University of Amsterdam

MSc Artificial Intelligence

Master Thesis

End-to-end learning of latent edge weights for Graph Convolutional Networks

by

Wolf Bernard Willem Vos

10197923

August 2017

36 EC

February - July 2017

Supervisors:	*Assessor:*
Dr P. Bloem	Dr M. W. van Someren
Z. Sun (ING)	
Dr A.L. De Souza (ING)	
Dr Ir F.M. Jansen (ING)	

ING

UNIVERSITY OF AMSTERDAM

Abstract

We present Latent-Graph Convolutional Networks (L-GCN), an approach for machine learning on any kind of graph structure, including directed graphs, multi-graphs and knowledge graphs. Our approach extends Graph Convolutional Networks (Kipf and Welling, 2016) by allowing for end-to-end training and by supporting any kind of data available on the edges in the network, such as numerous transactions within a company network, different relations between entities in a knowledge graph or different flights between airports. The edge features are mapped to latent graphs which are in turn used in the GCN's localpooling operation. Taking this approach allows us to use the predictive power that is contained within the edges of the graph. We achieve competitive results on four graph datasets (CORA, RITA, AIFB and NELL).

Contents

Definitions

Symbol	Definition	Type
$\mathcal{G}$	graph network	$(\mathcal{V}, \mathcal{E})$
$\mathcal{V}$	set of nodes in graph network	$v_{1,2,\ldots,N}$
$\mathcal{E}$	set of edges in graph network	(v_i, v_j)
N	number of nodes in a graph network	$\mathbb{N}$
N_e	number of edges in a graph network	$\mathbb{N}$
C	number of node features	$\mathbb{N}$
C_e	number of edge features	$\mathbb{N}$
X	node features	$\mathbb{R}^{N \times C}$
E	edge features	$\mathbb{R}^{E \times C_e}$
A	adjacency matrix	$\mathbb{R}^{N \times N}_{\geq 0}$
$\hat{A}$	normalized adjacency matrix	$\mathbb{R}^{N \times N}_{\geq 0}$
$\dot{A}$	latent-graph tensor (contains latent adjacency matrices)	$\mathbb{R}^{L \times N \times N}_{\geq 0}$
D	diagonal degree matrix of an adjacency matrix	$\mathbb{R}^{N \times N}_{\geq 0}$
W	weight matrix	$\mathbb{R}^{C \times F}$
b	bias vector	$\mathbb{R}^{F}$
w	weight of an edge	$\mathbb{R}_{\geq 0}$
r	relation vector	$\mathbb{R} \cap [0, 1]$
$\mathcal{R}$	types of relations	r
L	number of latent-graphs	$\mathbb{N}$
l	layer or latent-graph index in a model	$\mathbb{N}$
$H^{(l)}$	output matrix of layer l	$\mathbb{R}$
$h_i^{(l)}$	output vector of layer l for node i	$\mathbb{R}$
Z	predictions	$\mathbb{R} \cap [0, 1]$
F	number of outputs	$\mathbb{N}$
Y	labels	$\mathbb{N} \cap [0, 1]$
I	identity matrix	$\mathbb{N} \cap [0, 1]$
$\mathcal{N}_i$	set of neighbours for node i	v
B	number of shared weight matrices in Schlichtkrull et al. (2017)	$\mathbb{N}$
V_{rb}	weight matrix for relation r and shared weight matrix $b \in B$	$\mathbb{R}^{C \times F}$
a_{rb}	weight for shared weight matrix b and relation r	$\mathbb{R}$

Square brackets containing scalars $\begin{bmatrix} a & b & c \end{bmatrix}$ denote a vector. Square brackets containing matrices $\begin{bmatrix} X_1 & X_2 & X_3 \end{bmatrix}$ denote matrix concatenation.

Chapter 1

Introduction

Recently, deep learning algorithms have been used to tackle many problems with great precision and accuracy. The power of deep learning is used in several fields such as image classification, image generation, speech recognition/synthesis and other classification or regression tasks (LeCun et al., 2015). The datasets used for these kinds of tasks are structured or unstructured datasets, e.g. transaction data from a bank or images/sound, respectively. These types of datasets are generally composed of unconnected data points (i.i.d.). Another type of data, graph structured data, is seldom used to its full potential within the field of deep learning.

In Kipf and Welling (2016), the Graph Convolutional Network (GCN) is introduced. This is a deep learning algorithm designed to operate on graph structured datasets in an end-to-end setting. Graph structured datasets comprise two elements: a set of nodes $\mathcal{V}$, which are also called vertices, and a list of connections between nodes, called edges $\mathcal{E}$. Together these form the graph $\mathcal{G}$.

The power behind the GCN model is that unlike traditional deep learning methods this approach also takes the state of the neighbouring nodes into account when making a prediction. This is done by pooling the features of the neighbours at every step. In this current research the main focus lies in improving the performance of the GCN algorithm, and specifically its performance on graphs where the edges contain valuable information. This type of edge information is completely unused in the GCN algorithm by Kipf and Welling (2016).

