Node2vec [13], second Lognet [32] that uses a graph neural network. Both methods are not capable of generating these embeddings inductively. The inductiveness of TripLan2vec will enable triple embeddings to be used in different types of downstream tasks than currently possible. Moreover, combining language models with graph structure has the potential of making the embeddings more meaningful than embeddings purely based on one aspect.

1.3 Research Questions

The research will be guided by answering the following research questions:

> **Main research question**: *How well do natural-language-enhanced triple embeddings perform?*

The main question is decomposed into the following sub questions:

> **Sub question 1**: *What is the best strategy for evaluating TripLan2vec triple embeddings?*

To answer how well natural-language-enhanced triples perform, first an evaluation metric needs to be determined. This is done by comparing different existing evaluation metrics on whether they play into the strengths of the developed technique, whether the metric can be used to compare with other methodologies, and if possible if the metric can give an indication on the methods performance in real world tasks.

Sub question 2*: How well do inductive natural-language-enhanced triple embeddings perform?*

Besides testing how well the triple embeddings perform with full training data, it is also important to demonstrate the inductive capabilities. This is done by taking increasingly smaller subsets of the full training data and performing the same evaluation method mentioned in the first sub question.

Sub question 3*: What is the best strategy to determine the optimal parameters and hyper parameters for the neural network that maps from language embedding space to triple embedding space?*

The mapping from language embedding space to triple embedding space will be done with a neural network. To get an optimal mapping function, it is important to carefully evaluate the parameters chosen. To do this an evaluation method is required that preferably does not required the entire training and evaluation pipeline to be run for every possible combination of parameters.

2 Core Concepts

This chapter gives an overview of how neural networks work, how they can be used to gain word embeddings, and finally what knowledge graphs are.

2.1 Feedforward Neural Networks

A neural network is a powerful classification tool that is inspired by the biological structure of the brain [36]. It consists of neurons (nodes) and synapses (edges) that connect different nodes with each other, in a feedforward neural network (FNN) these edges all 'point in the same direction'. This means that there is a one way flow of information from a set of input nodes through different layers of middle nodes (referred to as hidden nodes/layers) to a set of output nodes. In Figure 1 a simple example of a FNN is shown, this example has one layer of hidden nodes, four input nodes (I) and two output nodes (O). As in the brain, edges have a varied signal propagation strength, some edges give a strong signal when triggered others a weak signal, in a neural network this variance is referred to as edge weight. In a node the sum of all incoming edge signals is fed into an activation function, the result of which is propagated further into the network via its outgoing edges. The activation function simulates the activation of a neuron, this can be as simple as a threshold value, if the total input is too low the node propagates nothing, if the total input is higher than the threshold it propagates maximally (for example 1.0). In practise this activation function is often continuous. A trained network can transform a set of input numbers into a set of output numbers, or an input vector into an output vector, and "it has been shown by numerous workers that [multi-layer feedforward neural networks] can accurately represent the mappings of essentially all reasonably well-behaved functions" [22]. FNNs have been applied to many different problems, from classifying fruits based on images [43], or recognizing handwritten text [29], to creating word embeddings [24].

2.1.1 Practice of Training a Feedforward Neural Network

To perform the process of training an FNN a number of practical considerations need to be made: number of layers, size of the layers, training time, and using stochastic or batch learning[19] to name a few. In practice it is hard to determine beforehand what the best parameters will be, so it is to be expected that during the thesis, time will be spend on finetuning the network and figuring out what works best. Some decisions can be made beforehand, for example in theory a single layer network would suffice to map one vector space to another as stated before[22], this also has the advantage that it limits training time. For other parameters it would make sense to look at applications of FFNs in similar projects and mimic their parameters as a starting point.

2.2 Entity Embedding Techniques

2.2.1 Entity Vector Embeddings

Vector representations of concrete entities are a way to encapsulate the 'meaning' or 'semantics' of a concrete entity in numbers. These vector representations are helpful when

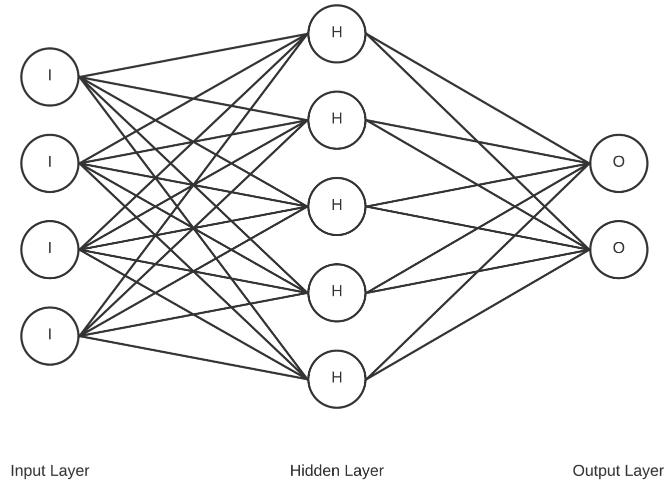

Figure 1: An example of a single layer, feedforward neural network.

training neural networks, if a networks is supposed to predict the next word in a sequence for example it would not make sense to use dictionary order as a numeric representation. With dictionary order the sentence 'petting the car' would be scored better as 'petting the dog' when the desired result is 'petting the cat'. With a vector embedding an entity is represented not by one number but by a sequence of numbers (or a point in multi-dimensional space), this allows for a much more nuanced representation. Node2vec [24] was for a while the state of the art method of gaining word embeddings, it is based on the assumption that the meaning of a word is defined by the company it keeps. If two words (e.g. cat and dog) are often surrounded by the same words for example 'petting the cat' and 'petting the dog', they should have a vector representation that is more similar. Their approach was extremely effective and even enabled logical relationships to be solved with vector math, for $X = vector('biggest') - vector('big') + vector('small')$ the closest word embedding to X is $vector('smallest')$.

2.2.2 Skip Gram Architecture

To create these word embeddings based on the neighbouring words, a more complicated network architecture is needed than a simple FNN. In Figure 2 the tail end of the skip gram architecture is shown, as this will be relevant in later sections. It is a network with two layers but both layers have multiple instances. The first layer represents the entity embedding and has one instance per entity. The second layer is a prediction layer that predicts the neighbouring entity, and has a different instance per neighbour (in this case for neighbours before and after the target entity). The resulting vector is not a vector embedding but a one-hot-encoded vector. This means that every dimension of the vector represents an entity and is set to zero, except the target entity which is set to one, with just three entities {'Cat', 'Bird', 'Koala'} the vector $[1.0, 0.0, 0.0]$ represents

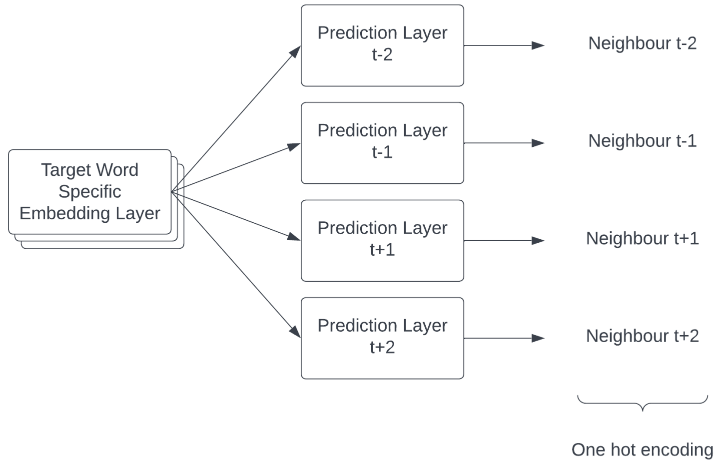

Figure 2: Skip Gram Architecture

'Cat'. In reality the resulting vector will likely be more fuzzy, for example the vector $[0.7, 0.3, 0.0]$ represents mostly 'Cat' but also 'Bird'. To ensure that the network only returns normalized vectors (sum of total values equals one and all values are between zero and one) a Softmax layer is used [5]. After training the prediction layers are discarded and the weighs of the embedding layers are kept as they are embeddings of the entities. An alternative to skip gram is the continuous bag of words architecture, in this case the two layers are switched, meaning that instead of using a target entity to predict multiple neighbouring entities, multiple neighbouring entities are used to predict one target entity. Both methods are introduced with Word2vec [24], but skip gram is preferred when the data corpus is not extremely extensive.

2.2.3 Negative Sampling

To provide the normalized one-hot-encoded vector the Softmax module needs to sum all incoming edges to all terms of the final vector for each neighbour prediction. [24] This could result in a large number of operations if for example the final one-hot-encoded vector has 10,000 terms and the hidden layer 100, this is already one million operations. In the original paper this is mitigated by using Hierarchical Softmax [24], here the output layer is structured like a tree. The tree structure means that only the nodes between the correct neighbour and the root of the tree need to be calculated and updated, with a well balanced tree this scales logarithmic instead of linear. But in a followup paper Mikolov et al. proposed a more efficient approach, that is also more accurate than pure Hierarchical Sofmax, Negative Sampling [25]. Negative Sampling works, in the case of Word2vec, by sampling a number of false or negative neighbours for a target-neighbour word pair, and only updating the nodes of these words. So instead of updating between 10,000 or one million plus nodes (in the case of pure Softmax) only between 2 to 20 are updated.

2.3 BERT

BERT [9] stands for Bidirectional Encoder Representations from Transformers, and like Word2vec generates word embeddings in vector space. But where Word2vec generates one universal vector representation BERT generates word representations with sentence context in mind, this means that the word 'bank' for example might have a different vector representation depending on its use. At first it might seem like this makes the word embeddings less useful because they are no longer universal between sentences, but it allows for the distinction between word uses. In both 'river bank' and 'bank robber' the word 'bank' occurs, but they have a completely different meaning that Word2vec can not encapsulate. Distinct word embeddings by context is not the only thing that makes BERT special, ELMO [23] is another word embedding technique that does this. What distinguishes BERT from ELMO is that ELMO only uses context from previous words in the sentence while BERT uses context from the full sentence, this is also what makes BERT bidirectional, demonstrated again with the text snippets 'river bank' and 'bank robber' it is immediately clear that context can occur on both sides of the target word. Recently BERT has been improved upon in some ways by GPT-3 [6] partially by being trained on an astronomically larger dataset, but as GPT-3 is not open source BERT is still preferred.

2.3.1 Using BERT

Because BERT needs context BERT takes one or two complete sentences as input, and returns a list of word embeddings for each word in the sentence(s). These word embeddings are then ready to be used directly for downstream tasks, but BERT also provides a fine-tuning feature, so BERT can actually be trained (or fine-tuned) to perform the downstream task directly. An example of this is next-sentence-prediction [9] where BERT is fine-tuned to use the generated word embeddings to determine if two sentences are likely to be consecutive. Two other important considerations to be made are input length and model size. The input of BERT is limited to 512 word tokens which are something else as words. To limit the number of possible words to embed, BERT splits longer words into word tokens. For example 'surfboarding' will be split into the tokens 'surf', '##board', and '##ing', the hashtags are there to indicate that the token is part of a previous word. As for model size, BERT has two options: base and large, BERT-Base has 12 hidden layers and BERT-Large 24, there is a significant difference in performance but they both outperform GPT-2 and ELMO.

