Transformer-based knowledge graph completion

(Uzupełnianie grafów wiedzy przy użyciu transformerów)

Dawid Wegner

Praca magisterska

Promotor: dr hab. Jan Chorowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

29 września 2021

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Streszczenie

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Contents

1. Introduction

In this chapter we will define the notion of knowledge graph along with the problem of completing it. Furthermore, we will discuss the main motivations for solving the considered problem and give an overview of the past work. Finally, we will provide an outline of our approach to the problem of completing a knowledge graph.

1.1. Problem formulation

Consider a set of objects $e \in E$ and a set of relations $r \in R$. Formally, a **knowledge graph** is a set of triples $(e_1, r, e_2) \in K$, which represents relations between the considered objects. Particularly, $e_1 \in E$ is called to be in a (directed) relation $r \in R$ with $e_2 \in E$ if and only if $(e_1, r, e_2) \in K$. We will refer to an object $e \in E$ as an **entity** and an object $r \in R$ as a **relation**. Besides it, when considering a triple (e_1, r, e_2), we will refer to e_1, e_2 as a **head entity** and a **tail entity**, respectively. From the practical perspective, we can think of our structure as a directed graph where vertices are represented by entities and edges are labelled by relations. A toy example illustrating a knowledge graph is shown in *Figure* 1.1.

While we would expect a knowledge graph to contain any information that is useful for a user, in practise the vast majority of relations are missing. This fact becomes intuitive when one realizes that adding one edge to a graph can add information about relations between several entities that are not adjacent to the added edge. As a result, the number of missing relations can grow exponentially with the increase of the number of edges. In practice, for a typical knowledge graph, except facts that contained directly in edges, a large portion of information can be inferred by taking into account multiple connections at once. The considered problem is known as the **Knowledge Graph Completion** problem.

Formally, we are given a knowledge graph G and our goal is to find triples $(e_1, r, e_2) \in K$ that are likely to represent true facts. Specifically, given a head entity $e_1 \in E$ and a relation $r \in R$, we aim to find a tail entity $e_2 \in E$, such that (e_1, r, e_2) is true. The analogous goal can be defined for a missing tail entity, given a head entity and a relation. The knowledge graph completion methods can be divided into those that

extract information only from a graph and those that utilize additional information e.g. text-based definitions of entities. In this thesis, we only leverage information that is contained in a given graph, particularly our models falls into the graph-only category.

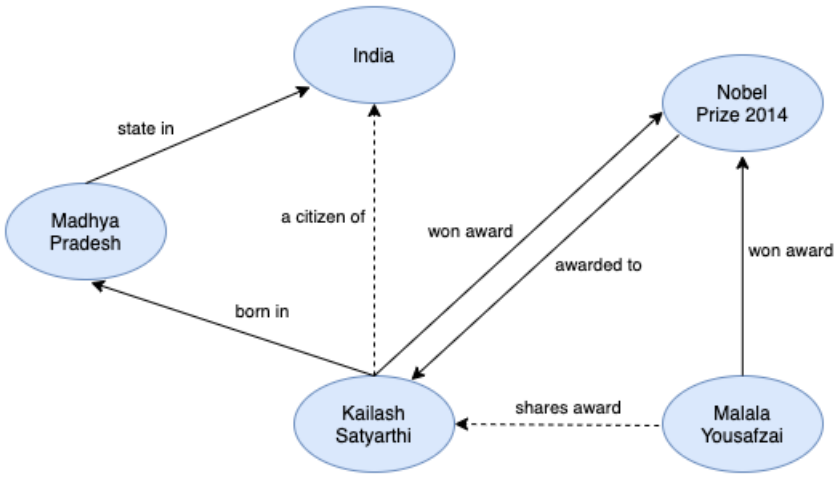

Figure 1.1: An example of a knowledge graph. Solid edges represent triples that are present in the knowledge graph, while dashed edges represent knowledge that is missing, but can be inferred directly from the graph.

1.2. Motivations for solving the problem

The first question to ask could be whether we really need a sophisticated model in order to complete a knowledge graph. Actually, most of knowledge graphs are very dynamic systems that evolve over time, making it impossible for humans to complete them by hand. Additionally, the problem of completing a knowledge graph is the very complex one. While simple heuristic approaches have been tried in the past, they only covered simple cases like inverse relations. As a result, in order to extract more sophisticated relations, we need a model that understands entities and their relations more deeply.

Solving the stated problem is motivated further by the fact that it has many real-world applications. First of all, consider a situation when you ask a voice assistant to answer a question about the nationality of your favourite actor. The answer to this question can be fetched from a knowledge graph, assuming that it is complete enough to contain this information. In order to be able to answer more questions, one needs to assure that the graph is complete. The class-leading voice assistants and search engines actually make use of knowledge graphs to provide answers to users' questions. Those graphs are developed by parsing online encyclopedias like Wikipedia as well as news websites and as a result are often incomplete.

Knowledge graphs are are also useful in many other areas. For instance, they are used to boost recommendation engines of popular content and social media platforms. Furthermore, knowledge graphs are leveraged in the healthcare industry, opening medical researchers to gain more insights from data. Additionally, they are also used to represent the relations between words in human languages. Lastly, knowledge graphs are leveraged by financial services to secure human decisions and prevent money laundering initiatives.

1.3. Related work

In the recent years, various approaches have been tried to tackle the problem of knowledge graph completion. The most successful ones are based on data-driven models, often called machine learning models. These methods will be discussed more deeply in the *Chapter* 2. For our task specifically, we can divide the considered models into ones that operate on triples (representing graph edges) alone and those that utilize graph context e.g. neighbours of a head/tail entity. Recently, a new branch of models that leverage additional text-based information has been established, but those models are not considered in this thesis.

While the first triples-only methods were proposed in 2013, the new models are still developed, surpassing the old ones. One of the first baseline models, learning latent representations of entities and relations, is known as the *TransE* model introduced in [1]. The model was later extended in [15] to *STranse*, which allows entities to have relation-specific representations. Several other improvements, utilising convolutional neural networks to learn better representations, have been proposed, including *ConvE* model presented in [19]. While many other triples-only models have been developed, they retain the same quality of predictions. Therefore, we compare our methods only with the above-mentioned models.

In the past few years, several models have been developed to utilise graph-context information. The first applications of those models to the considered problem have been proposed back in 2017. While many of the approaches failed to surpass triples-only models, a few models have provided a considerable quality improvement. One of the successful approaches include [21] that tries to walk through a graph to find a matching entity. The other popular methods are based on graph neural networks, especially on the *GCN* architecture proposed in [23]. The best-performing ones include the *SACN* model presented in [22] and recently developed *CompGCN* model introduced in [29]. Although many other models have been proposed, as shown in [28], some of the authors have reported inflated quality results due to improper evaluation protocol. Therefore, we compare our techniques only with the mentioned models.