Graph networks and their contents are formally defined as graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with nodes $v_i \in \mathcal{V}$ and edges $(v_i, v_j) \in \mathcal{E}$, where v_i and v_j are two nodes that are connected in the network. These kind of graphs are called **undirected** graphs, where the edge has no direction because it describes a symmetric relation. An example of such a graph is a

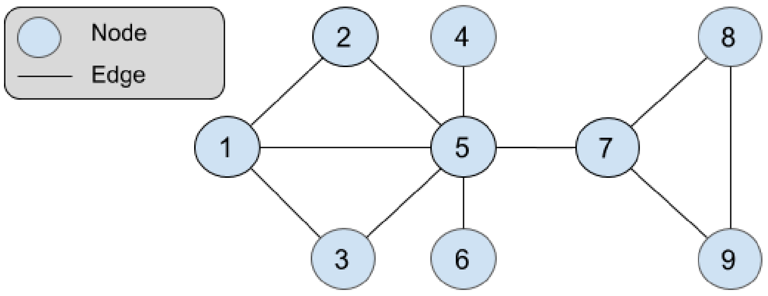

FIGURE 1.1: Example of an undirected graph with 9 nodes and 11 edges

network of colleagues, i.e., if person A works together with person B, person B also works together with person A. (figure 1.1).

A more informative type of graph is a **directed** graph, in this case the directions of the edges are known (figure 1.2). A citation network is an example of such a graph which contains scientific papers as nodes and citations as edges. In such a network article A can cite article B without article B citing article A. In this case the edges $(v_i, v_j) \in \mathcal{E}$ are ordered pairs, where v_i is the originating node and v_j is the receiving node.

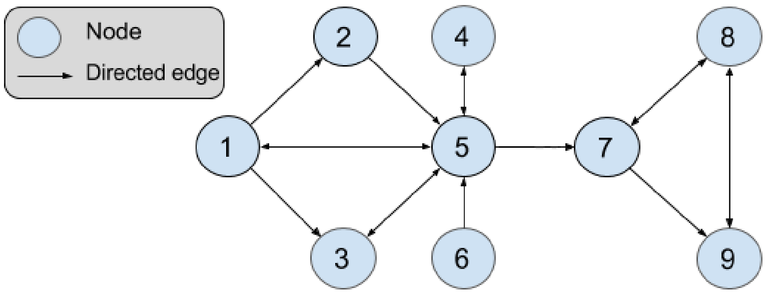

FIGURE 1.2: Example of a directed graph with 9 nodes and 11 directed edges

Both directed and undirected graphs can also be **weighted**, meaning that each edge has a positive value denoting the strength or weight of the connection. In the example of the network of colleagues, this can be the amount of years two persons have been colleagues. For the citation network this can be the amount of references to the other article, more references resulting in a stronger connection between the two articles. In the case of a weighted graph network, the formal definition of the edges changes slightly to the following form: $(v_i, w, v_j) \in \mathcal{E}$, where w is the weight of the edge (figure 1.3).

In the aforementioned types of graphs, only one type of edge can be described, i.e., if two people have worked together in the colleague network or if an article is cited by another paper. In a more sophisticated type of graph network, called a **knowledge graph**, multiple types of edges can be described (figure 1.4). As an example, we could enrich the graph network about colleagues with edges that describe friendship. Supporting multiple types of edges results in the following description of the edges $(v_i, r, v_j) \in \mathcal{E}$,

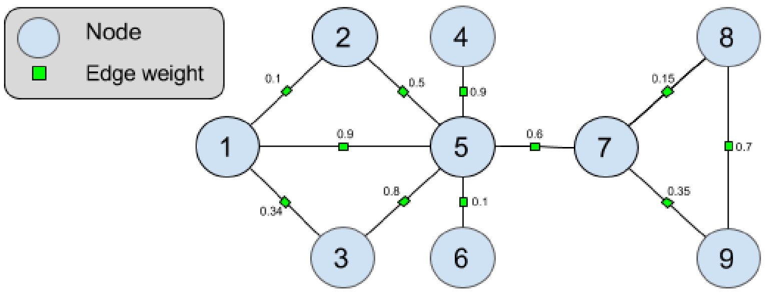

FIGURE 1.3: Example of a weighted graph with 9 nodes 11 weighted edges

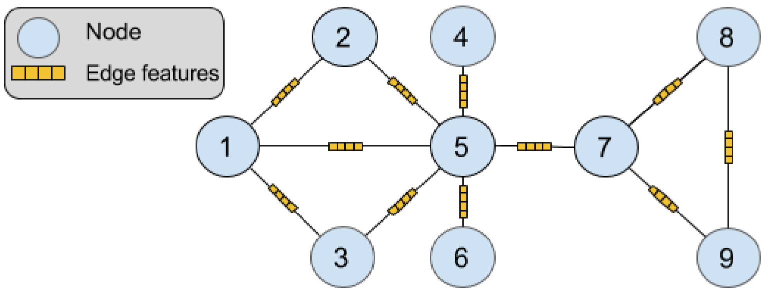

FIGURE 1.4: Example of a knowledge graph with 9 nodes and 11 edges with 4 edge features or types of relations per edge

where r denotes a vector with binary values denoting the absence or presence of a type of edge, also called a relation. For this example, if $r = [1, 1]$ then v_i and v_j are both colleagues and friends, if $r = [0, 1]$ it is only known that they are friends.

In the last type of graph network, it can occur that two nodes have multiple edges of the same type, but with different characteristics (figure 1.5). For instance, if we look at a network of companies that are connected through their payment transactions, the companies can have any number of transactions. Some companies have thousands of transactions between each other while other companies may only have one (figure 1.6). Additionally, each transaction can have different characteristics. The formulation of the edges changes slightly again: $(v_i, E_{ij}, v_j) \in \mathcal{E}$, where E_{ij} is a vector containing the characteristics of the edge, and we refer to these characteristics as edge features. These edge features can be anything that represent the nature of the edge connecting the two nodes v_i and v_j. We will refer to this kind of graph as a **feature multi-graph**.