2.3.2 BERT Sentence Embeddings

BERTs main purpose is to either extract word embeddings or use the finetuning feature to perform more complicated tasks, but it is also possible to extract sentence embeddings from BERT. There are two main approaches for this, either an average is taken from the combined word embeddings, or the context token of the entire sentence is taken. The context token is the same information the network uses to perform the context specific embeddings discussed earlier. But it has been shown that these sentence embeddings are worse than many much simpler approaches [34] for tasks like sentence similarity using cosine similarity. This is because the vector spaces are anisotropic [11], meaning

that most vectors are concentrated in a small space relative to the total available vector space. This causes the distance metric used in most embedding classification tasks to be nonsensical because all vectors are close to each other. Most applications of BERT (e.g. KG-BERT) solve this by using the fine-tuning feature, but this requires training time and BERT sentence comparisons are much more time consuming than cosine similarity (65 hours compared to 0.01 seconds [34]). It has been shown that this can be solved by performing post processing, for example normalizing the vector space so it becomes isotropic [21]. But because, for the goal of gaining graph embeddings, no cosine similarity will be performed on sentence embeddings, it does not need to be a problem. BERT for Link Prediction [7] shows that BERT sentence embeddings can be leveraged successfully in downstream tasks (that do not require cosine-similarity).

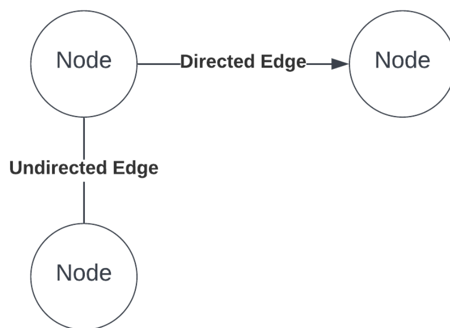

Figure 3: Example of a small graph, consisting of three nodes and two edges, of which one is directed and one is undirected.

2.4 Knowledge Graphs

Knowledge graphs are a concept that arose concurrent to the semantic web as a method to structure data on the web [45]. As the name suggests it uses a graph structure to store and represent data, in Figure 3 a graph is shown with three nodes and two edges, one undirected and one directed. A graph thus consists of a set of nodes that are connected to each other by edges, these nodes and edges can be labeled but the edges can also be given a direction (as in Figure 3) or given a weight (or all of the above). Figure 4 is a small example of a knowledge graph, the nodes can represent entities, so people, objects, or concepts and the edges represent the relationship between these entities. For example, 'Ada Lovelaces occupation is being a mathematician', in this example 'Ada Lovelace' is the subject, 'occupation' is the predicate, and 'mathematician' is the object, a subject, predicate, and object together are referred to as a triple. One of the advantages of knowledge graphs is that combined with universal identifiers it allows for easy combination of different knowledge bases [45]. If there is another knowledge graph about London, it could be combined with the one in Figure 4 to infer more facts about Ada Lovelace.

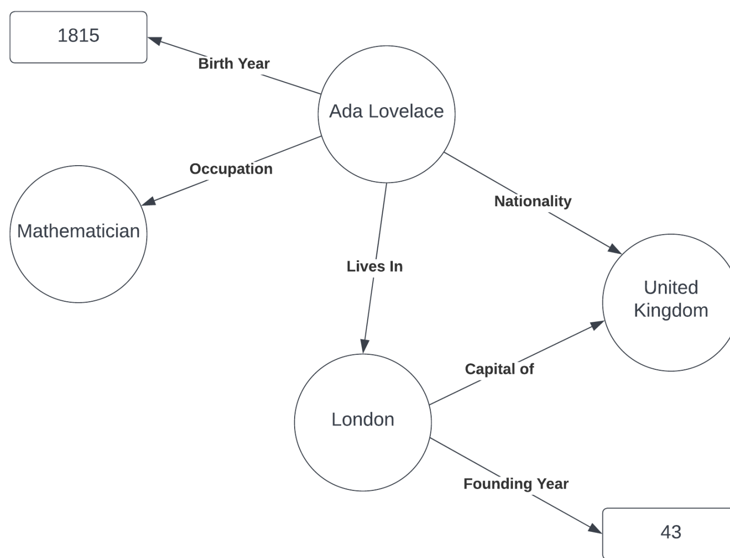

Figure 4: Example of a knowledge graph on Ada Lovelace.

3 Overview of Knowledge Graph Embedding Techniques

In this chapter an overview will be given on knowledge graph embedding techniques that are relevant to this thesis. Special attention will be given to what the specific technique is trying to solve and possible limitations of the technique.

3.1 TransE and Derivatives

An early and very effective knowledge graph embedding technique is TransE [4], it sought to create embeddings for each entity and predicate by minimizing the following equation: $error = vector(subject) + vector(predicate) - vector(object)$. Each entity and predicate is initialized by a random vector and after optimization the resulting vectors can be used for a number of tasks, such as entity or predicate classification, or triple completion. Triple completion works well as a consequence of the optimization function, by definition the sum of the subject vector and predicate vector should be roughly equal to the object vector. So when checking for what language 'Ada Lovelace' is likely to speak, the function $vector('AdaLovelace') + vector('Speaks') = x$ could be solved for the closest match. Overall this technique works very similar in practice to Word2vec discussed in the previous chapter. Although TransE is effective in what it set out to do, it has some limitations. Firstly, as demonstrated by the example of what language Ada Lovelace speaks, the technique works best with one to one relations. So if multiple people speak English or Ada Lovelace speaks multiple language the embeddings become ambiguous [41]. This has spurred many followup research based on TransE that use different mathematical operators to create and enhance entity embeddings [40], such as translating hyperplanes with TransH [41], or rotation with RotatE [35]. Another limitation is that it only uses direct relations, an embedding of an entity is only directly influenced by the direct neighbours it has. This can be limiting because sometimes second degree neighbours might be very important for the meaning of an entity. For example, it might not necessarily be important for the life of Ada Lovelace that she lived in London, but rather that she lived in a city where many other scientist lived as well, or in a city that housed many scientific institutions. Methodologies that take this into account are discussed in Section 3.3.

3.2 BERT Powered Approaches

3.2.1 KG-BERT

A completely different approach to perform triple completion or other classification tasks on knowledge graphs, is to ignore the graph structure completely and just use textual descriptions of the entities and predicates. This is what KG-BERT [42] set out to do. Using Wikipedia to enrich knowledge graphs with semantic descriptions of entities and predicates they finetuned a BERT model to recognize correct triples. This model could then be used, naturaly, to recognize correct triples, but also to find the best match for missing fields in a triple. For the triple ('London', 'Is A', ?), multiple possible candidate objects could be inserted and fed to the model to see which ones match best. Although extremely computationally intensive it outperformed RotatE in some triple completion

benchmarks. One interesting fact is that for their specific application of BERT the smaller BERT-Base model outperformed BERT-Large.

3.2.2 BERT for Link Prediction

BERT for Link Prediction [7] is another approach that like KG-BERT leverages the power of existing language models to improve knowledge graph embeddings. Instead of using the finetuning functionality of BERT like KG-BERT, they extract the sentence context token and use that as an entity embedding. They then use this embedding as the input for existing entity embedding techniques like TransE. Because the entity vectors are now fixed all that remains is to find a fitting predicate vector that satisfies the constrained $error = BERT(subject) + vector(predicate) - BERT(object)$ the best. This has two advantages above pure TransE, fist it allows for inductive embeddings, and secondly it enriches the embeddings with natural language context. The inductiveness follows from the fact that any arbitrary text can be embedded as a context token using BERT and thus can be used for link prediction.

3.3 Walk Based Approaches

Both TransE derived methods and the two BERT based methods either only take direct graph neighbourhood into account or not at all. This can be limiting as like stated before the second degree neighbourhood (so neighbours of neighbours) is sometimes very relevant for the meaning of an entity. Node2vec [13] is an embedding technology that creates node embeddings using a method inspired by Word2vec, and is capable of taking into account an arbitrarily large neighbourhood of a node. It does this by using the same Skipgram architecture as Word2vec and treating nodes as words in a sentence. Because graphs don't naturally lend themselves to a linear representation they extract random walks from the target graph. This works by picking a node and then walking over the graph, only selecting nodes that are neighboured with the previous node, until a certain length of path is reached. All nodes that were visited are stored in a list, and many of these lists together form the training data. Node2vec is very effective in creating node embeddings that lend themselves well to clustering sub communities, finding nodes with a similar structural role in a network, node classification, and link prediction. In classification it beats the then state of the art of Spectral Clustering [38], Deep Walk [30], and LINE [37]. Despite its effectiveness it has one major limitation when applied to knowledge graphs, it does not take into account what type of relation two nodes have. So in the example of Figure 4 the way Ada Lovelace and London relate to each other is treated the same as the way London and the United Kingdom relate to each other. This means that Node2vec only works for knowledge graphs with just one type of predicate.

3.3.1 Triple Embeddings

To enable graph algorithms like Node2vec to be applied on knowledge graphs some kind of transformation is needed to make the graph homogeneous (have only one type of edge). One possible transformation is using a line graph as Triple2vec does [12]. In a line graph the entire triple, so (subject, predicate, object), is a node, and the edges

represent the way these triples relate to each other. This is a homogeneous relation without direction or any other attributes except for a weight indicator. Triples are connected with an edge if they have a common entity. In Figure 5 a line graph representation of the knowledge graph from Figure 4 is shown. Weights between triples are determined by how similar the predicate of the two triples are, in Triple2vec this is determined by comparing how many common entities predicates have. Using this line graph representation Fionda & Pirro have used two graph embedding algorithms to gain triple embedding, first they used the Node2vec algorithm [12] and later a graph neural network [32]. Both approaches beat knowledge graph embedding techniques like RotatE in classification tasks but also (predicate agnostic) graph techniques like Deep Walk and Node2vec itself. Although the graph neural network approach outperforms the Node2vec approach, this thesis will focus on the Node2vec approach because it is more established and the proposed treatment of this thesis is based on the Skipgram architecture used by Node2vec. Despite its effectiveness one limitation of a triple embedding is that it is hard to make predictions on unseen triples. Triples that are not part of the initial training data do not have a triple embedding and thus no classification can be done on them, while for other methods (TransE or RotatE) as long as the individual parts of the triple are known (have been embedded) they can be combined to make predictions on the triple as a whole.

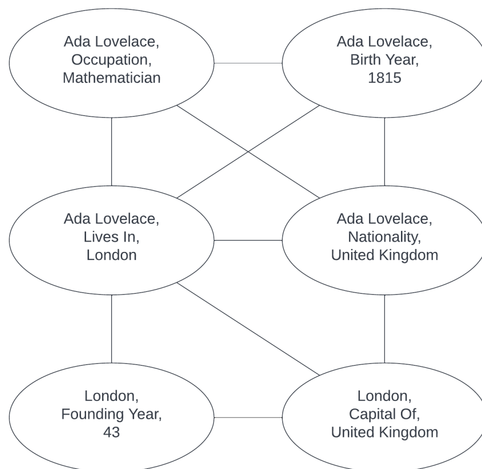

Figure 5: The knowledge graph from Figure 4 represented as a line graph.