1.4. Our approach

One of the most ground-breaking machine learning architectures in the recent few years is the *Transformer* model introduced in [17]. It has been successfully applied to many sequence-to-sequence problems, outclassing other models. Moreover, several improvements have been proposed to the original architecture. For instance, the authors of *BERT*, introduced in [24], proposed a language representation model that is pretrained on specially designed tasks to later boost its quality on the downstream task. Just like in the case of the original transformer architecture, the proposed BERT model outperformed other architectures in many classification tasks, especially in the area of Natural Language Processing. Following the success of the proposed approach, researchers have developed even better architectures. One of the recently developed models include *ALBERT*, proposed in [26], that is a lite version of the BERT model. The other interesting approach is taken in [27], which trains the BERT model adversarially. Besides it, several components of the baseline architecture have been improved over time, for instance the authors of [25] showed that changing the location of normalization layers lead to better performance of the model.

Motivated by the recent advancements in the transformer-based models, we decided to tackle to the problem of knowledge graph completion, using the above-mentioned models. In particular, we explore both triples-only and graph-context methods, determining whether a transformer can benefit from an additional context information. Besides it, we compare the developed methods with the related models on two benchmark datasets, known as WN18RR and FB15K-237.

2. Background

In this chapter we will review the theory behind the techniques used to complete knowledge graphs. *Section* 2.1 is devoted to the basic properties and concepts of neural networks. In the subsequent sections discuss more specialized models that are used to tackle the knowledge graph completion problem. In *Section* 2.2 and *Section* 2.3 we cover models learning graph representations. *Section* 2.5 provides and overview of how reinforcement learning can be utilized to tackle the considered problem. In *Section* 2.4 we discuss the transformer architecture, which is the main building block for our methods. Finally, in *Section* 2.6 we show how the methods discussed in the previous sections have been utilized in the past to tackle the considered problem.

2.1. Artificial neural networks

Neural networks are machine learning models inspired by a human brain. While the first neural models have been proposed decades ago, their popularity was very limited due to the poor quality of these models. In the recent few years, the situation has changed as researchers developed sophisticated algorithms to optimize such models while in parallel computing resources have been grown significantly. At this point, neural networks have dominated other approaches to artificial intelligence, finding applications in practically every areas of human life.

2.1.1. Feed-forward neural networks

Formally, a neural network is a complex mathematical function that maps input vectors to output vectors. One of the simplest neural network can be defined using the formula

$$y = f(Wx + b) \tag{2.1}$$

where x, y denotes input and output vectors of size n, m respectively, W is a $n \times m$ matrix of parameters, b denotes a so-called bias vector of size m and f is a fixed activation function that provides adds non-linearity to our formula. The more complex neural network would repeat the process of mapping vectors several times e.g.

$$y = f_3(f_2(f_1(W_1x + b_1)W_2 + b_2)W_3 + b_3) \tag{2.2}$$

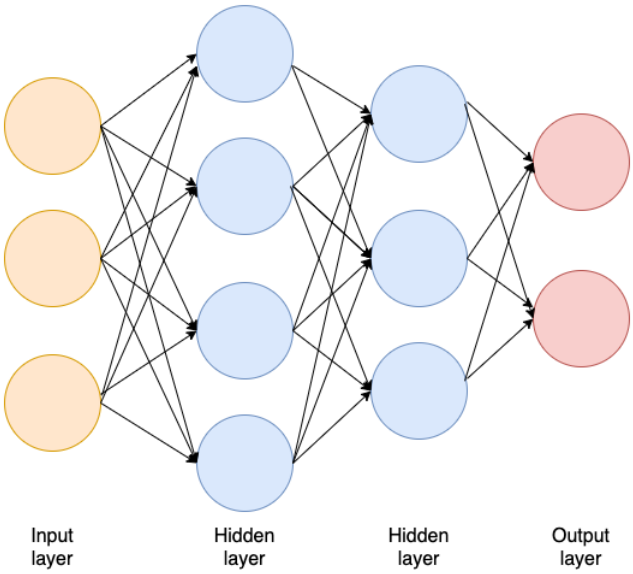

Figure 2.1: An example of a feed-forward network containing 2 hidden layers (marked in blue). The input layer (marked in blue) is composed of three neurons, while the output layer (marked in red) is composed of two neurons.

Due to its linear form, such network is often called a feed-forward neural network. An input vector x is often called an input layer, the intermediate vectors are called hidden layers and an output vector y is called an output later. The above-defined neural network has 2 hidden layers, defined as outputs of f_2 and f_3 activation functions. A feed-forward neural network can be illustrated in the form presented in *Figure* 2.1.

While we have shown how a neural network maps input vectors to output vectors, it is still unclear how such model can be utilised to make predictions. The idea is to utilise so-called training dataset that is composed of input and expected output pairs. Specifically, it is leveraged to learn our neural network how to map inputs to outputs. Additionally, in order to make use of training dataset, one needs to convert given inputs and outputs to real-valued vectors. As neural networks are supervised on training data, they belong to the subcategory of machine learning called supervised learning. After training a neural network, its quality is rated on a validation dataset, which is disjoint with training dataset, providing an estimation of quality for examples that were not seen by the model. In particular, neural networks are known for fitting to training data too tightly while providing a poor quality on unseen examples. The techniques for preventing this phenomena, called overfitting, are discussed in *Subsection* 2.1.5. In case multiple models are tested, the validation set is used to pick the best one, while the quality of the chosen model is additionally evaluated on a test dataset as it can be a little biased to the validation dataset. The task-specific functions that rate models are often called evaluation metrics.

In order to fit a neural network to a given training dataset, one needs to find parameters that provides the best correspondence between predicted output vectors and expected output vectors. The trainable parameters include linear transformation matrices and bias vectors. For instance, for a neural network defined by *Equation* 2.2, the trainable parameters include $W_1, W_2, W_3, b_1, b_2, b_3$. Intuitively, the more parameters the network has, the higher its expressiveness. The optimization process of neural networks is performed by a specially designed algorithm that is described in *Subsection* 2.1.4.

2.1.2. Convolutional neural networks

Consider a task of classifying images into one of given categories e.g. $C = \{cat, dog\}$. It is easy to develop a feed-forward neural network that would treat images as input vectors and apply a few layers in order to obtain 1-dimensional output, representing a probability of an image to represent a cat. Actually, this approach works decently, but it suffers from the overfitting problem, causing that the quality of such model on unseen examples is lower than expected. The problem with the feed-forward approach is that such network tries to learn the dependencies between all pixels at the same time. The more natural way would be to learn a model to firstly recognize some low-level patterns like edges and then to recognize larger objects. Convolutional networks are models that implement this idea, enforcing the model to learn from neighbouring features.