The GCN model only supports undirected or weighted graphs, but an approach presented by Schlichtkrull et al. (2017) extends the GCN model in such a way that it also supports directed and knowledge graphs, this approach is called the Relational-Graph Convolutional Network (R-GCN). Our approach is similar to the R-GCN model but it also supports feature multi-graphs, thus supporting all types of graph networks.

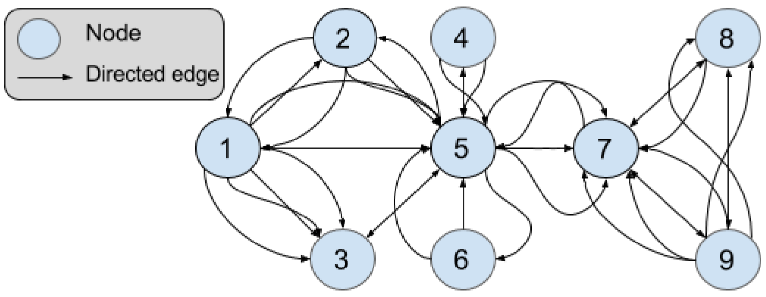

FIGURE 1.5: Example of a feature multi graph with 9 nodes and multiple edges per node pair, edge features are omitted due to readability.

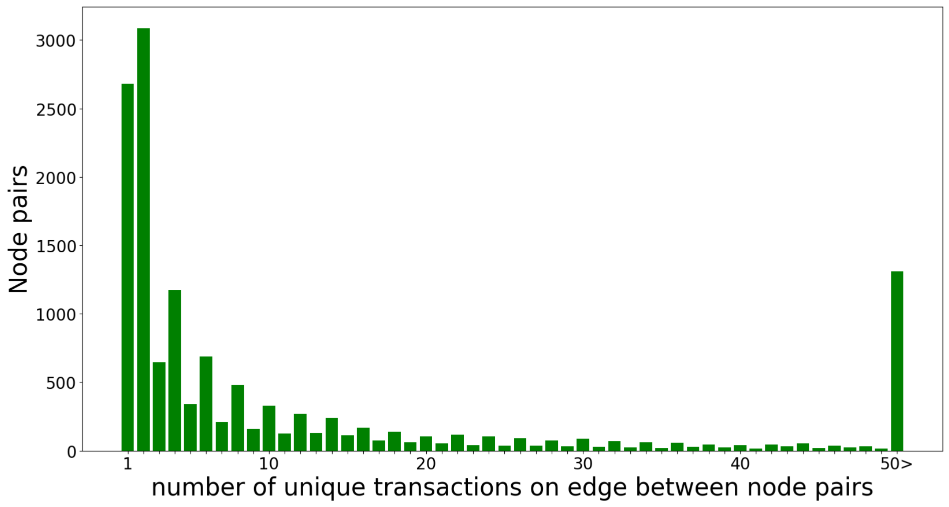

FIGURE 1.6: Distribution of transactions between companies on a subselection of companies in Haarlem, the Netherlands. All the node pairs that have 50 of more transactions are grouped. The maximum amount of transactions between two companies was 4126 in this example.

To achieve this we introduce the Latent-Graph Convolutional Network (L-GCN), an end-to-end deep learning method that, next to the node features in the original GCN, is capable of using the predictive power of the edge features within a graph network. It does this by translating edge features into a weight that can be used in a simple weighted graph, i.e., we learn a function $f(E_{ij}) = w$ that determines the weight w of the connection between node i and j based on the corresponding edge features E_{ij}. In the simplest case, function f returns a single value per edge, which can be directly interpreted as an edge weight. This means that the outputs of function f, which are the weights in the latent graph, determine which neighbours are important. In contrast to the plain GCN model in which each neighbour is regarded as equally important.

If we increase the output size of function f to L as such: $f(E_{ij}) = [w_1, w_2, \cdots, w_L]$, we allow the model to learn L different weights for L different relations in a latent space.

Next to serving as edge weights, these weights also serve as a latent representation of the relation between the nodes. We will refer to this set of weights as a **latent relation**.

Our intuition is that similar relations are also similar in the latent space. Striking a parallel between the semantics of different relations in network theory and the semantics of different words in Natural Language Processing (NLP). The latent relation can thus be interpreted as an embedding, just like embeddings are used in NLP (Mikolov et al., 2013; Le and Mikolov, 2014).

With the ability to translate any type of edge data to a weight or a latent relation, we allow the L-GCN algorithm use all the available data within graph networks. In this work, the improved algorithm achieves state-of-the-art results on the AIFB datasets and competitive results on the other four datasets used: CORA, RITA, MUTAG and NELL (section 4.1).

Since this research was conducted at ING bank, the initial goal was to apply the GCN on a transaction dataset containing companies as nodes and transactions as edges. Unfortunately this dataset was not as suitable as we expected because of some privacy and hardware issues. Additionally, the main goal at ING was to do peer detection, a task that can be performed with GCNs but it is different from the research that has been done during this work. Next to that, the peer detection problem is unsupervised and therefore it is hard to measure the performance without a labeled dataset. Although we got a lot of inspiration from the ING dataset, we chose to omit it and focus solely on the L-GCN approach for the other datasets.