3.3.2 Random Walks

The efficacy of Node2vec is heavily dependent on their choice of random walk generation, as it is the only thing differentiating them from Deep Walk [13]. The Node2vec random walk generation method is designed with the goal of controlling the balance between the two extreme options, shown in Figure 6. The blue walk starts at node C meanders through the network visiting every node just once and covering a large area

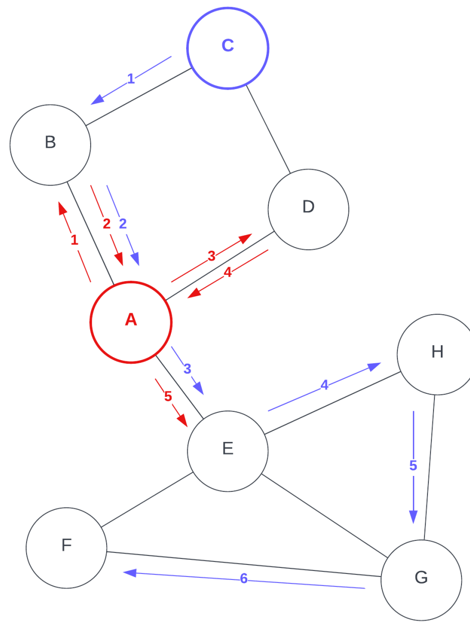

Figure 6: Both ends of the possibility space of generating random walks, red: breadth first and blue: depth first.

of the graph, the red walk starts at node A and only visits direct neighbors of A. Both types of walks provide a different kind of context, the red walk gives full information on the direct neighbourhood of node A nothing more, and the blue walk gives only partial information on the direct neighbourhood of node A but does give a broader context on the indirect neighbourhood of A. In the paper on Node2vec [13] they discuss the advantages of both approaches. The direct neighbourhood of a node provides a lot of information on that node, LINE [37] uses the direct neighbourhood in two ways: Firstly they look at what they call first-order proximity, which direct neighbours are strongly connected to a node, as this can indicate a similarity between the nodes. Secondly they look at second-order proximity, similarity of neighbourhoods between nodes, if two nodes are not directly connected but share a lot of common neighbours they might still be very similar. On the other hand RoIX [14] shows that looking at the broader context of a node, thus neighbours of neighbours, helps with recognizing nodes that have the same structural function in a network. For example, Ada Lovelace lives in London and speaks English, Alexander Von Humboldt lives in Berlin and speaks German, in a knowledge graph they would not have many common neighbours but they serve the same function in the graph, they are both people and scientists. Just looking at the direct neighbourhood might tell a lot on the similarity between Ada Lovelace and London but a broader context is needed to say something on the similarity between Ada Lovelace and Alexander Von Humboldt. Node2vec [13] balances these approaches by mixing fluently between them, it does this by using formulas 1 and 2. When performing a random walk the formula gives the chance of visiting a given neighbour depending on the distance to the previous node in the path. p_{vx} is the (relative) chance that the node will be selected given current node v and possible next node x, it is formed by a bias $a_{pq}(t,x)$ and an edge weight w_{vx}. The bias takes the previously visited node t and possible next node x, and can be controlled by parameters p and q. The bias depends on the distance between the previous node t and explored node x, this distance is either 0 meaning node t and x are the same, 1 meaning node t and x are direct neighbours, or 2 meaning node t and x are not connected in the graph. p determines the chance when the distance is 0 and q determines the chance when the distance is 2, a lower p makes the random walk more resemble the red walk from Figure 6 while a lower q makes it resemble the blue path more. To clarify, Figure 7 shows the bias for the neighbours of V after visiting T and V. These continues parameters make it possible to find a perfect balance between a more local focused path and a more meandering path. But it does mean that the effectiveness of the walks depend on the choice of parameter values, and thus a lot of time might need to be spend to determine these values.

$$p_{vx} = a_{pq}(t,x) * w_{vx} \tag{1}$$

$$a_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases} \tag{2}$$

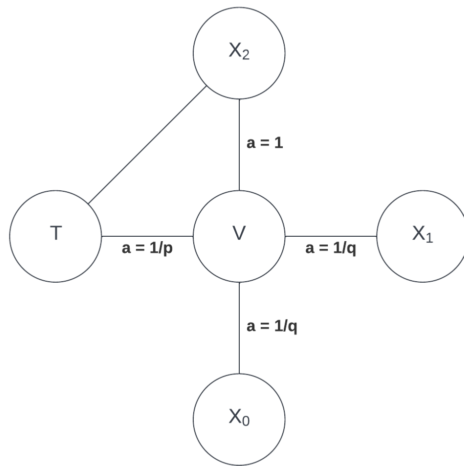

Figure 7: Node2vec guided walk chances, with V as current node and T as previous node.

4 Methodology: Training Pipeline

The methodology of the research is split in two parts, first this chapter will provide an in depth description of the TripLan2vec pipeline, the next chapter will concern its evaluation. This chapter will first give an overview of the entire TripLan2vec pipeline, then a section on the parameter evaluation benchmark, after which every individual process of the pipeline is explained.

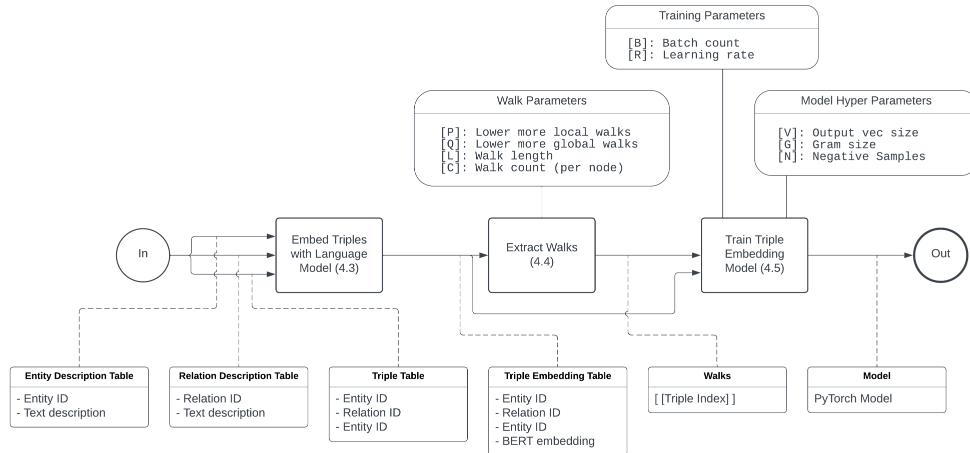

Figure 8: An overview of the TripLan2vec pipeline.

4.1 Pipeline Overview

In Figure 8 a schematic overview of the pipeline is given. At the core of the schema are the three main processes shown as rectangles, with the pipeline input and output shown as circles. The processes, input, and output are connected with arrows that indicate the flow of information, and each information flow is accompanied by a dotted line that provides documentation on what the data type of the flow is. The rounded rectangles connected to the final two processes provide information on the parameters and hyper parameters of the processes. The global input of the training pipeline consists of three files: (1) A file with all entities as a list of tuples of an entity id and a textual description of the entity. (2) A file with all predicates as a list of tuples of a predicate id and a textual description of the predicate. (3) Finally a file of the knowledge graph as a list of triples (head ID, predicate ID, tail ID). In cases where the target graph is stored in a different configuration, a prepossessing step is required to restructure the data accordingly. The three input files are then used by the language embedding process to create a file containing the triples plus an embedding. The walk extraction process uses this file to build a line graph and extract the walks. The walks are stored in a file as a list of lists of triple ids. The model training process uses both the triple plus embedding file and the walks to train a network that can map from language embedding space to triple embedding space. The final output

of the pipeline is the model trained by the last process, this model is stored as a PyTorch [47] model object.

4.2 Parameter Analysis

Parameter analysis is done by running a simplified training and evaluation pipeline. Training is simplified mostly by reducing the number of batches to one million, and is performed on the WordNet18 [8] dataset. With one million batches every node is visited 10 times as a positive sample and between 100 to 400 times as a negative sample, depending on the number of negative samples. Despite this low number of visits the resulting triple embedding is still descriptive enough for some prediction tasks. After the network is trained the first layer/mapping layer is taken to embed a subset of 50,000 triples from WordNet18. These triple embeddings are then used to train a neural network that is asked to predict the predicate of the input triple. The final score is then determined by how high a percentage of the triples could be correctly classified to their respective predicate. This method does not resemble a real world application, but it is a quick way to quantify how descriptive an embedding is. For each tested combination of parameter values the model is trained once and evaluated 20 times. Then for each parameter the value with the highest median is taken as best performing. For numerical parameters such as triple embedding vector size or walk guiding parameters p and q, it is not strictly needed to test if the results are statistically relevant. This is because if the best and second best value are not statistically distinguishable, it does no harm to pick the one with the highest median performance. For each parameter the benchmark results are presented in a table with for every value the first, second, and third quartile. Quartiles are the values that separate the data when split into four equal parts. The first quartile is the value for which 25% of the data is smaller, the second quartile splits the data exactly in half (equal to the median), the third quartile is the value for which 75% of the values are smaller, the zeroth quartile is the lowest value, and the fourth quartile is the highest value. Quartiles are chosen because it is a good way to compare data without being biased by outliers.

4.3 Triple Description to Language Embedding

Because the language embedding step is relatively straight forward and has already been elaborated upon in Section 2.3 and Section 2.3.2 it will be described concisely. The triple description to language embedding step of the pipeline works on any knowledge graph that is accompanied by natural language descriptions of all entities and all predicates. Once such a graph is provided, a triple description can be generated by concatenating the descriptions of the individual parts of the triple to form one sentence. This sentence can then be used as input for any sentence embedding tool. In this specific implementation BERT is chosen. To gain a sentence embedding from BERT the final layer of the model is summed resulting in one vector with 768 elements. The final result of this process is a list of triples in tuple form: head element ID, predicate ID, tail element ID, and an embedding. The index of each triple is used as triple ID for further processes.

4.4 Walk Extraction

Walk extraction consists of three steps, first the knowledge graph is transformed to a line graph, then edge weights are calculated between the triples/nodes, and finally the walks are extracted from the line graph.

4.4.1 Line graph Transformation

In Section 3.3.1 an explanation and motivation is provided for what a line graph is and why it is useful, the actual transformation from knowledge graph to line graph is relatively simple. The list of triples generated by the language embedding step is used as input, then for each triple an entry is made in a dictionary, this dictionary stores for each triple all triples that have at least one entity in common. Initially this is just stored as a list, but in the next step weight values are added to the triples. To speed up walk extraction the triples are split into three distinct lists, one list for which at least one entity matches the head entity of the key triple, one list for which at least one entity matches the tail entity of the key triple, and a list for which each triple matches both the head an tail entity of the key triple. The grounds for splitting the triples in this way will be explained in Section 4.4.4.

4.4.2 Determining the Edge Weights

Weights are added to the line graph with the same method as Triple2vec [12] which is based on predicate relatedness. The more related two predicates of neighbouring triples are the higher the weight should be. This is achieved by constructing a matrix with a column and a row for every predicate, for each combination of predicates p_i and p_j the matrix value is described in Formula 3. Number of entities in common is shorthand for number of triples with predicates p_i and p_j that have entities in common, ITF is defined in Formula 4 with the same shorthand. Now every row of this matrix is a vector representing a predicate, each value of the vector is a similarity indication for all available predicates. To get the final weights for edges between predicate p_i and p_j the cosine similarity of the respective rows of the matrix (row i and j) is taken, the result of which is the final weight value $w_{i,j}$. All these calculations are done once and the resulting weights are stored in a dictionary and added to the line graph.

$$Matrix\ Value = log(1 + number\ of\ common\ entities\ p_i\ and\ p_j) \times ITF \quad (3)$$

$$ITF = log \frac{number\ of\ entities}{times\ p_i\ and\ p_j\ have\ enties\ in\ common} \quad (4)$$

4.4.3 The complexity of Node2vec walk extraction

Now that a line graph is constructed and weights have been added to the graph, the walks can be extracted. The original plan was to use the approach used by Node2vec [13]. This approach allows for guiding the walk characteristics and has been shown to improve performance. Guiding is done by biasing the edge weights (added in the previous

step) depending on inter connectivity between possible next nodes and the previous node in a walk. One major downside of the approach is that each step requires that for all possible next nodes (neighbours of the current node in the walk) the biased weights need to be calculated. The complexity for this is $O(p \times l \times b)$, where p is the number of paths, l is the length of the paths, and b is the average number of neighbours for each node, also known as branch rate. Because the number of paths is often determined by the number of nodes in a graph the complexity can also be rewritten as $O(f \times n \times b)$, where f is a factor determined by multiplying the number of paths per node and path length, n is the number of nodes, and b is the branch rate as before. When applying the algorithm on a larger graph the process becomes more expensive because there are more nodes, but in many cases a larger graph also has a higher branch rate making the process even more expensive. An intuitive example of this is a social network. A reasonable assumption for a network mapping all acquaintance relations in a single street contains around a 100 people and around 5 connections per person. If this network is scaled up to map all acquaintance relations in a country the number of people grows, but also the number of connections per person, because generally not all friendships occur in the same street. The fact that the size of the graph is multiplied by the branch rate causes that the complexity of the algorithm can scale anywhere between linear or cubic depending on the graph characteristics.
One way to mitigate this, could be to cache the biased weights for each neighbour, this way the biases have to be calculated only once. Now all that needs to be done is to pick a random neighbour based on the pre-calculated biases, this can be done with a divide and conquer algorithm in $O(log(b))$ where b is the branch rate again. With a branch rate of 500, an algorithm with a logarithmic complexity would be more then 55 times faster $(500/log_2(500))$. The challenge of caching is the storage size, because the pick chance of a neighbour depends on its distance to the previous node in the walk, multiple biased weights need to be stored per neighbour. In fact each neighbour of a node can be a previous node in a walk, so the complexity of the storage size for the full cache is $O(n \times b \times b)$. A naive implementation with caching in Python resulted in a memory requirement of more than 60GB when generating walks for the WordNet18 [8] dataset. Because it scales quadratic on branch rate it is not hard to imagine a graph for which the cache size will defy the working memory capacity of any available computer. When developing Triple2vec, Fionda & Pirro deal with these speed and memory challenges with a low level code implementation and executing it on video cards instead of CPU [12].