In the case of convolutional networks, an input vector is represented by a 3-dimensional tensor of size $H \times W \times C$ interpreted as a height, width and the number of channels respectively. A single convolutional layer maps the input tensor, often called as input feature map, to the output tensor, which is called an output feature map. A single output neuron is computed by multiplying element-wise a patch of input feature map with a learnable 3D matrix of size $\tilde{h} \times \tilde{w} \times C$. The process is repeated by going through the input feature map with a sliding window of size $\tilde{h} \times \tilde{w} \times C$. The whole process for an input feature map with the number of channels $C = 1$ is shown in *Figure* 2.2.

The discussed algorithm maps an input feature map of size $H \times W \times C$ to an output feature map of size $\tilde{H} \times \tilde{W} \times 1$, using a learnable kernel matrix of size $\tilde{h} \times \tilde{w} \times C$. In practice, this process is repeated $\tilde{C}$ times for independent kernel matrices, where $\tilde{C}$ is called the number of filters. In this case, the output feature map is of size $\tilde{H} \times \tilde{W} \times \tilde{C}$.

Convolutional networks are learnt by modifying parameters contained in kernel matrices. Mathematically, nothing prevents composing a neural network that is developing of a few convolutional layers followed by feed-forward layers. In the first case

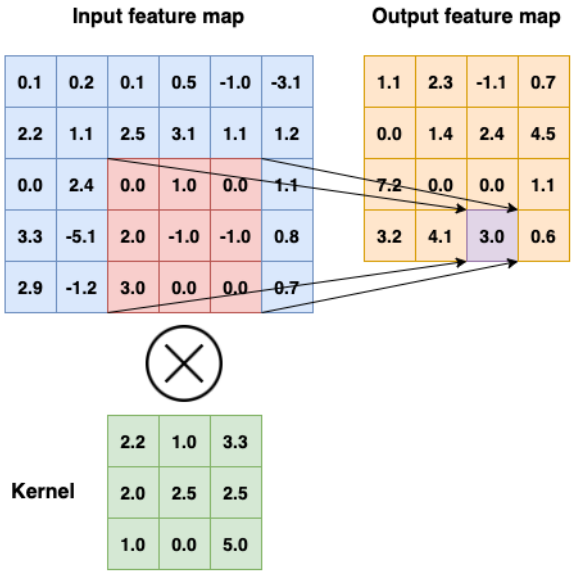

Figure 2.2: An example of how convolutional layer maps input feature map (marked in blue) to output feature map (marked in orange). A single patch (marked in red) is multiplied element-wise with a learnable kernel matrix in order to produce one output feature (marked in purple). A sliding window goes though the entire image, multiplying input matches with a kernel matrix. Note that while in this example an input feature map contains only one channel, in case it would contain more channels C, a kernel matrix would be of size $C \times 3 \times 3$.

inputs can be treated as 3D tensors, while in the latter case, they can be transformed to 1D representations.

2.1.3. Loss functions

In order to train a model, one needs to define a similarity between predicted outputs $\tilde{y}$ and ground-truth outputs y. A function that defines this similarity is called a loss function and depends on a considered task. One popular case is when a target variable y is continuous and the loss function should be based on the distance between y and $\tilde{y}$. In the field of machine learning, this is often called a regression task. While many different functions can be designed to optimize such model, the most popular one is

$$L_2(y, \tilde{y}) = ||y - \tilde{y}||_2 = \sum_i (y_i - \tilde{y}_i)^2$$

which is called L2 loss function. The other case is when $\tilde{y}$ represents an unnormalised probability distribution for a set of established classes (e.g. $C = \{cat, dog, rabbit\}$), while y is so-called one-hot vector encoding classes that are true for a specific sample as 1.0 and classes that are false as 0.0. In this case, the popular choice is to first

apply a softmax function to normalize $\tilde{y}$

$$\hat{y}_i = \frac{e^{\tilde{y}_i}}{\sum_j e^{\tilde{y}_j}}$$

and then apply cross-entropy loss function

$$L(y, \hat{y}) = -\sum_i y_i log(\hat{y}_i)$$

penalizing predictions for classes for which $y_i > 0$. Except natural intuitions that the above-defined loss function should be minimized in order to provide a good estimate of y, it also has mathematical foundations that can be found in [10]. In case there are only two classes, the discussed loss function is often called a binary cross entropy and the task of optimising such function is referred as binary classification.

2.1.4. Optimization

In the previous subsections we mentioned that a neural network is optimized by adjusting parameters contained in matrices and bias vectors. Besides it, we defined a loss function that should be minimised in order to fit to the expected outputs. In this subsection, we show how to actually minimize a loss function.

First of all, our training data samples should be split into batches of fixed size B, often referred as batch size In each training step, a batch of inputs is pushed to a model and the parameters of the model will be slightly modified to lower the value of a loss function. Specifically, a technique called gradient descent is utilised to get a direction that should be followed in order to minimize the loss function. To do so, one may compute a gradient of the loss function with respect to all parameters, using so-called backpropagation algorithm, and obtain a vector that defines the desired direction. Then, the vector of parameters can be updated by applying

$$\theta_{k+1} = \theta_k - \alpha \nabla_\theta L(\tilde{Y}, Y)$$

where α is called a learning rate. The process of updating parameters is repeated for all training batches and repeated E times, where E is referred as the number of epochs. While this naive algorithm can stuck in a local minima, several advancements have been developed in order to make it possible to converge to global optima. One of the currently popular approach is known as Adam optimizer, introduced in [7]. Over time, researchers have also developed a specialized layers that speed up the convergence. One of the most successful layers is Batch Normalization, proposed in [6], normalizing batches of inputs to the parameterised Gaussian distribution. The other popular alternative is Layer Normalization, introduced in [9], which normalizes each sample separately, making the computations more precise for small batch sizes.

2.1.5. Preventing overfitting

Neural networks are great in recognising patterns in training data. In particular, a feed-forward neural network with 1 hidden layer can represent any function, which is known as the universal approximation theorem. But in practice, we are not interested in a model that remembers all training data i.e. overfits to it. Instead, we would like to design models that generalize well, providing good quality of predictions on unseen examples. In the recent years, several techniques have been developed in order to reduce the overfitting of neural networks.

One popular way of enforcing the network to generalize is adding a dropout introduced in [4]. The idea is to (in each training step) set some random subset of neurons in a specific layer to 0.0 and rescale the other neurons by 1.0 / $(1.0 - r)$, where r (often called a dropout rate) denotes the percentage of neurons that were set to 0.0. The neurons that are set to 0.0 can be treated as removed. The process is illustrated in *Figure* 2.3. During the evaluation of the network on unseen examples, all neurons are included and no scaling is applied. The intuition behind this approach is that it enforces the network to spread out its weights to many neurons rather than relying on a small number of connections.