In chapter 2 we will first go into depth on Graph Convolutional Networks, Relational-Graph Convolutional Networks, embeddings in NLP, and other graph based algorithms. The methods used and their implementations are described in chapter 3. In chapter 4, the experiments and results are described. Finally, in chapter 5 we will review the L-GCN algorithm, discuss its strengths and weaknesses, and indicate where it can be used successfully.

Chapter 2

Related Work & Background

2.1 Graph Convolutional Network

In the Graph Convolutional Network (GCN) algorithm (Kipf and Welling, 2016), a layer in the network is formulated as shown in equation 2.1 and figure 2.1.

$$H = \sigma(\hat{A}XW) \tag{2.1}$$

Where X $\in \mathbb{R}^{N \times C}$ is the input to a layer with C node features (or input channels) and N nodes. $\hat{A} \in \mathbb{R}^{N \times N}$ is the normalized adjacency matrix (see section 2.1.1 for the normalization method). $W \in \mathbb{R}^{C \times F}$ is the weight matrix and σ is an element-wise activation function, both of which are incorporated in each layer of the GCN, similar to the weight matrices and activation functions in regular, fully connected neural network layers. F are the number of outputs of that layer. We will refer to the first operation: $\hat{A}X$ as the *localpooling* operation. This name is logical since this operation combines and averages the features of the first-order neighbourhood. The second operation XW is identical to the operation that can be found within a fully connected layer in a neural network: it takes linear combinations of the feature values in X, and thus each set of

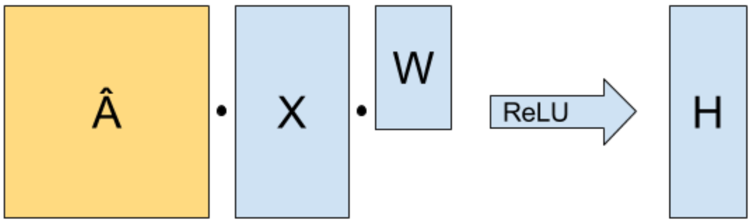

FIGURE 2.1: Schematic modular depiction of one GCN layer with a ReLU activation function

node features is treated in the same way because W is shared across all nodes. The W matrix contains trainable parameters which are usually learned by a gradient descent algorithm such as Adam (Kingma and Ba, 2014).

All edges are captured in the adjacency matrix, making it impossible to have multiple edges between two nodes. Alternatively, it is possible to weight the edges. This can be done by using the number of different edges between node pairs, thus neighbours with many edges are deemed more important in the *localpooling* operation. But this might not always be ideal since edges can have different meanings. In the example of the transaction dataset, it is difficult to determine if having one transaction of 10.000 euros is more important than having ten transactions of five euros. You could prefer one over the other, but in any case, these solutions are still hand-crafted and not data-driven, and thus not ideal.

In Kipf and Welling's experiments, a simple two-layer GCN architecture (equation 2.2) is used to do semi-supervised multi-class node classification. During the training phase of the classification task, the full network structure is used, but only a small percentage of the nodes' labels are used. A cross entropy error $\mathcal{L}$ over all labeled examples $\mathcal{Y}_L$ is used as a loss function (equation 2.5). The activation functions used by the authors are the softmax and the rectified linear unit (ReLU), which are described in equation 2.3 and 2.4, respectively. The softmax is used for the last layer because it is a classification task, the softmax transforms the output Z to the probabilities for the F classes as follows

$$Z = \text{softmax}\Big(\hat{A}\ \text{ReLU}\Big(\hat{A}XW^{(1)}\Big)W^{(2)}\Big), \tag{2.2}$$

where

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}, \tag{2.3}$$

and

$$\text{ReLU}(x) = \max(0, x). \tag{2.4}$$

The cross entropy loss is defined as

$$\mathcal{L} = -\sum_{l\in\mathcal{Y}_L}\sum_{f=1}^{F} Y_{lf}\ln Z_{lf}, \tag{2.5}$$

where Y are one-hot vectors of the known labels for each node, functioning as a selector.

Using this simple but elegant model, Kipf and Welling (2016) manage to achieve state-of-the-art performance in semi-supervised classification on several datasets (e.g. Citeseer, Cora, Pubmed and NELL), surpassing popular methods such as Planetoid (Yang et al., 2016) and DeepWalk (Perozzi et al., 2014).

Kipf and Welling (2016) list three main limitations of the GCN algorithm: memory requirements, limiting assumptions, and the lack of support for directed edges or edge features. The implementation currently uses full-batch gradient descent, since mini-batching is complicated by the fact that nodes can be connected to any other node. Thus, all nodes should be in the same batch, which requires large amounts of memory. During the design of the model some limiting assumptions are made. The most important assumption is on the importance of the self-links. In the GCN, a node's own features are considered equally important as the features of a neighbouring node, which might not always be the case. To solve this, Kipf and Welling (2016) propose a trainable parameter λ that scales the importance of self-links: $\tilde{A} = A + \lambda I_N$. The final noted limitation is the lack of support for edge features or directed edges. This means that all graphs that are used should be undirected. Partial support for these types of edges is added in succeeding work (Schlichtkrull et al., 2017), described in section 2.2.