4.4.4 FastWalk: Extracting Walks from a Line Graph

Because in this case walks will be extracted from line graphs instead of more general graphs, an alternative walk extraction algorithm can be used that leverages a property of line graphs. This algorithm does not have the exact same behaviour as the one described by Node2vec [13], but it is a close approximation that has the same speed complexity of the cached variant without using any cache. This property of line graphs has not been leveraged before to generate graph walks to the best of my knowledge. In the base case in Figure 9 it is again demonstrated what the transformation from graph to line graph looks like. On the left side two nodes are shown, alpha and beta, connected with a labeled edge C. Both alpha and beta have other edges connected to them, $\{X0, P, X1\}$ for alpha and

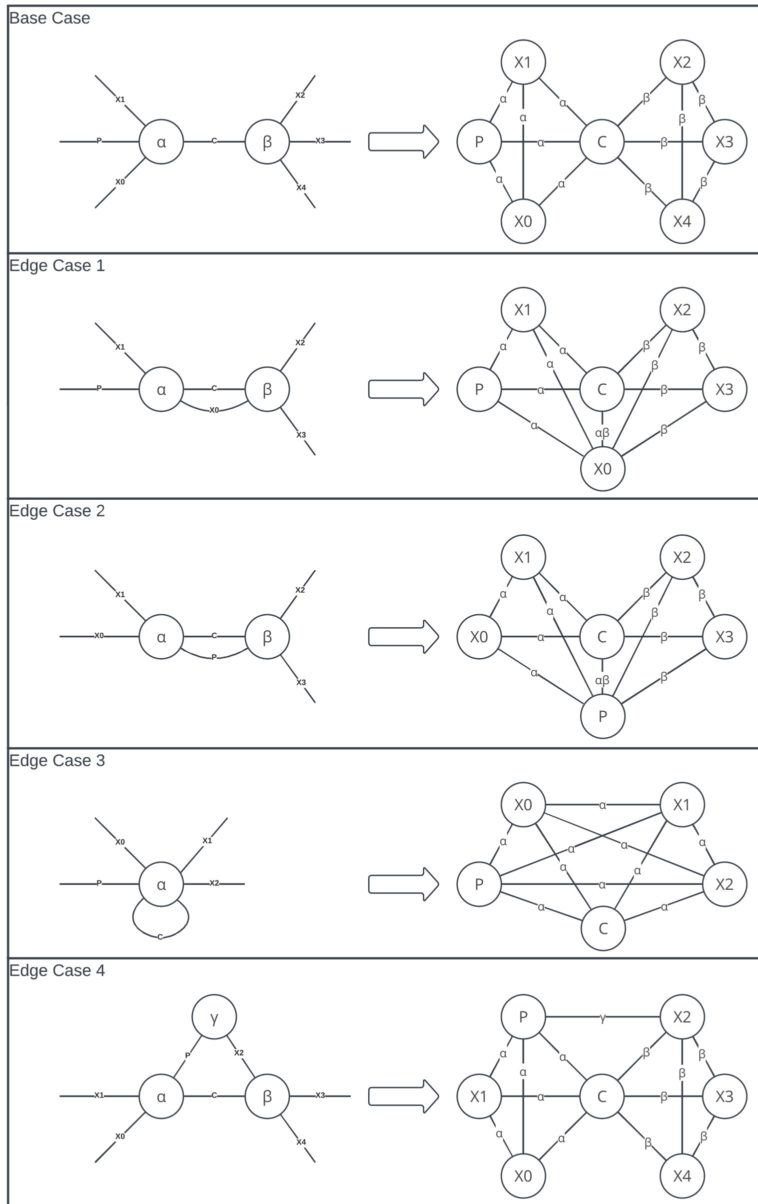

Figure 9: Examples for each of the discussed cases for the FastWalk algorithm. For each example the knowledge graph representation is shown on the left, and the line graph representation is shown on the right. The previous triple in the walk is annotated with a P and the current triple of the walk is annotated with a C. The alpha entity is always the entity connecting triples C and P, the beta entity is the other entity of triple C (if C has two distinct entities).

$\{X2, X3, X4\}$ for beta. On the right this graph has been transformed to a line graph, edge C has become a node and is now directly connected to all edges connected to alpha or beta. All previous edges of alpha are also themselves interconnected because they all share a common neighbour (alpha) the same is true for beta. In a normative case every node in a line graph is connected to two clusters that are internally highly connected but between each other loosely connected. This is the property that can be leveraged by a walking algorithm to improve performance. Formula 5 shows again the three options for guiding bias a: (1) The distance between the previous node and the candidate node is 0, meaning they are the same node. (2) The distance between the previous node and the candidate node is 1, meaning they are directly connected to each other. (3) The distance between the previous node and the candidate node is 2, meaning the only thing connecting the two nodes is the current node in the walk.

$$a_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases} \tag{5}$$

When exploiting the properties of a line graph to speed up node selection, it is important that no single step in the algorithm requires that all neighbouring nodes are checked or compared, as this would make the algorithm exactly as slow as the original one. The maximum complexity of an operation is $O(log(b))$ as this is the complexity of randomly selecting a weighted element from a list. In the following section it is explained how this is achieved for the base case as well as all relevant edge cases discovered by the author. And for the one edge case that could not be solved within the required complexity limits it is argued why this is not possible.

4.4.4.1 Base Case

In the base case of Figure 9 the current node (in the line graph) is annotated with C and the previous node with P, the other candidate nodes are annotated with X and a number. Dealing with the first option is easy, if the previous node is equal to the candidate node the distance is zero. Checking this can be done with one simple operation of $O(1)$. Dealing with the last option is also easy, because the previous node and the current node have entity alpha in common, all neighbours that have entity beta as common entities are by definition distance 2. Checking this in constant time is possible if the line graph is stored so that all neighbours of a node are available in two lists: alpha group and beta group. The alpha group is the set of neighbouring triples that all contain entity alpha, by convention this is always the entity that the current and previous node have in common. The beta group is the set of neighbouring triples that all contain entity beta, this is the entity that the current and previous node do not have in common. To determine the beta group, the common entity between between the previous and current triple needs to be determined which can be done in $O(1)$ time complexity. For the second option (distance of one) the same check as before can be made but this time the alpha group will be selected, but this is not sufficient. This group not only contains all nodes with distance one, it also contains the previous node with distance zero (P). This can be solved by picking two distinct random nodes from the alpha group and only picking the second

node if the first one happens to be equal to the previous node P, this process is twice as slow as picking one node but is still $O(log(b))$. The last step is to determine from which of the three options a node is picked. This is done by calculating the total chance of picking a node from each option, and then picking one at random based on these chances. For distance zero this chance is equal to the weight of the edge between the previous node and the current node times $1/p$. For distance one this is the sum of all weights of the edges between the current node and the nodes from group alpha, minus the weight between the previous node and the current node. For distance two this chance is equal to the sum of all weights of the edges between the current node and the nodes from group beta times $1/q$. These biased weight sums can be calculated when constructing the graph so they don't add complexity. The complete process looks like this: First determine the pick chances of each distance option, and pick one based on these chances. If option one is chosen simply return the previous node. If option two is chosen pick a random node from group alpha (that is not the previous node). If option three is chosen pick a random node from group beta. This process has the exact same behaviour as the slower variant, but it only works if the alpha and beta groups are completely distinct. Unfortunately there are multiple edge cases where this assumption does not hold, some of these can be dealt with in $O(log(b))$ time complexity, but one can not and requires $O(b)$.

4.4.4.2 Edge Case 1

The first case is when there is a parallel edge $X0$ that connects the head and tail entity of the current node together as well. This means that node $X0$ is both counted in the alpha group and beta group when constructing the graph, making it count twice and thus making the chance it gets randomly picked erroneously higher. This can be solved by introducing a third group when constructing the graph, the parallel group. This group contains all nodes that are formed by a parallel triple to the source/current node. Because these parallel nodes are connected to all nodes in group alpha en beta, they should always be counted as distance one. When calculating the pick chance of option two (distance one) the sum of weights from group alpha and group parallel need to be combined. If option two is selected a second weighted decision needs to be made whether to pick from group alpha or group parallel.

4.4.4.3 Edge Case 2

The second edge case is similar to Case 1 but this time the previous node is a node from the parallel group, now all nodes are distance one (except the previous node itself). This is solved by treating the alpha, beta, and parallel group as all being distance one and setting the biases accordingly.

4.4.4.4 Edge Case 3

In edge case 3 the head and tail of the triple that forms the current node are the same. This means that there is just one group of nodes that are all neighbours of each other. This is a problem when constructing the graph because all nodes should be part of the alpha and beta group, making them all count twice. This is solved by only having an alpha group, and treating the beta group as empty.

4.4.4.5 Edge Case 4

Edge case 4 occurs if three triples form a triangle, when in such a situation both the previous and current node are part of that triangle the third node of the triangle will be classified incorrectly. In Figure 9 Edge Case 4 this is illustrated, a triangle in the knowledge graph is also a triangle in the line graph. This triangle means that there is a connection between group alpha and beta, meaning that not all nodes from group alpha are distance two removed from all nodes of group beta. There are multiple strategies to deal with this and none of them are possible within $O(log(b))$ time complexity. In an unlikely situation where it is possible to determine which nodes from group beta are also neighbouring node P within the time constraint, it is still not possible to assign the correct biases to these nodes within the time constraint. To assign the bias correctly all outliers need to be removed from the beta group, because they are not part of the set of nodes with distance two form the previous node. If the number of outlier nodes is constant this could be done within $O(c \times log(b))$ time complexity, where c is the number of outliers. But in reality the number of outliers is not constant, in a worse case it could be proportional to the number of nodes within group beta. This would mean that the time complexity is $O(b \times log(b))$ slower than the original algorithm which is $O(b)$. If the outlier groups are cached separately form the alpha, beta, and parallel groups, this discrimination can still not be done within the time constraint. This is because there is not a constant number of outlier types, there could be multiple entities that form a triangle with entities alpha and beta, and for each entity a separate outlier group is needed. The number of groups could again be, in a worse case, proportional to the branch rate, resulting in a time complexity of $O(b)$ for determining what outlier group is distance one and what outlier group is distance two. So with or without caching, solving edge case 4 is not possible in a faster way than the original algorithm.