The other technique to reduce the overfitting is to add an additional loss that penalizes our parameters. One popular way of doing that is to add a regularization term to our loss

$$L(y, \tilde{y})^{\theta} = L_m^{\theta}(y, \tilde{y}) + \lambda \sum_i \theta_i^2$$

where θ denotes a vector of parameters, L_m is a loss provided by a network and λ is a parameter controlling the regularization strength. This regularization technique is often called a weight decay. The intuition behind this approach is that it enforces the network to keep parameter values close to 0.0 and avoid extreme values, making it less flexible.

The other regularization technique, which is designed for classification tasks, is label smoothing. The idea behind this method is to replace hard $0, 1$ labels with their soft versions. Specifically, when label smoothing parameter is set to $c \in (0, 1)$, true labels are set to $(1 - c) + c$ / N, while false labels are set to c / N, where N denotes the total number of classes. The main motivation behind this approach is that training dataset can contain mistakenly labeled examples. Another common situation is when multiple output labels are correct, while a dataset is constructed in a way that only a value for one class is set to 1.0. In this case, label smoothing encourages a model to assign some non-zero probability to each class, enabling it to assign higher probabilities to labels that may be correct.

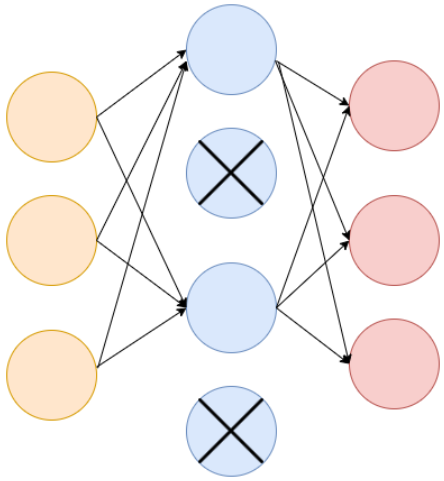

Figure 2.3: An illustration of dropout to applied to a hidden layer (marked in blue). The crossed out neurons are set to 0.0, which has an effect of ignoring their input and output connections. In this case dropout rate $r = 0.5$.

2.2. Graph embedding methods

Consider a dataset containing users' features and the problem of classifying each user to one the specified categories. In order to tackle this task, one may utilise neural networks discussed in *Section* 2.1 to learn a mapping from user's features to categories. Now, let's assume that one of our features is a list of friends of a user. In other words, users are interconnected, forming a graph. While we could try to naively encode this information by creating a one-hot vector indicating a list of friends for each user, there are more sophisticated ways to include this information. One of the most successful approaches is to use graph embedding methods that aim to learn low-dimensional vector representations of nodes.

2.2.1. Classification of graph embedding methods

Most of embedding methods are unsupervised. The goal is to produce generic, continuous representations of nodes based on their connectivity. The representation of a node is often referred as embedding. The learnt embeddings can be later used by some supervised model to perform a node classification task (e.g. classify users of a social network), link prediction task (e.g. predict new friendships) or graph classification task (e.g. classifying chemical structures). Most of embedding methods are based on so-called homophily hypothesis, which says that nodes that are highly interconnected and belong to similar network clusters should have similar embeddings. The other category of considered methods is based on structural equivalence hypothesis, saying that nodes having structural roles in a graph should be embedded closely together. Lastly, while most of methods allow to learn only fixed representations for

nodes contained in the training dataset i.e. transductive methods, there exist more specialized methods allowing predictions to be made on unseen nodes i.e. inductive methods.

2.2.2. DeepWalk algorithm and its extensions

One of the baseline models that outclassed previous approaches is DeepWalk, introduced in [5]. At each step of the algorithm, a random node $v \in V$ is chosen. Then, a random path $W = \{w_1, w_2, \ldots, w_k\}$ is constructed by starting in the node v and repeatedly going to one of the neighbours of the currently last node on constructed path. The generated paths are later used to enforce nodes having similar neighbours to be closely embedded. Specifically, let $v_j \in W$ be a random node from the generated path and its context to be defined as a set of nodes $K = \{w_{j-c}, w_{j-c+1}, \ldots, w_{j+1}, w_{j+c}\}$. Now, draw one vertex $v_c \in K$ and define $p(v_c|v_j)$ with the formula

$$p(v_c|v_j) = \frac{e^{R_j C_c^T}}{\sum_k e^{R_j C_k^T}} \tag{2.3}$$

where R_i denotes the i-th row of the learnable embeddings matrix, while C_i is the i-th row of the learnable context matrix. The goal of the above-defined model is to maximize this probability for generated v_j, v_c nodes. As a byproduct of this procedure, we get the embeddings matrix R, such that the i-th row represents the i-th node. The intuition behind this approach is it enforces nodes that have many common neighbours to have similar embeddings as they are often multiplied with the same context vectors. A toy example of how the presented algorithm could embed a small graph in 2-dimensional space is shown in *Figure* 2.4.

Interestingly, the presented algorithm was invented previously to embed words in a way that similar words have similar representations. The original method is called a Skip-gram model and was introduced in [2]. Additionally, to speed up the computation time of *Equation* 2.3, the authors proposed to replace the original softmax function with a hierarchical softmax, keeping nodes in leaves of a binary tree. The other popular method is to use negative sampling, which tries to maximize

$$p(v_c|v_j) = e^{R_j C_c^T}$$

for positive pairs, while minimizing the same formula for negative pairs that are created by drawing two random words. The analogous extensions can be applied to the DeepWalk algorithm.

While the original DeepWalk algorithm traverses the graph using DFS strategy, other strategies could be utilised. The authors of [11] generalized DeepWalk to choose between one of strategies: BFS, DFS, going back to the previous node. The proposed node2vec algorithm interleaves strategies by taking a random one in each step of

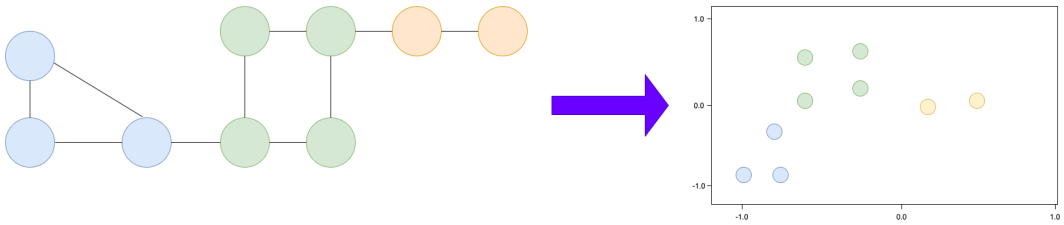

Figure 2.4: An example of how DeepWalk algorithm could embed nodes of a given graph in 2 dimensional space. The nodes are marked with colors to show the correspondence between the locations in the graph and locations in the embedding space.

constructing the path. A distribution defining a probability of choosing a specific strategy impacts the learnt representations. Most commonly, the parameters defining a distribution of strategies are finetuned to provide the best quality.