2.1.1 Adjacency matrix normalization

The *localpooling* operation mentioned in section 2.1 allows the model to learn and predict characteristics about edge or node features with the additional knowledge of the neighbourhood. Unfortunately, multiplying an adjacency matrix A containing binary elements with node features X, and thus naively taking the sum of all the features of the neighbourhood yields poor results and makes training impossible in some cases. This is due to several limitations of neural networks.

First of all, the fact that nodes in the network have different numbers of neighbours/edges results in higher values for well connected nodes and lower values for nodes with a low number of connections. This can make it hard for the network to distinguish between nodes in smaller neighbourhoods. Especially in scale-free networks it can be hard to retain comparable activations of different nodes.

Secondly, after summing multiple node feature vectors, the result generally does not retain a mean close to zero ($\mu = 0.0$) and a variance close to one ($\sigma = 1.0$), which can yield numerical instabilities or vanishing/exploding gradients during the back-propagation procedure which in turn halts the training process.

A solution inspired by spectral graph theory, is that the adjacency matrix should be normalized in the following way: $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ where $D = D_{ii} = \sum_j A_{ij}$ is the diagonal degree matrix of A (Defferrard et al., 2016). This normalization solves the aforementioned problems, but the algorithm does not take a node's own features into account when performing the *localpooling* operation.

In order to retain a node's own features, an identity matrix I_N, which is of the same shape as the original adjacency matrix A, is added to the normalized adjacency matrix: $I_N + D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$. The augmented adjacency matrix combines the node's own features and the features of nodes in the first order neighbourhood of that node. The problem, as the authors state, is that this augmented adjacency matrix has eigenvalues in the range of $[0, 2]$, which can also result in numerical instabilities or vanishing/exploding gradients if applied repeatedly. Therefore, they propose a renormalization trick (Kipf and Welling, 2016): $I_N + D^{-\frac{1}{2}}AD^{-\frac{1}{2}} \rightarrow \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ with $\tilde{A} = A + I_N$ and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$. $\tilde{D}$ is basically the degree matrix of $\tilde{A}$ but the self links are not counted twice, as in a regular degree matrix. In practice this renormalization boils down to a row-wise and column-wise normalization. As in the original paper, we will denote the renormalized adjacency matrix as $\hat{A}$. In short, you add self-links and normalize the adjacency matrix in the same way.

2.2 Relational-Graph Convolutional Network

As stated in chapter 1, in more recent work the GCN algorithm is extended to support directed edges and multiple types of relations (Schlichtkrull et al., 2017). This allows the GCN model to be applied on knowledge graphs. A knowledge graph consists of edge tuples (v_i, r, v_j). In this work the formula for each layer changes so that each different type of relation (and its inverse) has its own adjacency matrix A_r (i.e. for in-going and out-going edges) and its own weight matrix W_r. The authors denote the activation of the hidden layers as

$$H_i^{(l+1)} = \sigma\left(\sum_{r\in\mathcal{R}}\sum_{j\in\mathcal{N}_i^r}\frac{1}{c_{i,r}}W_r^{(l)}h_j^{(l)} + W_0^{(l)}h_i^{(l)}\right), \tag{2.6}$$

where h is the output of the previous layer (or input if it is the first layer) for one node, $\mathcal{N}_i^r$ is the set of neighbours of node i for relation r and l denotes the layer number in the neural network. In this case W_0 is the weight matrix for the nodes' own features and $W_r^{(l)}$ is a weight matrix for that type of relation $r \in \mathcal{R}$. $c_{i,r}$ is a problem specific normalization constant. The notation of this formula does not include an adjacency matrix A as before but the summations function in a similar way. σ denotes a non-linear element-wise activation function such as a ReLU or sigmoid. Because the computational complexity increases drastically when the number of types of relations increase, Schlichtkrull et al. (2017) reduce the number of weight matrices to a smaller number B. These B weight matrices $V_{0,1,\ldots,B}$ are trained and linearly combined via trainable parameters a_{rb},

$$W_r^{(l)} = \sum_{b=1}^{B} a_{rb}^{(l)} V_b^{(l)}. \tag{2.7}$$

Using this generalization, each relation r has the same weight matrices but it can have a different linear combination of the B matrices. This reduces the computational complexity and alleviates some memory issues, additionally, Schlichtkrull et al. (2017) claim that this reduces overfitting on rare relations and can improve performance on classification and prediction tasks. The R-GCN algorithm achieves state-of-the-art performance on some datasets but lags behind on others. In this approach the amount of trainable parameters are reduced during the second operation, and thus the amount of weight matrices that need to be computed are reduced. In our approach, we seek to reduce the amount of trainable parameters in an earlier stage of the pipeline. Namely, before the *localpooling* operation, effectively reducing the number of adjacency and weight matrices. Next to the R-GCN, the inspiration for this approach comes from embedding methods in NLP.

2.3 Embeddings in natural language processing

Within the field of natural language processing (NLP), creating a low dimensional dense vector representation of a word and its semantics has been around for quite a while (Dumais, 2004; Blei et al., 2003). With the increased popularity of deep learning and neural networks in general, these embeddings, also called latent representations, are generally learned by using a (deep) neural network. In the Word2vec approach, presented by Mikolov et al. (2013), a linear embedding is learned for each word in a corpus.