4.4.5 Variable Speed vs Variable Accuracy

Edge case 4 means that it is impossible to exploit the properties of a line graph in a way that has the exact same behaviour as the Node2vec [13] algorithm but with a lower time complexity. But the line graph properties can still be used to make two different types of algorithms that have advantages and disadvantages. The first type of algorithm handles all edge cases regardless of the time complexity and is thus in behaviour exactly the same as the Node2vec algorithm. Edge case 4 means that the time complexity is variable depending on the number of triangles in the graph, closer to $O(f \times n \times log(b))$ if there are fewer triangles, closer to $O(f \times n \times b)$ if there are more triangles. In many cases this could still be a great improvement over the original algorithm. The second type of algorithm handles all edge cases that can be solved within time complexity $O(log(b))$ and accepts that it has a slight error compared to the Node2vec algorithm. This error is again variable depending on the number of triangles in the graph, lower with less triangles, higher with more triangles. Because the walk extraction is a stochastic process meant to capture certain characteristics from a graph, a slight inaccuracy in the guiding of a walk does not mean the final resulting walks are incorrectly capturing the graph, they just capture it in a different way. This raises the question what the effects on the final triple embeddings are, depending on if the walks are generated using Node2vec, or the FastWalk algorithm.

4.4.6 Walk Parameters

There are four important parameters to generate walks. The walk characteristics can be guided with p and q, the exploration rate can be determined with walk length and walk count. For this thesis walk length and walk count are taken from Triple2vec [12], they used a walk length of 100, and a walk count of 10. This means that per node in the graph 10 walks are started of length 100. p And q are graph specific, so some trial and error is required to determine the most optimal values. The results of the benchmark are shown in Table 1 (for an explanation on how to interpret the results see Section 4.2), based on these results the values $p = 2$ and $q = 1$ are chosen for the WordNet [8] data set.

Table 1: Values for p and q.

Quartile	p=1 q=1	p=0.5 q=1	p=2 q=1	p=1 q=0.5	p=1 q=2	p=2 q=0.5	p=1.5 q=0.8
Q1	49.2%	48.5%	52.6%	**54.2%**	48.8%	45.8%	48.6%
Q2	51.4%	52.3%	**55.3%**	54.8%	51.3%	47.8%	51.2%
Q3	53.6%	54.3%	**56.9%**	56.5%	53.2%	49.7%	52.1%

4.5 Mapping from BERT Sentence Embeddings to Triple Embeddings

After the random walks are extracted, the walks are combined with the language vectors generated in the first step of the training pipeline. These two datasets are used to train a neural network that can map sentence vectors into triple vectors.

4.5.1 Software and Hardware used

The network is programmed in the Python programming language because this is widely used in the machine-learning field, so there is a lot of high quality documentation available. A second reason is that the PyTorch library is programmed in Python, and PyTorch is the current go-to library for working with neural networks. One advantage of PyTorch over competitors such as TensorFlow is that it is designed in a imperative way, making it easy for programmers to add custom code to a network. This customizability allows for more advanced network architectures, which is exactly what is needed for this thesis. For the hardware the code is mostly run on an Intel i5 computer with 16GB of RAM, longer simulations are run on Google Colab because of problems with the PyTorch library on mac. All simulations are run on CPU instead of GPU because the extensive swapping of the final network layer required a lot of IO between the CPU and GPU making CPU perform slightly better.

4.5.2 Network Architecture

The network consists of two parts, first a single layer network that does the actual transformation, then a single layer network that performs an auxiliary task based on the output of the first layer. The auxiliary task is used to update the weights of the

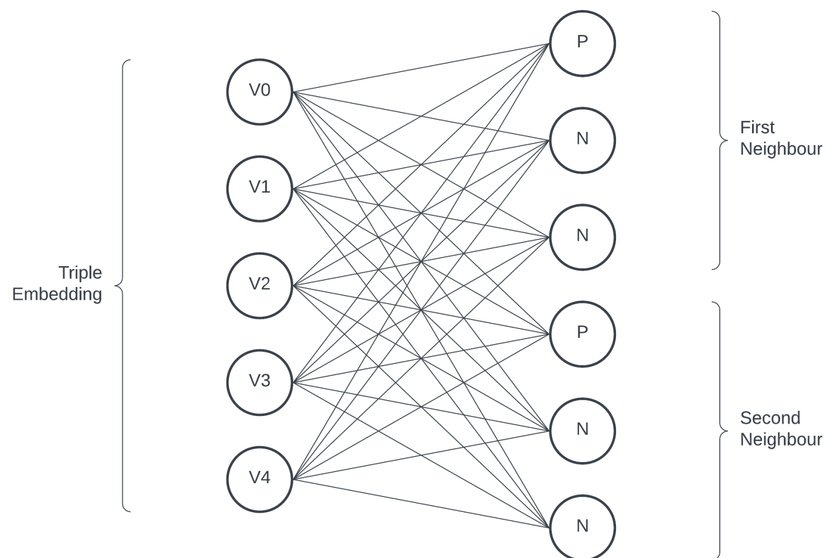

Figure 10: A visualisation of the final layer of the neural network, it depicts the weights connecting the triple embedding $[V0 - V4]$ to the individual neighbour prediction nodes. The nodes called P are the positive examples (actual neighbours of the input triple), the nodes called N are negative samples (false neighbours).

transformation layer and should thus be chosen in such a way that it best represents the real world task the network should perform. In this case the goal is to map from language embedding space to triple embedding space, and it has been shown by Triple2vec [12] that predicting neighbouring nodes in a random walk is a good auxiliary task to gain triple embeddings. As discussed before it is inefficient to make a prediction on each and every node on whether it is a likely neighbour, so instead negative sampling is used. In Figure 10 it is shown what this looks like as a concrete neural network, note that only the layer performing the auxiliary task is shown. The left set of neurons or input neurons is the triple embedding gained from the mapping layer, the right side or output is a collection of neurons that use the input triple embedding to predict whether the input triple is a likely neighbour to it. In this example there are six neurons, three to predict the direct neighbour of the input node, three to predict the neighbour once removed. For each set of three there are two negative samples, and one correct sample, the desired output of this network is thus $[1, 0, 0, 1, 0, 0]$. After the network did its prediction and the error rate compared to the correct output is used to update the weights, a new set of positive and negative samples will be used. This means that the final layer of the network needs to be swapped with a new layer representing these positive and negative samples. This is an expensive process that slows the overall training down. A trick to mitigate this is to take a set of positive samples and let them be each others negative samples. Table 2 shows an example batch for the network from Figure 10. In the left column the input node is shown, the other columns depicts the expected output, 'A n+1' and 'A n+2' indicate the first and second 'true' neighbour of node A. Because the negative samples of each input case are themselves positive samples of other input cases, the entire batch can be executed without an expensive layer swap in between. If the number of negative samples is increased this process is even more efficient.

Table 2: Training batch

In	A n+1	B n+1	C n+1	A n+2	B n+2	C n+2
A	1	0	0	1	0	0
B	0	1	0	0	1	0
C	0	0	1	0	0	1

4.5.3 Network Hyper Parameters

The network has many parameters that have a big impact on performance, in an ideal situation all combination of many possible values for each parameter is tested against the final benchmark on a validation dataset. But this is an expensive process proportional to $runtime \times |parameter\ values|^{|parameters|}$, especially when taking into account the stochastic nature of training a neural network, meaning that every test needs to be run multiple times. Instead a dummy benchmark is used as explained in the introduction of this chapter, and the parameters are tested individually. In this section a short explanation is provided for each parameter, as well as the considerations for choosing a value. When needed the parameter evaluation benchmark is used to determine the optimal value, but in some cases the network does not produce usable vectors so no further testing is required.

4.5.3.1 Activation Function

The activation function of a layer is applied to the aggregated sum of the input nodes, this is used to determine at what value a neuron returns a high value or a low value. There are multiple functions available all with there own characteristic. But in this case evaluating the different activation functions is an easy task because only the ELU ($a = 1$) [20] function leads to a converged network. Every other activation function that is tested (Sigmoid, RELU, Softsignh, Arctan, Leakyrelu) [20] results in a network that produces triple embeddings that only consist of zeros.

4.5.3.2 Loss Function

After the network produces its prediction a function needs to be applied to compare the difference between the prediction and the desired result. As stated before when explaining the Softmax function, it is wasteful to train a network to exactly match the expected output data of zeros and ones, when giving high and low values also suffices. With desired result $R : [1, 0, 0]$ and predictions $A : [761, 23, 42]$ and $B : [0, 0, 1]$ a simple distance metric would say that B is more correct but a simple distance metric after a Softmax layer would say A is more correct. Another approach (that in reality is very similar) is using cross entropy loss, in this case a distance measure between two chance distributions (the prediction and the desired result) is used to score the result. Part of the build in cross entropy loss function in PyTorch is a Softmax layer, so in reality it is the same as applying a Softmax layer but with a smarter loss function. In practice this smarter loss function is also more effective, where Softmax combined with sum of squares sometimes resulted in a non converging network, cross entropy loss always led to a converging network. One complexity is that in this network multiple distinct predictions are made, namely one prediction per neighbour, so the cross entropy loss function is applied to each neighbour separately and the results are then summed.

4.5.3.3 Vector Size

The vector size is the size of the final triple embedding. The smaller the easier it is to extract information from the vector, the larger the more data can be stored in the vector. To choose the right vector size, multiple values are evaluated in a range from 16 to 256 doubling in size between each step. In Table 3 the results of this are shown (for an explanation on how to interpret the results see Section 4.2), it is clear that a vector size of 128 is optimal, this is also the size of the vectors Triple2vec uses [12].

Table 3: Vector Size of triple embedding.

Quartile	16	32	64	128	256
Q1	31.3%	35.2%	42.2%	**55.2%**	3.4%
Q2	32.5%	36.3%	42.9%	**56.0%**	18.3%
Q3	33.4%	37.1%	45.8%	**57.3%**	23.4%

4.5.3.4 Learning Rate

The learning rate determines the size of the step with which the weights of the network are changed after each batch. If the learning rate is too low it takes longer to train the

network, if the learning rate is too high there is a risk that the optimal value is never found because the network over-corrects overstepping the optimal value. To choose the right learning rate the following values are evaluated $[1e-1, 1e-2, 1e-3, 1e-4, 1e-5]$. The lower values resulted in a network that only returned zeros as triple embedding, and values $[1e-1, 1e-2]$ gave a converging network. A value of $1e-2$ is chosen because it is the lowest functioning learning rate, and a lower learning rate means a higher chance of finding the optimal solution.

4.5.3.5 Number of Negative Samples

The number of negative samples determines how many predictions the network needs to make per neighbour. In Figure 10 two negative samples are used so the network needs to make three predictions. According to the Word2vec paper [25] the number of samples depends on the corpus size, around 5 when the corpus is really big and around 10-20 when it is smaller. Because the datasets used in this case are less big, between 10 and 20 is preferred. Normally the less negative samples the quicker the training process, because less operations need to be performed. Because of the bulking of negative samples there are in this case also speed advantages to having more negative samples, more negative samples means that the expensive swapping action needs to be performed less. On the benchmark the values $[9, 19, 39]$ are tested, note that together with the one positive sample they add up to a batch of $[10, 20, 40]$ respectively. From Table 4 it is clear that the less negative samples the better the performance. But diminishing returns in score between 9 and 19 mean that 19 is chosen as a sweet spot between performance and speed.

Table 4: Number of negative samples.

Quartile	9	19	39
Q1	**32.3%**	31.3%	27.0%
Q2	**33.2%**	32.5%	28.8%
Q3	**34.3%**	33.5%	30.5%

4.5.3.6 Size of the Gram

The size of the predicted gram is the number of neighbours for which predictions are made, in Figure 9 the gram size is 2. The bigger the predicted gram the more context is used to shape the triple embedding, but nodes that are further away from the input node are also less predictable. When making the gram too big training energy is wasted on trying to find structure in something unstructured. Finding the perfect gram size is done by starting with size two and increasing it by 1 each step, until performance starts to decrease. From Table 5 it is clear that performance increased the bigger the gram up until size 3.