The other interesting extension proposed in [14] utilises the DeepWalk algorithm to embed nodes that have similar structural role closely together. The idea behind struc2vec is to first construct a graph encoding structural similarities between nodes and then learn embeddings by performing random walks. The authors proposed to define a structural distance between nodes $u, v \in V$ based on the number of neighbours $R_k(u), R_k(v)$ at specified distances $k \in \{1, \ldots, K\}$. Then, the constructed distances matrix is utilized to define a probability of transitioning from u to v. As a result, random walks enforce nodes having similar structure to have similar embeddings.

2.2.3. Learning representations of unseen nodes

Though we have shown many extensions of the DeepWalk algorithm, none of them scale to unseen nodes. The naive approach to extend it would be to rerun random walks when new nodes arrive, but it makes the computations very costly. In case the nodes are represented by additional pre-defined features, one may try to learn embeddings by mapping input features. One way of applying this idea is to use a model introduced in [12]. The authors propose to learn a specific supervised task along with the embeddings of nodes at the same time. Specifically, one of hidden layers of the supervised model can be treated as an embedding layer, representing an input node with a given features. In order to enforce the neighbouring nodes to have similar embeddings, random walks are applied to the embedding layer, causing the information about the neighbourhood to backpropagate to the input layer. The model is trained jointly on the supervised task and random walks.

The other popular approach, introduced in [13], learns embeddings without a use of additional features. The idea of the authors is to apply a few feed-forward layers to map an adjacency vector of a specific node to its embedding and then apply

a few more feed-forward layers in order to reconstruct the input adjacency vector. Except the reconstruction loss, additional L2 loss is applied between embeddings of neighbouring nodes, enforcing them to lie close to each other. The model is trained on both tasks jointly, ensuring that embeddings have the two above-defined properties.

2.3. Graph neural networks

In *Section* 2.2 we have shown how to learn generic representations of nodes, encoding the information about the neighbourhood. In the present section, we will show how to learn task-specific embeddings. Firstly, we will define the notion of message passing, showing how it can be leveraged to exchange the information between nodes. Then, we will present a few approaches that utilise this technique to boost predictions quality of supervised models.

2.3.1. Message passing

Passing messages between processes is a very known technique in computer science. In the context of this thesis, we will consider a graph, whose neighbouring nodes exchange some information by sending and receiving messages. Formally, assume that each node i has been assigned some representation $h_i^{(0)}$. In the k-th step, we will assign the (k + 1)-th representation of the node i by applying

$$h_i^{(k+1)} = f_k(h_i^{(k)}, \{h_j^{(k)} : j \in N(i)\}) \tag{2.4}$$

where $N(i)$ denotes the neighbours of the node i and f_k is an aggregation function. An example illustrating the described procedure is shown in *Figure* 2.5. This process is performed for each node and repeated K times, allowing a node to obtain messages from its k-hop neighbourhood. The final representations $h_i^{(K)}$ are utilised to perform the desired classification or regression task. In order to perform backpropagation and as a result update parameters of aggregation functions f_k, all of them must be differentiable.

Interestingly, all graph neural networks models can be formulated with *Equation* 2.4. The main difference between the models is the definition of the aggregation functions f_k. The other difference is how the final representations $h_i^{(K)}$ are used. In the simplest case, one may perform a node classification task. On the other side, the final representations can be aggregated to perform a link prediction or even a graph classification task. Additionally, the above-defined message passing formulation can be easily extended to take into account different edge types or features.

In case initial representations of nodes $h_i^{(0)}$ are not learnable, new nodes can be added to a graph and used for the underlying classification or regression task, without much

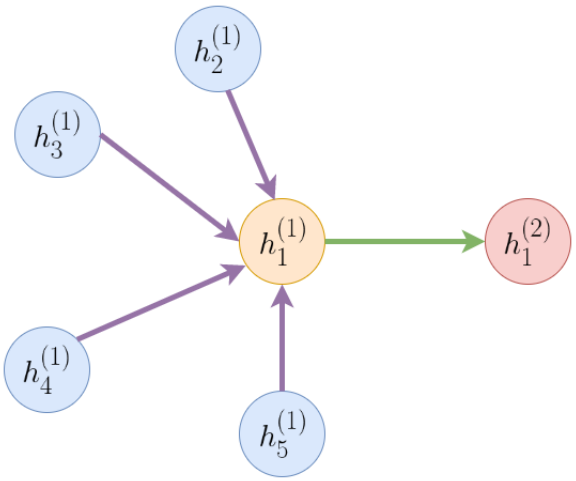

Figure 2.5: An example of how message passing is used to exchange the information between nodes. The embedding $h_1^{(1)}$ (marked in orange) of the node h_1 is updated by aggregating the embeddings of its neighbours (marked in blue). This process results in the new embedding $h_1^{(2)}$ of the node h_1 (marked in red).

loss of quality. In the other case, one needs to start with a random representation of the added node and retrain the model to learn its representation. The other posibility is to use graph embedding methods discussed in *Section* 2.2.

2.3.2. Overview of models

Over time, researchers have developed multiple ways of modelling aggregating functions f_k. One of the most successful approaches, proposed in [23], utilises the following aggregation function

$$H^{(k+1)} = f_k(H^{(k)}) = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(k)}W^{(k)}))$$

where σ denotes an activation function, $\tilde{A}$ is the adjacency matrix with added self-loops, D is a diagonal matrix, such that D_{ii} represents the order of the node i and W is layer-specific trainable matrix. The formula has its mathematical justification, relying on the graph Laplacian. From the more intuitive point of view, $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ can be thought of a convolution matrix that is based on the connectivity of nodes. The discussed model is called GCN and has been successfully applied to many tasks involving graphs.

The other popular choice for the aggregation functions f_k, proposed in [16], is Long Short-Term Memory (LSTM). In short, LSTM is a recurrent model that can take variable-length inputs, in parallel reducing the well-known vanishing gradient problem. More details of the LSTM model can be found in [10].

Yet another approach is to use a so-called self-attention mechanism that is discussed in more detail in *Subsection* 2.4.1. In the context of graph neural networks, the above-mentioned mechanism is used to learn how much attention should be put to a specific neighbour, when receiving its message. Specifically, let $\alpha_{ij} \in [0, 1]$ be a learnable weight, denoting how much attention the node i should put to its neighbour node j and for non-neighbouring nodes set $\alpha_{ij} = 0.0$. Now, let the aggregation function be defined by the following formula

$$h_i^{(k+1)} = f_k(h_i^{(k)}, K) = \sigma(\sum_{h_j^{(k)} \in K} \alpha_{ij}^{(k)} W^{(k)} h_j^{(k)})$$

where K denotes a set of the node i neighbours, σ is an activation function and $W^{(k)}$ is a learnable matrix. Additionally, in order to take into account the current representation of the node i, self-connections are added to the graph. The model is optimized to update attention weights α and parameter matrices $W^{(k)}$, providing additional insights on the importance of specific neighbours.