The goal of learning an embedding or latent representation is to translate a high dimensional sparse vector into a low dimensional dense vector whilst keeping data points that are close together in the feature space also close together in the latent space. In the example of word embeddings such as Word2vec (Mikolov et al., 2013), words that are semantically similar ought to be close-by in the latent space. A prime example is the king/queen example within word embeddings. In this case the semantic meaning of king and queen are similar, yet not the same. King is male and queen is female, but one can argue that if you remove the male part from a king, and add a female part to it, a queen emerges. This is exactly how a well-trained word embedding functions.

$$
\begin{aligned}
\text{king} &= \begin{bmatrix} 0.2 & 0.4 & 0.9 & 0.7 \end{bmatrix} \\
\text{male} &= \begin{bmatrix} 0.1 & 0.3 & 0.0 & 0.4 \end{bmatrix} \\
\text{female} &= \begin{bmatrix} 0.2 & 0.7 & 0.0 & 0.3 \end{bmatrix} \\
\text{queen} &= \begin{bmatrix} 0.3 & 0.8 & 0.9 & 0.6 \end{bmatrix} \\
\text{queen} &\approx \text{king} - \text{male} + \text{female}
\end{aligned}
\tag{2.8}
$$

These embeddings provide the machine learner, which is generally a neural network, with an understanding of the words in a more semantic way. This aids almost all classification, generation, and translation tasks within the field of NLP. Traditionally, words would be represented as a vector of the length of the total corpus with a one on the corresponding index, a so-called one-hot vector. This is also the case for the different types of relations within the R-GCN model, and thus, semantically similar relations like $\{A, \text{is\_friend}, B\}$ and $\{A, \text{is\_colleague}, B\}$ are currently treated as two completely different entities in Schlichtkrull et al. (2017). Our intuition is that relation embeddings can improve performance on several tasks, just like word embeddings improved the performance of numerous tasks in the NLP domain.

2.4 Other graph based algorithms

In this section we will discuss algorithms that aim to solve similar tasks to the (R-)GCN algorithms, i.e., machine learning tasks that specialize in graph structured datasets.

2.4.1 DeepWalk

The DeepWalk algorithm (Perozzi et al., 2014) is a graph embedding algorithm. This algorithm is designed to create node embeddings for nodes in a graph network. The embeddings are created by taking n random walks from the center node and applying the skipgram algorithm (Mikolov et al., 2013) on the walks. This boils down to predicting the probability of the walk given the root node. The GCN outperforms DeepWalk by a large margin in all experiments performed by Kipf and Welling (2016). These experiments consist of a semi-supervised classification task on four datasets (CiteSeer, CORA, Pubmeb and NELL). Our approach differs greatly from DeepWalk as our approach supports multiple relations, edge features, and multi-graphs.

2.4.2 RDF2Vec

RDF2Vec (Ristoski and Paulheim, 2016) is an approach similar to DeepWalk that aims to create graph embeddings from a Resource Description Framework (RDF). In contrast to DeepWalk, Ristoski and Paulheim (2016) use two different strategies to extract subgraphs: 1. Graph walks and 2. Weisfeiler-Lehman subtree RDF graph kernels. The former is similar to the approach in DeepWalk, but the authors note that their main difference is that their work uses a directed network instead of an undirected network. The latter is, as noted by the authors, similar to an approach called Deep Graph Kernels (Yanardag and Vishwanathan, 2015), which has some similarities to GCNs.

2.4.3 GraphSAGE

Another graph embedding approach is called GraphSAGE (Hamilton et al., 2017). The authors propose a method for inductive node embedding where node features are used. In their approach, a node's neighbourhood is aggregated by a set of trainable aggregator functions. Their approach generalizes well since the aggregator functions can be applied to unseen nodes and neighbourhoods at predicting time, producing an embedding for the node and it's context.

The set of aggregator functions comprises three architectures: The first function, the mean aggregator, takes the elementwise mean of the neighbourhood's features, which is similar to the GCN's *localpooling* operation. The second function, the LSTM (Long-Short Term Memory network) aggregator, can yield better performance, but the drawback is that it is not permutation invariant. This can be a problem, since edges are generally not ordered or sequential. The last function, the pooling aggregator, is also similar to the GCN's *localpooling* operation, only that the pooling is done in a slightly different order. In the GCN approach, the neighbourhood is aggregated by mean pooling and afterwards the result is multiplied by the trainable weight matrix w. In contrast, the GraphSAGE approach first applies w to each neighbour individually before max pooling all the neighbours.

The big difference between the aggregator functions and the *localpooling* operation is that all the GraphSAGE aggregators are applied on a sub-sample of the graph (e.g. n neighbours from the first order neighbours and n neighbours from the second order neighbours), resulting in lost information. One can argue that this can be beneficial at training time, since it is similar to node dropout, which is used in Berg et al. (2017). At predicting time, this sub-sampling will result in reduced performance.

In the experiments performed by Kipf and Welling (2016), generally two GCN layers are used, which is similar to the approach in Hamilton et al. (2017). A simplified interpretation of the approaches shows the similarity.

Let FC denote a fully connected layer, and LP denote a *localpooling* operation. The subscript either denotes the layer number or which type of pooling: mean or max.

$$\text{GCN} : LP_{mean} \rightarrow FC_1 \rightarrow LP_{mean} \rightarrow FC_2 \rightarrow \text{prediction} \tag{2.9}$$

$$\text{GraphSAGE} : FC_1 \rightarrow LP_{max} \rightarrow FC_2 \rightarrow LP_{max} \rightarrow \text{prediction} \tag{2.10}$$

2.4.4 Message Passing Neural Networks

Gilmer et al. (2017) give an extensive overview on the most popular graph bases neural networks, or Message Passing Neural Networks (MPNNs) as they call them (Duvenaud et al., 2015; Li et al., 2015; Battaglia et al., 2016; Kearnes et al., 2016; Schütt et al., 2017; Bruna et al., 2013; Defferrard et al., 2016; Kipf and Welling, 2016). A general framework, the Neural Message Passing framework, and notation is introduced for these kind of algorithms and they also introduce a new model that outperforms all other models on quantum chemistry datasets.