Table 5: Number of neighbours predicted.

Quartile	2	3	4
Q1	27.8%	**31.3%**	29.8%
Q2	27.9%	**32.5%**	30.7%
Q3	30.1%	**33.4%**	31.7%

4.5.4 Training the Network

Once the parameters are determined a training run of 10 million batches is done to train the model. After the training is completed the final layer of the model is discarded, and the first layer of the model is kept as this is the layer that performs the mapping from language embedding space to triple embedding space. This model can then be used in combination with the pre-trained language embedding model to generate inductive triple embeddings.

Table 6: Overview of all chosen parameters.

Parameter	Value
Walk Count	10
Walk Length	100
p	2
q	1
Activation Function	ELU
Loss Function	Cross-entropy loss
Embedding Vector Size	128
Number of Negative Samples	19
Number of Predicted Neighbours	3
Learning Rate	1e-2

5 Methodology: Evaluation

This chapter will give an overview of the selection process for an evaluation method. It will discuss the evaluation methods from the two most important papers this thesis is base on, Triple2vec [12], and KG-BERT [42]. It will discuss what evaluation method will be chosen and why it suits this thesis. Besides the evaluation of the entire pipeline, some time will be spend on the evaluation of the walking algorithm and negative sampling batching.

5.1 Evaluation Requirements

When selecting an evaluation method it is important to focus on two questions, namely: Can the evaluation method be used to compare different approaches? Does the evaluation method suit the technique?
Comparability is an obvious requirement as it allows for the model to be placed in context with other techniques. And gives an objective way to judge if the approach is useful. Optimal comparability is reached if the exact same task is performed on the same dataset with the same split between training data and test data. Here training data and test data refers to what data is used to train the model (training data) and what data is used to test the performance of the model (test data), this separation is done to check how well the model performs on unseen data. The fact that some of the data is unseen during training has some implication on whether transductive or inductive models can perform the required task. There are two ways to achieve the parity between task, dataset, and data split: take an existing benchmark that has been applied before, or create a new benchmark and apply it to existing state of the art techniques. In this thesis the first option is preferred, because it can be assumed that existing benchmarks are broadly sensible as they have been used before, and it reduces the work of applying the benchmark on competitors.
To check if a benchmark suits TripLan2vec it is first important to get an overview of the capabilities and requirements of the model. The first obvious requirement is textual descriptions for the triples, or at minimum textual descriptions of the individual entities and predicates to construct triple descriptions. Without descriptions there is no input for the language model and thus no input for the triple embeddings. Another requirement is that the task should be performed on entire triples as it is not possible to break up a triple embedding into entity and predicate embeddings, so no predictions can be made on them. To show the inductive capabilities of the model, the evaluation should demonstrate this inductiveness on triples. Meaning that the test data should contain triples that are not part of the training data, and ideally also entities and predicates that are not part of the training data.
As stated earlier, an existing evaluation benchmark is preferred and because this thesis builds mostly on Triple2vec [12] and KG-BERT [42] their evaluation methods will be explored first.

5.2 Evaluation Benchmarks

5.2.1 Triple Labeling

Triple2vec [12] is validated with triple labeling. They used three datasets: DBLP [15] a graph connecting authors to papers, topics and venues. Foursquare [16] a graph of interesting places, connected to users, points of interest, and timestamps. A subset of YAGO [15] on movies, actors, and directors. For each graph there is a node type that is labeled, this label is then propagated to neighbouring nodes in the knowledge graph via relevant predicates. In the YAGO graph this is done by propagating five broad genre labels from movies to connected actors, directors, and musicians. When the graph is transformed to a line graph, the union of the labels from the head and tail entity of the triple is taken to determine the triple labels. This level of indirection results in triples that don't contain the original movie still being associated with that movie. To test how well the triple embeddings can be used to predict these labels, the entire graph is first used to generate the triple embeddings, then a separate model is trained to predict the labels. In this case the training data contains the entire graph, but not all triples are labeled, some triple-label pairs are withheld to form the test data.
The power of this evaluation method is that it shows the extend to which the triple embeddings are capable of encapsulating the graph structure; triples not originally associated with labels can still be labeled based on proximity. Because triple embeddings have not been used before they had to create a new benchmark that is not well documented and for which the datasets (training data and test data split) are not publicly available. This violates the first requirement as it can not be guaranteed that that fluctuations in scores between methods are due to better performance or to differences in the training data and test data. Another problem with this evaluation benchmark is that the datasets are not accompanied by textual descriptions. Because most DBpedia entries contain a YAGO-ID it is possible to source textual descriptions from DBpedia for many entities. But ultimately it is not possible to create a satisfying dataset, because it is not known what the exact subset from the YAGO dataset is, and a test run of a similar subset of YAGO showed that for only 70% of the entities there is a textual description available on DBPedia. Even if the exact dataset is available the data loss due to missing textual descriptions is too high to get a satisfying comparison.

5.2.2 Recommendation Engine

Besides triple labeling Triple2vec [12] is also used to performed a downstream task as evaluation. In their case the MovieLens and KKBox datasets are used to train and test a recommendation engine [12], and they demonstrate that triple embeddings outperform some state of the art methods. Both datasets are graphs of users interacting with either movies or other items, interactions such as buying an item or saving an item to a wish list. The datasets are available, and the task is well documented, the downside arrives with the lack of natural language descriptions. The datasets originally come without descriptions, in the case of movies descriptions could be sourced from IMBD or DBpedia, but generating or sourcing natural language descriptions for users is near impossible. Even if a database with natural language descriptions of individuals existed, it would be highly confidential and/or ethically questionable. So using a recommendation engine as

evaluation task is not viable on the grounds that there exists no suitable datasets.

5.2.3 Triple Classification

Triple classification is the task of determining whether a given triple is true or false, and is the main task by which KG-BERT is evaluated [42]. To perform triple classification a graph is transformed into a list individual triples, this list is then split into training data and test data. Then a set of false triples is generated which is again separated into training data and test data. For the sake of consistency it is important that the set of false triples in the test data is constant between methods. For the training data variation is not a problem, as novel techniques for false triple generation are a valid way to improve performance. Once training data and test data is available, a classifier can be trained on the training data, and then be applied to the test data to determine the accuracy rate. Because KG-BERT also requires textual descriptions for their method they have selected existing datasets containing natural language descriptions for the entities and relations, the WordNet11 [26] dataset and a subset of the FreeBase [3] dataset (FreeBase13). WordNet is a graph of English words connected with semantic relations such as 'synonym' or 'antonym'. FreeBase is a graph of human knowledge and the predecessor of Wikidata, it contains facts in triple representation. The advantage of selecting triple classification on WordNet and FreeBase is that textual descriptions are available, standardised training data and test data is available, and the task is applied on entire triples. One disadvantage is that it does not depend heavily on the graph structure of the data, every triple can be considered as its own entity. But overall it is a good benchmark to compare the TripLan2vec with KG-BERT and other methods that have performed this benchmark before. An extra bonus is that for the WordNet11 dataset inductive subsets are available, meaning training data that contain only part of full dataset. Inductive subsets of 5%, 10%, 15%, and 20% are available accompanied by triple classification results from other methods.

5.2.4 Link Prediction and Relation Prediction

Two other evaluation methods used by KG-BERT are link prediction and relation prediction [42]. With link prediction the task is, given an entity and a predicate what entity is a likely match. With relation prediction the task is similar but now a predicate should be predicted given a set of entities. Both of these tasks are a better fit with methods that treat entities and predicates as individual units. TransE for example can give a best matching entity embedding based on an input entity and predicate embedding [4]. This best matching 'hypothetical' embedding can then be used to find the closest matching existing entity. A similar technique is possible for finding the best fitting predicate. A system that considers triples as a whole, such as KG-BERT or TripLan2vec, can still be used to perform these tasks but less efficient and possibly less successful. KG-BERT performs relation prediction by using their model trained for triple classification, then to select the best fitting predicate they score each possible combination of the two input entities and all available predicates and take the best scoring triple. If a graph contains just 10 predicates this is doable, but in the case of link prediction this becomes inhibitively expensive. WordNet11 contains about 40.000 individual entities, to select for the best fitting entity 40.000 classifications need to be performed that all need

to be scored, moreover it has been shown that applying BERT is astronomically slower than cosine similarity vector comparisons [34]. These benchmarks can still be used but more in a theoretical sense, because it is unlikely that such a computationally expensive method will be used for these tasks.

5.2.5 Conclusion

If time requirements are not an issue the preferred benchmark would be use the triple labeling benchmark from Triple2vec [12], because it plays into the strength of triple embeddings that capture graph structure. But a lack of suitable datasets would require the creation of custom datasets, that then require the benchmarks to be rerun on baseline methods to create a fare comparison. So triple classification (used by KG-BERT [42]) is chosen as the main benchmark, because the task is performed on entire triples, the datasets are already available with text descriptions, and the evaluation has been performed before, inductively and transductively.

5.3 Evaluation of FastWalk and Negative Sample Batching

Evaluating the efficacy of the produced triple embeddings is the most important evaluation requirement, but because some individual processes of the pipeline producing the triples are novel themselves, it would be valuable to evaluate them individually as well. The most grounded way of evaluating part of a pipeline is to perform an A/B test on the pipeline, once with the process in question, and once without. This is the approach that Node2vec [13] used when evaluating their guided walk algorithm. The two processes that need to be evaluated are the fast walk algorithm, and the batching of negative samples. Not only would this require the pipeline to be run more than three times, for both processes an alternative needs to be developed and tested. Especially in the case of the Node2vec [13] algorithm, it is extremely computationally expensive to run on big graphs, or time consuming to develop a reasonably fast implementation. For this reason it is decided that the entire pipeline shall be tested as a whole, with the caveat that the pipeline could be less effective due to one of the novel processes.
Because the FastWalk algorithm produces a concrete intermediate result, namely walks, it is possible to perform some analysis on the types of walks it generates compared to Node2vec. This analysis can then be used to make an educated guess whether it is a viable alternative to the Node2vec walk algorithm. The goal of the Node2vec algorithm is to guide the walks to be more local or more global, so the analysis should try to quantify this in some way. Two metrics are chosen on walks of length 50 with different values of p and q: First, what is the smallest number of unique notes that make up 25% of the walk, to give an indication of locality. Second, how many unique nodes are visited in total by a walk, to measure the spread of the walk. For both values an average will be taken to cancel out randomness. And a smaller graph is used for walk extraction, to mitigate the inefficiencies of the Node2vec walking algorithm.

5.4 Exploratory Evaluation

Because inductive triple embeddings are new, it is also valuable to perform some less rigid exploratory evaluation, where comparability is not prioritized. The findings

of this exploratory evaluation can give an indication of the strengths and weaknesses of TripLan2vec, possible uses of TripLan2vec, and what future research might be valuable in the field of inductive triple embeddings.

5.4.1 Datasets

To reduce training and finetuning time, all exploratory evaluation is first performed on the WordNet11 [26] dataset using the same TripLan2vec models as are trained for triple classification. To increase validity evaluation is also performed on the UMLS [8] dataset, this dataset only contains 5,000 triples compared to the more than 100,000 triples of WordNet11 making the training process less time consuming.

5.4.2 Cosine Similarity

Cosine similarity has been shown to be an effective operation when comparing how similar vector embeddings are [24]. But as is mentioned in Section 2.3.2 BERT sentence embeddings do not work well together with cosine similarity, so it is interesting to test whether the TripLan2vec triple embeddings inherit that same property from the input vectors. To test this the average cosine similarity of a set of neighbouring triples is compared to the average cosine similarity of a set of unrelated triples, under the assumption that neighbouring triples are more similar than unrelated triples. Triple2vec [12] does not mention how well their triple embeddings work together with cosine similarity.