2.4. Transformers

Before the rise of transformers, sequence-to-sequence tasks were modelled with recurrent neural networks. These approaches have had many problems as vanishing gradients, causing the quality of models to degrade for long sequences of inputs. While researches have developed many approaches to reduce this issue, training recurrent models was still inefficient due to their inability to being parallelized. Indeed, the main assumption behind recurrent neural networks is that the outputs of the k-th step cannot be inferred until the outputs of all previous $k - 1$ steps have been calculated. The situation has changed when researchers have proposed [17], introducing the transformer architecture that avoids all of the above-mentioned issues. In the following subsections we define the key components of the proposed transformer architecture and the extensions that have been developed over time.

2.4.1. Self-attention mechanism

In the abstract form, attention mechanism can be defined as mapping a given query Q and a list of n key-value pairs (K, V) to output values. More formally, we assume that a query Q is a matrix of size $1 \times d_k$, while a list of keys and their corresponding values form the matrices K and V of size $n \times d_k$, respectively. While the attention mechanism can be modelled in many ways, one of the most popular forms is the so-called scaled dot-product attention

$$attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.5}$$

where d_k is used to scale the computed weights. The illustration of the above-defined mechanism is shown in *Figure* 2.6. Intuitively, it computes a convex sum of values, based on the similarity of the corresponding keys with the query Q. When $Q = K = V$, the described mechanism is called self-attention. Additionally, one may define attention weights, which in case of the self-attention have the form

$$\alpha_{ij} = softmax(\frac{QK^T}{\sqrt{d_k}})_{ij} = softmax(\frac{VV^T}{\sqrt{d_k}})_{ij}$$

which indicate how much attention the i-th value (query) puts to the j-th value. Indeed, the attention mechanism was developed in order to learn how much attention should be put into other elements of a sequence. Over time, several extensions of the *Equation* 2.5 have been proposed to enable the model to learn better-suited combinations of values V e.g.

$$attention(Q, K, V) = softmax(\frac{(QW_Q)(KW_K)^T}{\sqrt{d_k}})(VW_V)$$

where W_K, W_V, W_Q are learnable matrices mapping keys, values and query, enabling the dot-product between the query and keys to be computed in a custom space as well as allowing linear transformations of values.

The introduced self-attention mechanism is the key component of a transformer layer that will be discussed more deeply in *Subsection* 2.4.4. In particular, the transformer architecture makes use of multiple attention mechanisms, each learnt using an independent linear transformation of input keys, values and queries. Then, the outputs of all attention mechanisms are concatenated and projected the final output, using learnable matrix. This mechanism is called Multi-Head Attention and is used in practice instead of a single attention function. Intuitively, this extension of the basic mechanism allows the model to spread its attention to different features.

2.4.2. Pointwise feed-forward layer

While the attention mechanism allows a model to exchange the information between elements of a sequence, the elements may need to update their representations by itself. One of the most intuitive ways to model it, would be to use a few feed-forward layers, introduced in *Subsection* 2.1.1, for each element of the sequence independently. The layer that follows this pattern is called the pointwise feed-forward layer. Formally, it takes a matrix of elements of a sequence and applies a few linear transformations, each followed by an activation function. Importantly, the same transformations are applied to all elements of the sequence. This means that a model is enforced to learn generic transformations, regardless of the characteristics of a specific element.

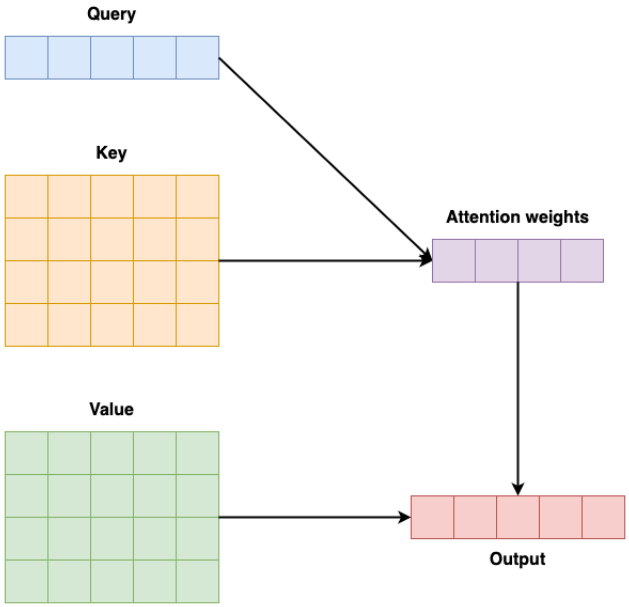

Figure 2.6: An illustration of the attention mechanism. In the first stage of the procedure, a query (marked in blue) and keys (marked in orange) are used to compute attention weights (marked in purple). Then, the output (marked in red) is formed by multiplying a matrix forming attention weights with a matrix values i.e. computing a convex combination of values.

2.4.3. Positional embeddings

It's a common case that the order of input sequence elements is important. For instance, consider a task of translating a sentence from one language to another. In this task, different permutations of words can lead to various semantics of the whole sentence. In order to encode the information about positions of the elements, one may keep a vector representation of each position and then merge it with a representation of an element on a specific position. Such representation of a position is often referred as a positional embedding.

The representation of an element can be merged with a position embedding by concatenating their vectors. In case both vectors have the same dimensionality, instead of concatenation, it is common to sum the corresponding embeddings element-wise. One approach to assign a representation to a word is to come up with a function that could identify each position uniquely. In this case, the most popular choice is to use the one based on different frequencies of some cyclic functions e.g.

$$PE_{(p,k)} = \begin{cases} sin(\frac{p}{100000^{k/d}}) & \text{if } k = 2i \\ cos(\frac{p}{100000^{k/d}}) & \text{if } k = 2i+1 \end{cases}$$

where p is a position that will be represented by an embedding, i refers to an individual dimension of the embedding and d denotes the total number of dimensions.

The other popular choice is to start with random positional embeddings and update them during the training process, based on the gradient information.

2.4.4. Transformer encoder and decoder

The introduced multi-head self-attention layer along with the pointwise feed-forward layer form together a transformer layer, which is illustrated in *Figure* 2.7. The idea behind this layer is to exchange the information between the elements of a sequence and then update each element independently, leveraging the two above-mentioned sublayers. Additionally, in order to increase the flow of information, skip connections between the sublayers are applied. Specifically, the outputs of the attention sublayer are summed with the original inputs as well as the outputs of the pointwise sublayer are summed with the outputs of the attention layer. This allows the model to utilise the information contained in the previous layers. Besides, it reduces the vanishing gradients problem i.e when the gradient is vanishingly small.