Within the framework, the model is split up into two parts: a message passing phase and a readout phase. The message passing phase, which runs for T time steps (similar to the number of layers in a GCN), is also split up into two parts: a message passing function and a node update function. The message passing function M_t is concerned with passing a message m from one node to another. The node update function U_t is tasked with processing the received messages and updating the current hidden state h of the nodes. The hidden state of node i at the initial time step ($t = 0$) are its node features X_i. The readout phase is tasked with providing a feature vector of the nodes, this can be a probability distribution over classes in a node classification task for example. The message function is defined as

$$m_i^{t+1} = \sum_{j \in \mathcal{N}_i} M_t(h_i^t, h_j^t, E_{ij}), \tag{2.11}$$

where E_{ij} are the edge features on the edge between node i and j, and h_i^t denotes the hidden state of node i at time step t. This hidden state is updated as follows

$$h_i^{t+1} = U_t(h_i^t, m_i^t + 1). \tag{2.12}$$

The message passing function M_t and the node update function U_t have model specific definitions. For the GCN model the message passing function is defined as:

$$M_t(h_i^t, h_j^t) = (\deg(i)\deg(j))^{-\frac{1}{2}} A_{ij} h_j^t, \tag{2.13}$$

and the node update function as

$$U_i^t(h_i^t, m_i^{t+1}) = ReLU(W^t m_i^{t+1}). \tag{2.14}$$

The readout function is similar for every described MPNN and provides the output of the model defined as

$$\hat{y} = R(h_i^T | i \in \mathcal{G}). \tag{2.15}$$

In the GCN model, R is the softmax function.

2.5 End-to-end learning

As is discussed earlier, one of the main advantages of the L-GCN is the end-to-end nature of the model, allowing it to learn from node and edge features. End-to-end models have several advantages over models that solve a specific task and are combined in a later stage. First and foremost, the full model can be optimized from input to output during training time i.e., end-to-end, hence the name. This is done by creating one differentiable function that maps the input, in our case a network that contains nodes and edges, to a node label. During training time, the error propagation can flow from output to input, effectively minimizing errors in every stage of the pipeline. An intermediary step (in a non-end-to-end model) is for example the classification of the edges, and those edge classes are in turn used in the prediction of the node labels. The classification, or the actual edge classes, might not be useful in classifying the nodes, thus we allow the model to learn a representation of the edges themselves, doing this ensures that the representations of the edges contribute to the classification of the nodes. Secondly, no intermediate labels are necessary to train the separate modules in a non-end-to-end model, i.e., we might not have suitable edge classes available as labels, making supervised training impossible. The end-to-end approach is successfully applied by Bojarski et al. (2016), where they used an end-to-end system to drive an autonomous car, with input from a front-facing camera, and output directly connected to the steering wheel, allowing the model to steer the car based on the input from the front-camera. This model outperforms several systems that are built upon complex subtasks such as road and lane segmentation.

2.6 Software & hardware

This research builds upon code written by Kipf and Welling (2016), especially their Keras GCN implementation.[1] We extend their code and algorithm by using Keras (Chollet et al., 2015) and TensorFlow (Abadi et al., 2015) libraries. The CORA, RITA and NELL experiments are run on a personal laptop with an 4-core Intel(R) Core i7-6700HQ CPU @ 2.60GHz, a NVIDIA(R) GeForce(R) GTX 960M GPU, and 8GB of RAM. The AIFB and MUTAG experiments are run on a 32 $\times$ 8-core Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz, NVIDIA(R) Tesla(R) K80 and a NVIDIA(R) Tesla(R) K40c, and 256GB of RAM machine. For the AIFB experiments, the GPUs are used, for the MUTAG experiment we only use the 32 CPUs.

[1]The original repository can be found under: `https://github.com/tkipf/keras-gcn`

Chapter 3

Method & Approach

3.1 Latent-graph representation

In this chapter we define the Latent-Graph Convolutional Network (L-GCN) model by introducing some extensions to the original GCN model. First, we will explain the model for simple graphs with edge features, and afterwards we will generalize further and show that the model can be applied on graphs with any type of edge, focusing on graphs with multiple edges between node pairs.