5.4.3 Neighbour Prediction

To evaluate how well TripLan2vec captures the graph topology, it is used to predict whether two unseen triples are likely to be neighbours. Similar to triple classification, TripLan2vec triple embeddings are used to train a Support Vector Machine that is then evaluated on a test set. The training data for this model is generated by extracting pairs of neighbouring triples and pairs of unrelated triples from the training data and then classifying them as either being neighbours or not. The test data is sourced from the test dataset in a similar way. No comparative results are available for neighbour prediction because Triple2vec [12] is not capable of making inductive predictions, and KG-BERT [42] can only classify individual triples.

5.4.4 Entity extraction

A core property of a line-graph is that neighbouring triples always share an entity, together with the neighbour classifier trained in the previous section, this can be used to extract concrete entities from a textual triple description. To do this first the knowledge graph from the training data is reduced so that for each entity there are at least two triples containing it. Then for an unseen triple from the test data a triple embedding is generated using TripLan2vec, the neighbour classifier is used to score the likelihood for each triple in the reduced knowledge graph that it neighbours the target triple. For each entity in the knowledge graph the average neighbour score is taken and the top ten are compared with the actual entities of the target triple. Because there are two entities in a triple there are two success scores, number of times both entities are in the top ten, and

the number of times one entity is in the top ten. Both Triple2vec and KG-BERT are not capable of performing entity extraction, Triple2vec because it can not produce inductive triple embeddings, and KG-BERT because it is not trained to perform this task.

5.4.5 Evaluation of Individual Pipeline Processes

To evaluate to what extend the BERT language model and gram prediction add to the expressiveness of the triple embeddings, both aspects are evaluated individually against the entire pipeline. Special care is taken when selecting the evaluation task, because without BERT embeddings as input the pipeline is no longer inductive, for this reason transductive neighbour prediction is chosen. This means that for the neighbour predictor the training data and test data are sourced from the Triple2vec training data, so it operates only on known triples. Three tests are performed, on the entire pipeline, on BERT embeddings without gram prediction, and on gram prediction without BERT embeddings. To evaluate BERT embeddings without gram prediciton the BERT embeddings are treated as triple embeddings, and to evaluate gram prediction without BERT embeddings the gram predictor is trained with random sentence embeddings as input. For time considerations only the 10% inductive subset of WordNet11 and the full UMLS datasets are evaluated.

6 Results

6.1 Triple Classification

The results of inductive and transductive triple classification are shown in Table 7. It shows the results for TripLan2vec as well as numerous competitors on the WN11 dataset, this data is taken from the KG-BERT paper [42]. The results on the transductive dataset are shown in the 100% column, the inductive results are shown in the other columns along with the extend of inductiveness as a percentage. Here 5% indicates that only 5% of the available training data is used to train the model. The results of the methods are shown as percentages that indicate how many of the triples in the test set are correctly classified. Because there are just two classes (true and false) a score of 50% is as bad as random guessing, a score of 100% means that all triples are correctly classified.
The results are achieved by feeding the triple embeddings generated with TripLan2vec as training data to a classifier. Three classifiers are used, Logisitic Regression [27], a Support Vector Machine [27], and a neural network. All three have similar results but a Support Vector Machine performs slightly better than the others.

Table 7: Results triple classification on WN11.

Method	5%	10%	15%	20%	100%
KG-BERT [42]	83%	86%	87%	88%	94%
TripLan2vec	69%	71%	77%	78%	81%
TransE [41]	52%	53%	54%	57%	76%
DistMult [44]	55%	55%	56%	56%	87%
ComplEx [39]	55%	55%	55%	56%	–
TransD [17]	53%	54%	55%	57%	86%

From the results it is clear that KG-BERT outperforms TripLan2vec easily in every test, demonstrating that a finetuned BERT Model is superior to the custom network created for TripLan2vec. But it is also clear that inductively TripLan2vec outperforms every other method tested outside of KG-BERT, beating TransE even transductively.

6.2 Exploratory Evaluation

6.2.1 Cosine Similarity

To evaluate to what extend cosine similarity gives a meaningful indication of similarity, the average cosine similarity is taken of a list of neighbouring triple pairs and a list of unrelated triple pairs. For both lists the averages are 1.0, furthermore every single comparison results in a similarity of 1.0 or a value very close to 1.0. It can thus be concluded that as with BERT sentence embeddings the TripLan2vec embedding space is anisotropic [11], but like BERTs sentence embeddings the vectors are still meaningful as demonstrated with triple classification.

6.2.2 Neighbour Prediction

The results of neighbour prediction are shown in Table 8, the top row shows the extend of inductiveness as a percentage where 100% is the full training data, the bottom row shows the resulting score. As with triple classification there are just two classes (neighbour and not neighbour) so a score of 50% is as bad as random guessing and a score of 100% means that all triple pairs are correctly classified. Because there is no prior research on inductive triple embeddings there are no competitors to compare with, but with the full training data the score is significantly higher as random, and the efficacy is very resilient against smaller datasets.

Table 8: Results neighbour classification.

Dataset	5%	10%	15%	20%	100%
WN11	55%	69%	75%	82%	90%
UMLS	–	–	–	–	93%

6.2.3 Entity Extraction

Table 9 shows the results of hits@10 entity extraction on the WN11 and UMLS datasets, as well as the total entities in the training graph. The 'One entity' column indicates how many times one of the entities from a target triple is part of the top ten best fitting entities as scored by the entity prediction model. The 'Both entities' column indicates how many times both entities of the target triple are part of the top ten best fitting entities. Note that unlike the previous evaluation tasks there are more than two classes, so even though a score of 18% on both entities for WN11 is not particularly useful, it is still significant.

Table 9: Entity extraction using neighbour predictor hits@10.

Dataset	Total entities	One entity	Both entities
WN11 10%	11949	79%	18%
UMLS	135	100%	71%

6.2.4 Evaluation of Individual Pipeline Processes

In Table 10 the results are shown for the evaluation of the individual processes of the full TripLan2vec pipeline against the full pipeline. As with inductive neighbour prediction and triple classification a score of 50% is as bad as random and 100% is the best possible. First TripLan2vec is run with random vectors as input instead of BERT embeddings, then unprocessed BERT embeddings are used as triple embeddings, and finally the entire pipeline is run. From the results it is clear that both processes on their own have strong predictive power, with BERT sentence embeddings as the most predictive. But the complete TripLan2vec pipeline has the best performance, so it can be concluded that the language context and the graph context reinforce each other to create more meaningful embeddings.

Table 10: Results on transductive neighbour classification.

Method	WN11 10%	UMLS
TripLan2vec on Random Input Vectors	80%	57%
Unprocessed BERT Sentence Embeddings	95%	89%
TripLan2vec on Bert Sentence Embeddings	97%	94%

6.3 FastWalk

To evaluate to what extend FastWalk can guide walks similarly to Node2vec, both algortihms are run on the same graph with varying values for p and q. The graph used is a subset of WordNet18 [8], to make the run time of Node2vec manageable. For each node in the line graph 1 walk is generated of length 50, then the average number of unique nodes visited per walk is taken (spread), and the average (minimal) number of unique nodes making up 25% of the visited nodes by a walk is taken (locality). The results for a varying value of p are shown in Figure 11, FastWalk is shown in red, Node2vec is shown in blue, the spread is indicated with circles, and the locality with squares. In Figure 12 the results for a varying value for q are shown. From both plots it is clear that FastWalk and Node2vec behave differently under different values for p and q. However it is also clear that with FastWalk the characteristics of a walk are strongly guidable using p and q, but effective values for p and q are not directly translatable between algorithms; separate finetuning is required.

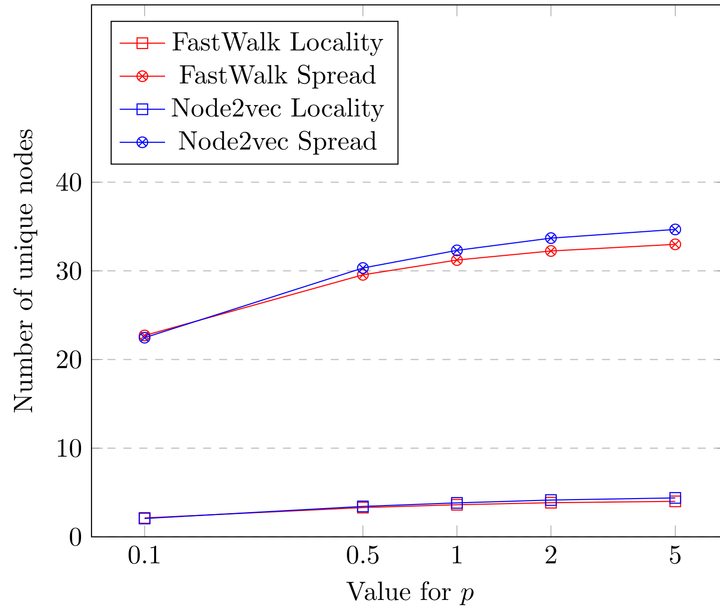

Figure 11: Walk locality and spread between FastWalk and Node2vec with $q = 1$

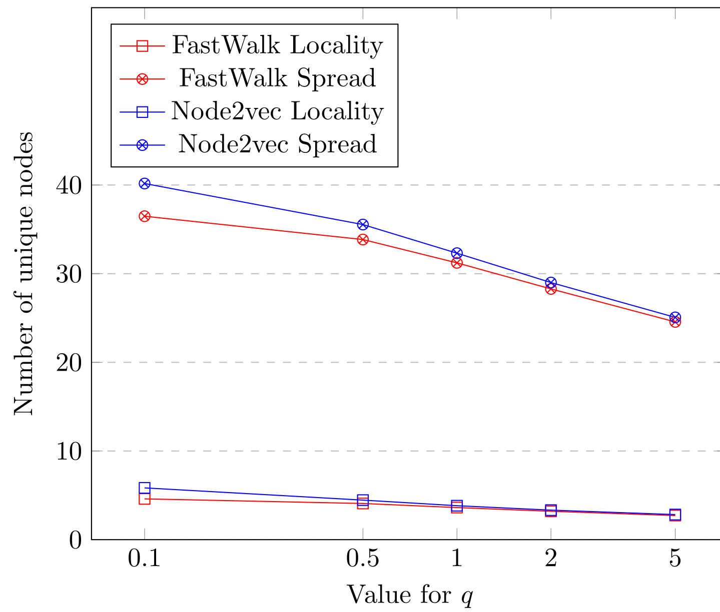

Figure 12: Walk locality and spread between FastWalk and Node2vec with $p = 1$

7 Discussion

In this section a discussion is provided of the results presented in the previous section, and the thesis at large. First the overall limitations of the research will be discussed, then the strengths of the TripLan2vec are discussed, and to conclude future avenues of research are given based on the experience of the author.

7.1 Limitations

7.1.1 Evaluation

The biggest limitation of this thesis is the evaluation of TripLan2vec. The main evaluation method chosen (triple classification) is likely not an optimal task for triple embeddings, despite performing reasonably well, it is not unlikely that the language enhanced triple embeddings would have been able to beat Triple2vec [12] on triple labeling. The reason that further evaluation was not performed is due to time limitations, it was too time consuming to both construct a knowledge graph that contains language descriptions and useful labels, and then reevaluate Triple2vec with it.
The main experiment (triple classification) is only performed on the WordNet11 [8] dataset, making the results less generalizable over other datasets. Initially the plan was to run the same evaluation on the FreeBase13 dataset, this is because the KG-BERT paper [42] contained results on triple classification for this dataset as well. But during the walk generation with FastWalk it was discovered that, likely due to either a bug or a naive implementation, the algorithm ran out of memory. This could have been solved but it was deemed better to spend that time on exploratory evaluation.