The transformer encoder is composed of a few stacked transformer layers. It takes as an input a sequence of n input elements, represented by their embeddings (with optionally positional embeddings applied), and returns a sequence of n embeddings, each representing the corresponding input element in the context of other elements. Similarly to the encoder, the decoder is composed of a few transformer layers. The main distinction is that it takes as its input a sequence of n output elements along with the outputs of the encoder. Most commonly, the outputs of the encoder are injected after a few first transformer layers by summing the output vectors element-wise. The sequence of output elements is often used to provide an information about the previous outputs. For instance, in the machine translation task, it's common to translate a sentence word by word. In this case, the output sequence could contain a partial translation of the input sequence.

The outputs of the decoder are used to supervise a model. The computed gradients are backpropagated back to the inputs of the encoder, allowing the decoder and encoder to be updated simultaneously. In case some input elements should not be visible by other elements, the corresponding attention weights are set to 0.0. The introduced layers and techniques form together the original transformer architecture, introduced in [17]. The model was developed with natural language processing applications in mind, but can be used in any sequence-to-sequence tasks.

2.4.5. Extensions to the original transformer architecture

Over time, several extensions to the original model were proposed. One of the most successful ones include BERT, introduced in [24]. In contrast to the original transformer architecture, the BERT model works only an input sequence. This makes it

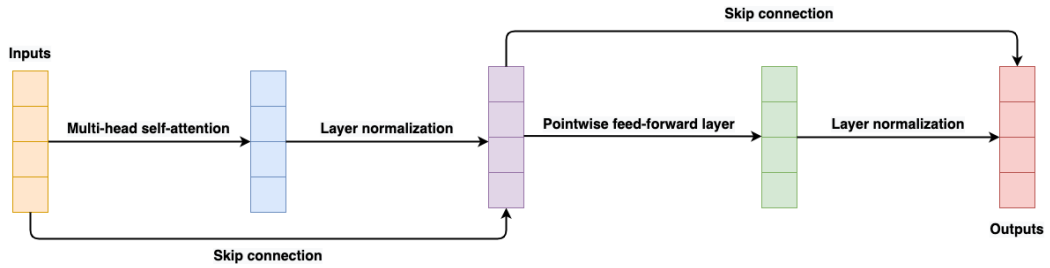

Figure 2.7: An illustration of the transformer layer. Given an input sequence with the encoded positions (marked in orange), the layer first applies the self-attention mechanism to allow the elements to exchange the information. Then, the outputs of the attention layer (marked in blue) are normalized and merged with the original inputs (marked in purple). Afterwards, each element of the sequence is transformed independently with the pointwise feed-forward layer (marked in green). Finally, the elements are normalized and merged with the outputs of the attention layer, forming the output embeddings (marked in red).

better suited for the tasks, in which the whole output sequence is expected to be predicted in one pass. As a result, the BERT model utilizes only the encoder part of the original transformer model. Additionally, the authors proposed to pretrain the proposed model on two tasks: restoring the masked input word and predicting the next sentence. The proposed approach effectively reduces the training time on the downstream task, leveraging the knowledge obtained during the pretraining process. Furthermore, the model was later extended in [26] to share the parameters of transformer layers, making it more parameter-efficient.

Another recently developed extension of the transformer architecture is ELECTRA, introduced in [27]. The authors proposed to train two models in parallel, called generator and discriminator. The generator takes as its input a sequence of elements, with roughly 15% of random tokens masked out. Its goal is to predict the masked tokens, using a few transformer layers followed by softmax. Then, the output distribution is utilised to sample a token for each position, forming a sequence s_G. The constructed sequence s_G is then pushed as an input of the discriminator, which is trained to recognise which tokens are real, considering the remaining ones as fake.

At first, the presented ELECTRA model may seem indistinguishable from Generative Adversarial Networks (GANs), introduced in [3]. However, if one delves into the details, there are several differences. Firstly, in the case ELECTRA, the gradients of the generator and the discriminator are disjoint, causing the information not to flow from one model the other one one. The other distinction is that the generator is trained to generate real samples that are leveraged by the discriminator, unlike in GANs where the discriminator tries to generate fake samples that are pushed as

inputs to the generator. Lastly, after the training procedure is finished, ELECTRA throws out the generator instead of the discriminator. In particular, the whole procedure can be treated as the form of pretraining as the discriminator is further used to train the model on the downstream task.

2.5. Reinforcement learning

Consider a supervised model whose goal is to learn to play a computer game. The model makes some decisions and based on them, the environment provides it new states. In this situation, the model does not know whether it made the right decision. As a result, it is not provided a ground-truth label and cannot be supervised. This type of problem, which involves maximizing the long-time reward based on a sequence of decisions, concerns the area of machine learning called Reinforcement learning. Formally, the goal of this learning method is to find a function π that maximize the so-called utility function

$$U^{\pi}(s) = \mathbb{E}(\sum_{t=0}^{\infty} \gamma^t R(s_t)|\pi, s = s_0)$$

where π is a so-called policy function that defines the actions taken by the model, s_t denotes the t-th state of the game, $R(s_t)$ is a reward function that assigns a reward to a given state and $\gamma \in (0, 1)$ is constant lowering the rewards that are given in the later stage of the game. While the policy π is deterministic, the environment can be nondeterministic (e.g. the provided state can be based not only on our actions, but also on how the other player reacts), thus the expected value is applied. The naive way of finding the optimal policy would be to observe that the optimal utility function is defined by

$$U^{\star}(s) = R(s) + \gamma max_a \sum_{s'} P(s, a, s')U(s')$$

for each $s \in S$, where $P(s, a, s')$ denotes the probability of landing in the state s', assuming that the current state is s and that the action a was taken. These equations are called the Bellman equations and while they are not linear, there exist an iterative algorithm for solving the system of such equations.

In practise, the state space is too large to directly solve the system of Bellman equations. The common solution is to use a function approximating utility function U. This function can be any learnable model and in particular, the most common choice is to use a neural network. In order to learn the approximation function U, one may supervise it by simulating some decisions. Instead of performing the exact simulation, it is common to rely on some approximations that are obtained using Monte Carlo methods. These ideas lead to popular Reinforcement learning algorithms, called Q-learning and Policy Gradients. More details behind the presented methods can be found in [8].

2.6. Models proposed for the graph completion problem

In this section, we will present the related work on the knowledge graph completion problem. The methods will be presented from the simplest context-free ones that are based on convolutional networks to the more advanced graph-context ones based on reinforcement learning or graph neural networks. While in the recent few years hundreds of different methods have been developed, we chose the ones that provided a significant quality boost compared to their predecessors. In parallel, we omit the methods that report inflated quality results due to improper evaluation protocol.