7.1.2 Finetuning

Extensive time was spent on finetuning the parameters and hyper parameters of the pipeline, but two assumptions were made to speed the process up that are likely broadly true but not fully accurate. Firstly, the best value for every parameter was evaluated independently of other parameters, under the assumption that a good value for parameter X under a certain set parameter values, is also a good value for parameter X under another set of parameter values. It is known that this is not the optimal method for choosing model parameters, and multiple better methods exist one of which is GridSearch [33], here multiple combinations of parameter values are run multiple times to establish the best combination. The second assumption was that parameters that perform well on a simplified task, also perform well on the main task. This meant that evaluation was greatly sped up, because the training time was reduced, and the evaluation task was chosen to be quick. But there is no reason to believe that this assumption always holds.

7.2 Strengths

Despite the limited evaluation, TripLan2vec was able to generate triple embeddings that are versatile in application, and resilient under increasing inductiveness. The triple embeddings generated using one model could both be used to perform triple classification, as well as neighbour prediction. Even when only 5% of the available training data

was used the embeddings still showed reasonable results, and out performed all other non-inductive methods with triple classification.
Another strength of TripLan2vec is its speed, transductive triple classification on WordNet11 was done in a little more than 14 hours, this includes generating the walks, training the mapping network, and training the triple classifier. This does not include parameter optimization, but a quick pipeline speeds this process up as well.

7.3 Future Work

7.3.1 Evaluation

A limitation of this thesis is the sub optimal evaluation, especially the lack of direct comparison between Triple2vec [12] and TripLan2vec. It would be insightful to see how these methods compare when evaluated on the same dataset, this would show to what extend the added language context improves the performance of triple embeddings. Moreover a direct comparison can be used to better quantify the difference in behaviour between FastWalk and Node2vec walk extraction, whether the time trade off is worth it.

7.3.2 Hybrid Models

One challenge of selecting a downstream tasks for TripLan2vec is that for many tasks, such as recommendation engines, the knowledge graph contains users or customers as entities. Because textual descriptions are required to generate triple embeddings, and customers are hard to meaningfully describe in a unique way using one sentence, TripLan2vec could not be used for this. It would be interesting to see if it is possible to create an embedding technique that can utilize language models when textual descriptions are available but fall back on pure graph structure when not.

7.3.3 Beyond Language Models

In this thesis the input embeddings for TripLan2vec are BERT sentence embeddings, but the modularity of the pipeline means that it is trivial to swap BERT with another embedding model. This could be another language model, but also another embedding modality altogether, such as image embeddings [18], or sound embeddings [2]. In future work it could be evaluated if these types of models can also be used to enrich triple embeddings or even generate them inductively.

8 Conclusion

Triple2vec [12] demonstrates that triple embeddings are a powerful addition to the knowledge graph embedding family, and KG-BERT [42] shows that language models can be utilized effectively in many knowledge graph related tasks. TripLan2vec is the attempt to combine both these insights to create triple embeddings that are versatile in application, and resilient against limited training data.

TripLan2vec produces embeddings that perform well at many different tasks, even when just 5% of available training data is used. In these inductive tests TripLan2vec outperforms all competing methods except for KG-BERT [42]. Thereby answering both the main research question *"How well do natural-language-enhanced triple embeddings perform?"* and sub question 2 *"How well do inductive natural-language-enhanced triple embeddings perform?"*. This result is achieved with a modular pipeline that utilizes a state of the art pre-trained language model, an innovative walk extraction algorithm that has a lower time complexity than current state of the art, and an innovative network architecture that can map from language embedding space to triple embedding space.

Sub research question 1: *"What is the best strategy for evaluating TripLan2vec triple embeddings?"* is answered pragmatically, by taking into account time constrains, comparability against other methods, and suitability against the strenghts and weaknesses of TripLan2vec. Triple classification is chosen because the datasets for this are available with standardized training, inductive training, and test data, as well as the performance characteristics from other methods allowing for easy comparison. Besides triple classification some exploratory evaluation is performed to demonstrate the versatility of triple embeddings and as an invitation for further research. From this exploratory evaluation it can be concluded that TripLan2vec embeddings don't work well together with cosine similarity, but are good at neighbour prediction.

Finally sub research question 3: *"What is the best strategy to determine the optimal parameters and hyper parameters for the neural network that maps from language embedding space to triple embedding space?"*, is answered by evaluating each parameter separately with a simplified evaluation method. This allows for a large amount of different combinations to be explored in a relatively short time, it is deemed that this is more important then exploring fewer parameter values more thoroughly.

To conclude, this thesis shows that it is possible to create inductive and transductive triple embeddings that can compete with the state of the art in some tasks, but also that more research into the field is required to unlock the full potential.

References

[1] Antoniou, G., and Van Harmelen, F. *A semantic web primer*. MIT press, 2004.

[2] Aytar, Y., Vondrick, C., and Torralba, A. Soundnet: Learning sound representations from unlabeled video. *Advances in neural information processing systems 29* (2016).

[3] Bollacker, K., Evans, C., Paritosh, P., Sturge, T., and Taylor, J. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), pp. 1247–1250.

[4] Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., and Yakhnenko, O. Translating embeddings for modeling multi-relational data. *Advances in neural information processing systems 26* (2013).

[5] Bridle, J. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. *Advances in neural information processing systems 2* (1989).

[6] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems 33* (2020), 1877–1901.

[7] Daza, D., Cochez, M., and Groth, P. Inductive entity representations from text via link prediction. In *Proceedings of the Web Conference 2021* (2021), pp. 798–808.

[8] Dettmers, T., Minervini, P., Stenetorp, P., and Riedel, S. Convolutional 2d knowledge graph embeddings. In *Proceedings of the AAAI conference on artificial intelligence* (2018), vol. 32.

[9] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[10] Ehrlinger, L., and Wöss, W. Towards a definition of knowledge graphs. *SEMANTiCS (Posters, Demos, SuCCESS) 48*, 1-4 (2016), 2.

[11] Ethayarajh, K. How contextual are contextualized word representations? comparing the geometry of bert, elmo, and gpt-2 embeddings. *arXiv preprint arXiv:1909.00512* (2019).

[12] Fionda, V., and Pirrò, G. Learning triple embeddings from knowledge graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2020), vol. 34, pp. 3874–3881.

[13] Grover, A., and Leskovec, J. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining* (2016), pp. 855–864.

[14] Henderson, K., Gallagher, B., Eliassi-Rad, T., Tong, H., Basu, S., Akoglu, L., Koutra, D., Faloutsos, C., and Li, L. Rolx: structural role extraction & mining in large graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining* (2012), pp. 1231–1239.

[15] Huang, Z., and Mamoulis, N. Heterogeneous information network embedding for meta path based proximity. *arXiv preprint arXiv:1701.05291* (2017).

[16] Hussein, R., Yang, D., and Cudré-Mauroux, P. Are meta-paths necessary? revisiting heterogeneous graph embeddings. In *Proceedings of the 27th ACM international conference on information and knowledge management* (2018), pp. 437–446.

[17] Ji, G., He, S., Xu, L., Liu, K., and Zhao, J. Knowledge graph embedding via dynamic mapping matrix. In *Proceedings of the 53rd annual meeting of the association for computational linguistics and the 7th international joint conference on natural language processing (volume 1: Long papers)* (2015), pp. 687–696.

[18] Kiela, D., and Bottou, L. Learning image embeddings using convolutional neural networks for improved multi-modal semantics. In *Proceedings of the 2014 Conference on empirical methods in natural language processing (EMNLP)* (2014), pp. 36–45.

[19] LeCun, Y. A., Bottou, L., Orr, G. B., and Müller, K.-R. Efficient backprop. In *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[20] Lederer, J. Activation functions in artificial neural networks: A systematic overview. *arXiv preprint arXiv:2101.09957* (2021).

[21] Li, B., Zhou, H., He, J., Wang, M., Yang, Y., and Li, L. On the sentence embeddings from pre-trained language models. *arXiv preprint arXiv:2011.05864* (2020).

[22] Maggiora, G. M., Elrod, D. W., and Trenary, R. G. Computational neural networks as model-free mapping devices. *Journal of chemical information and computer sciences 32*, 6 (1992), 732–741.

[23] Matthew Peters, Mark Neumann, M. I. M. G. Deep contextualized word representations. In *Proceedings of North American Chapter of the Association for Computational Linguistics* (2018).

[24] Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[25] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems 26* (2013).

[26] Miller, G. A. Wordnet: a lexical database for english. *Communications of the ACM 38*, 11 (1995), 39–41.

[27] Nasteski, V. An overview of the supervised machine learning methods. *Horizons. b 4* (2017), 51–62.

[28] Nickel, M., Tresp, V., and Kriegel, H.-P. A three-way model for collective learning on multi-relational data. In *Icml* (2011).

[29] Oh, I.-S., and Suen, C. Y. A class-modular feedforward neural network for handwriting recognition. *pattern recognition 35*, 1 (2002), 229–244.

[30] Perozzi, B., Al-Rfou, R., and Skiena, S. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining* (2014), pp. 701–710.

[31] Pirrò, G. Fact checking via path embedding and aggregation. *arXiv preprint arXiv:2011.08028* (2020).

[32] Pirrò, G. Lognet: Local and global triple embedding network. In *International Semantic Web Conference* (2022), Springer, pp. 336–353.

[33] Pontes, F. J., Amorim, G., Balestrassi, P. P., Paiva, A., and Ferreira, J. R. Design of experiments and focused grid search for neural network parameter optimization. *Neurocomputing 186* (2016), 22–34.

[34] Reimers, N., and Gurevych, I. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).

[35] Sun, Z., Deng, Z.-H., Nie, J.-Y., and Tang, J. Rotate: Knowledge graph embedding by relational rotation in complex space. *arXiv preprint arXiv:1902.10197* (2019).

[36] Svozil, D., Kvasnicka, V., and Pospichal, J. Introduction to multi-layer feedforward neural networks. *Chemometrics and intelligent laboratory systems 39*, 1 (1997), 43–62.

[37] Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., and Mei, Q. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web* (2015), pp. 1067–1077.

[38] Tang, L., and Liu, H. Leveraging social media networks for classification. *Data Mining and Knowledge Discovery 23*, 3 (2011), 447–478.

[39] Trouillon, T., Welbl, J., Riedel, S., Gaussier, É., and Bouchard, G. Complex embeddings for simple link prediction. In *International conference on machine learning* (2016), PMLR, pp. 2071–2080.

[40] Wang, Q., Mao, Z., Wang, B., and Guo, L. Knowledge graph embedding: A survey of approaches and applications. *IEEE Transactions on Knowledge and Data Engineering 29*, 12 (2017), 2724–2743.

[41] Wang, Z., Zhang, J., Feng, J., and Chen, Z. Knowledge graph embedding by translating on hyperplanes. In *Proceedings of the AAAI conference on artificial intelligence* (2014), vol. 28.

[42] Yao, L., Mao, C., and Luo, Y. Kg-bert: Bert for knowledge graph completion. *arXiv preprint arXiv:1909.03193* (2019).

[43] Zhang, Y., Wang, S., Ji, G., and Phillips, P. Fruit classification using computer vision and feedforward neural network. *Journal of Food Engineering 143* (2014), 167–177.

[44] Zhang, Z., Zhuang, F., Qu, M., Lin, F., and He, Q. Knowledge graph embedding with hierarchical relation structure. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing* (2018), pp. 3198–3207.

[45] Zou, X. A survey on application of knowledge graph. In *Journal of Physics: Conference Series* (2020), vol. 1487, IOP Publishing, p. 012016.

[46] Knowledge graph job application apple. `https://hired.com/job/siri-software-engineer-knowledge-graph-testing123123`. Accessed: 2023-03-20.

[47] The pytorch library. `https://pytorch.org`.