2.6.1. Context-free methods

One of the most influential idea to tackle the problem of knowledge graph completion, was to represent entities and relations by low-dimensional continuous vector representations, called embeddings. One of the most successful models that implemented this idea is TransE, introduced in [1]. The authors proposed to simply start with random representations of entities and relations and iteratively update them by optimizing the model with the following loss function

$$L(h, r, t, h', t') = max(0, ||h + r - t||_2 - ||h' + r - t'||_2 + \gamma)$$

where (h, r, t) represents a (positive) triple that belongs to the knowledge graph, while a (negative) triple (h', r, t') is formed by replacing either h or t with a random entity. Intuitively, this model tries to rotate vectors of entities and relations in a way that the norm of the expression $||h + r - t||_2$ is higher for positive triples than for negative triples. Additionally, the margin parameter γ controls how high the difference between the norms should be in order to have 0.0 loss. Interestingly, the discussed model can be treated as a 1D convolution layer of $D \times 3$ image with fixed $(1, 1, -1)$ filter weights. The TransE model was further extended in [15] to allow relation-specific embeddings of entities. More formally, the authors replaced the expression $||h+r-t||_2$ with $||W_{h,r}h+r-W_{t,r}t||_2$, allowing more flexible representations.

In the recent years, many improvements to the TransE model have been proposed. In particular, several methods applying convolutional layers on top of embeddings have been developed. For instance, in [19] the authors proposed to concatenate the known head/tail entity embedding with the relation embedding, forming $D \times 2$ image. After that, a convolutional layer, followed by a feed-forward projection layer, are applied to obtain the output embedding e_o of dimensionality D. Finally, a dot-product between e_o and each entity $e \in E$ is computed to find the most similar entity to e_o. The model is supervised by a softmax function that is used to maximize the similarity between e_o and the-ground truth tail/head entity that should be predicted. In order to allow the model to distinguish between input and output relations, separate relation embeddings are used for head and tail entities. The model not only optimizes

the weights of intermediate layers, but also finetunes entity and relation embeddings. Additionally, it makes use of Dropout and Label Smoothing to prevent overfitting.

2.6.2. Context-based methods

With the rise of machine learning models operating on graphs, several context-based methods for knowledge graph completion were proposed. S

xx

The other interesting approach was taken in [21]. The authors proposed to learn a model to walk though a graph in order to find a missing head or tail entity. More specifically, given a source entity and a relation, the goal is to take a sequence of decisions that will lead to the target entity, where in each step the model can go to one of its neighbours or remain in the current node. After a fixed number of steps, in case the model lands in the target entity, it is given a reward. Additionally, in order to increase the awareness about the history of visited nodes, it is included in the model's state.

TODO [23] TODO [29]

3. Developed methods

TODO

3.1. TODO

TODO

4. Experiments

TODO

4.1. TODO

TODO

5. Summary

TODO

5.1. The obtained results

TODO

5.2. Future work

TODO

5.3. Acknowledgments

TODO

Bibliography

[1] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, Oksana Yakhnenko - *Translating Embeddings for Modeling Multi-relational Data (2013)*

[2] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, Jeffrey Dean - *Distributed Representations of Words and Phrases and their Compositionality (2013)*

[3] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio - *Generative Adversarial Nets (2014)*

[4] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov - *Dropout: A Simple Way to Prevent Neural Networks from Overfitting (2014)*

[5] Bryan Perozzi, Rami Al-Rfou, Steven Skiena - *DeepWalk: Online Learning of Social Representations (2014)*

[6] Sergey Ioffe, Christian Szegedy - *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (2015)*

[7] Diederik P. Kingma, Jimmy Lei Ba - *Adam: A method for stochastic optimization (2015)*

[8] Richard S. Sutton and Andrew G. Barto - *Reinforcement Learning: An Introduction (2015)*

[9] Jimmy Lei Ba, Jamie Ryan Kiros, Geoffrey E. Hinton - *Layer Normalization (2015)*

[10] Ian Goodfellow, Yoshua Bengio, Aaron Courville - *Deep learning (2016)*

[11] Aditya Grover, Jure Leskovec - *node2vec: Scalable Feature Learning for Networks (2016)*

[12] Zhilin Yang, William W. Cohen, Ruslan Salakhutdinov - *Revisiting Semi-Supervised Learning with Graph Embeddings (2016)*

[13] Daixin Wang, Peng Cui, Wenwu Zhu - *Structural Deep Network Embedding (2016)*

[14] Leonardo F. R. Ribeiro, Pedro H. P. Saverese, Daniel R. Figueiredo - *struc2vec: Learning Node Representations from Structural Identity (2017)*

[15] Dat Quoc Nguyen, Kairit Sirts, Lizhen Qu, Mark Johnson - *STransE: a novel embedding model of entities and relationships in knowledge bases (2017)*

[16] William L. Hamilton, Rex Ying, Jure Leskovec - *Inductive Representation Learning on Large Graphs (2017)*

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin - *Attention Is All You Need (2017)*

[18] Dai Quoc Nguyen, Tu Dinh Nguyen, Dat Quoc Nguyen, Dinh Phung - *A Novel Embedding Model for Knowledge Base Completion Based on Convolutional Neural Network (2018)*

[19] Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, Sebastian Riedel - *Convolutional 2D Knowledge Graph Embeddings (2018)*

[20] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio - *Graph attention networks (2018)*

[21] Rajarshi Das, Shehzaad Dhuliawala, Manzil Zaheer, Luke Vilnis, Ishan Durugkar, Akshay Krishnamurthy, Alex Smola, Andrew McCallum - *Go for a walk and arrive at the answer: reasoning over paths in knowledge bases using reinforcement learning (2018)*

[22] Chao Shang, Yun Tang, Jing Huang, Jinbo Bi, Xiaodong He, Bowen Zhou - *End-to-end Structure-Aware Convolutional Networks for Knowledge Base Completion (2018)*

[23] Chao Shang, Yun Tang, Jing Huang, Jinbo Bi, Xiaodong He, Bowen Zhou - *End-to-end Structure-Aware Convolutional Networks for Knowledge Base Completion (2018)*

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova - *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (2019)*

[25] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, Tie-Yan Liu - *On Layer Normalization in the Transformer Architecture (2020)*

[26] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, Radu Soricut - *ALBERT: a lite BERT for self-supervised learning of language representations (2020)*

[27] Kevin Clark, Minh-Thang Luong, Quoc V. Le, Christopher D. Manning - *Electra: pre-training text encoders as discriminators rather than generators (2020)*

[28] Farahnaz Akrami, Lingbing Guo, Wei Hu, Chengkai Li - *Re-evaluating Embedding-Based Knowledge Graph Completion Methods (2020)*

[29] Shikhar Vashishth, Soumya Sanyal, Vikram Nitin, Partha Talukdar - *Composition-based multi-relational graph convolutional networks (2020)*