As introduced in section 2.1, the GCN model has a simple yet smart way of combining the node features and the network structure through the following formula:

$$H = \text{ReLU}(\hat{A}XW) \tag{3.1}$$

In our proposed model, where edge features, multiple edges, and latent relations are supported, the formula changes as shown in equation 3.2 and 3.3. We allow the network to learn the values of the different latent adjacency matrices, this is captured in the 3-dimensional sparse tensor $\dot{A} \in \mathbb{R}^{L \times N \times N}$, with L being the length of the latent relation vector and N standing for the number of nodes in the network. As $\dot{A}$ is a 3-dimensional tensor, each layer (the first dimension L) can be seen as a separate adjacency matrix describing one latent-graph. The resulting equation is as follows

$$\dot{A}_{lij} = f(E_{ij})_l \tag{3.2}$$

$$H = \text{ReLU}\Big(\dot{A}XW\Big), \tag{3.3}$$

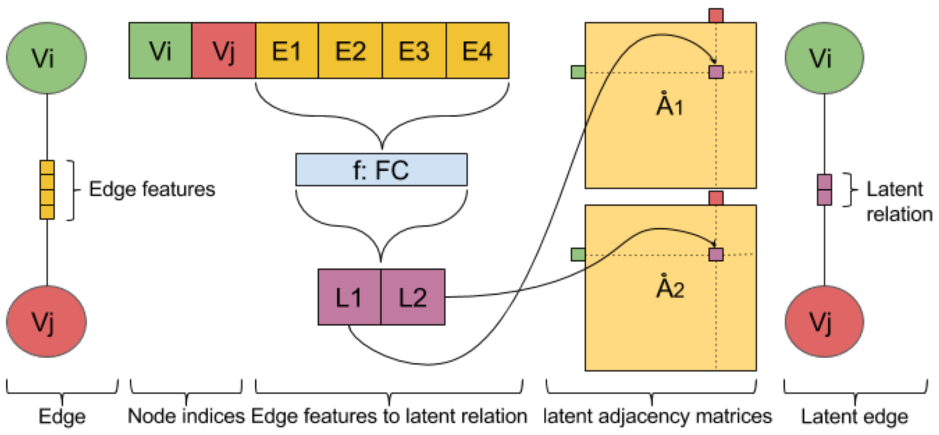

FIGURE 3.1: A schematic depiction of the mapping function from four edge features $E_{1,2,3,4}$ between node v_i and v_j to two latent graphs $\dot{A}_1$ and $\dot{A}_2$ and the corresponding latent relation. Function f is a fully connected neural network layer denoted as FC.

where $f(E)$ is a differentiable function that maps the edge features E to a latent relation vector of length L. These weights are then placed on respective adjacency matrices at the corresponding index, this is illustrated in figure 3.1. In the simple case where $L = 1$, this results in a single undirected weighted graph. If $L > 1$ this can be interpreted as a set of latent-graphs, together describing latent relations between nodes. Function f only maps existing edges to edges in the latent graphs, i.e. the number of edges is the same before and after function f. The mapping function is equal for every edge, thus the weights in function f are shared across all edges. In the case of a feature multi graph, where multiple edges between a node pair are present, we reduce all the edges between a node pair to one edge in each latent graph.

It is important to note that the output of function f has some restrictions because it will be used as a weight in an adjacency matrix. The restrictions mainly come from the adjacency matrix normalization: $\hat{A} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$ where $D = D_{ii} = \sum_j A_{ij}$.

The first restriction is that the elements in D, the row-wise sum of the latent-graph, cannot be negative. This is because during normalization we take the element-wise square root of D and the square root of a negative number is complex, which cannot be used as a weight in a graph. Intuitively, this also holds because having a negative weight in the *localpooling* operation results in subtracting the features of this neighbour. In some cases (perhaps if all node features are continuous) this might work, but in many cases this will result in ambiguous and unwanted feature vectors.

The second restriction is that the elements in the latent-graph normalizer D cannot be zero. This is the case because we normalize by dividing the adjacency matrix by the elements in D. If an element in D is zero, the result is undefined, which is also unusable as a weight in a graph.

To solve this problem we use a ReLU activation for at least the last layer in function f, this removes the negatives in the adjacency matrix and we also add an epsilon ($\epsilon = 0.01$) to each output of function f. Next to solving numerical issues, adding this epsilon also forces the latent-graph to have the same structure as the original graph. In the extreme case where all latent edges have a weight of 0, these will all become ϵ, thus all edges have the same weights, so the weights will become $1/\mathcal{N}$ (this is similar to the weights of the edges in Kipf and Welling (2016)). If one of the edges is weighted above zero, this edge will receive a weight proportional to ϵ.

3.1.1 RITA example

To develop some intuition into how this model works, we will introduce a small toy example. In RITA dataset, which contains airports as nodes and flights as the edges (section 4.1), the airports have several features and the edges contain aggregated flight statistics between any two airports. Let us assume a simple case where there are three edge features: average delay, average flight time and the total number of flights. These three features describe some kind of relation between the two airports, but using those values as weights for the relation is unlikely to yield good results, because a high delay rate should result in a different relation compared to a high number of flights. If we assume a latent-relation length of one ($L = 1$), we allow the network to learn a function f that maps the edge features to an appropriate edge weight. One can see that if the average delay is high, the weight of the edge should be lower and if the total number of flights is relatively high, the edge should have a higher weight since there is a strong connection between the corresponding airports.

3.2 Function f: from data to latent edge weights

In the case described above, with three edge features that are mapped to a single edge weight in a single latent graph, function f is a fully connected layer as can be found in a regular neural network. Which is described as

$$f(E) = \text{ReLU}(EW + b) \tag{3.4}$$

where $E \in \mathbb{R}^{N_e \times C_e}$ is the input matrix containing the aggregated flights statistics per row, N_e is the number of edges in the network, C_e are the number of edge features ($C_e = 3$ in this case), $W \in \mathbb{R}^{C_e \times L}$ and $b \in \mathbb{R}^{L}$ are a set of trainable parameters that map each row of the input to L weights in the latent relation vector ($L = 1$ in this case).

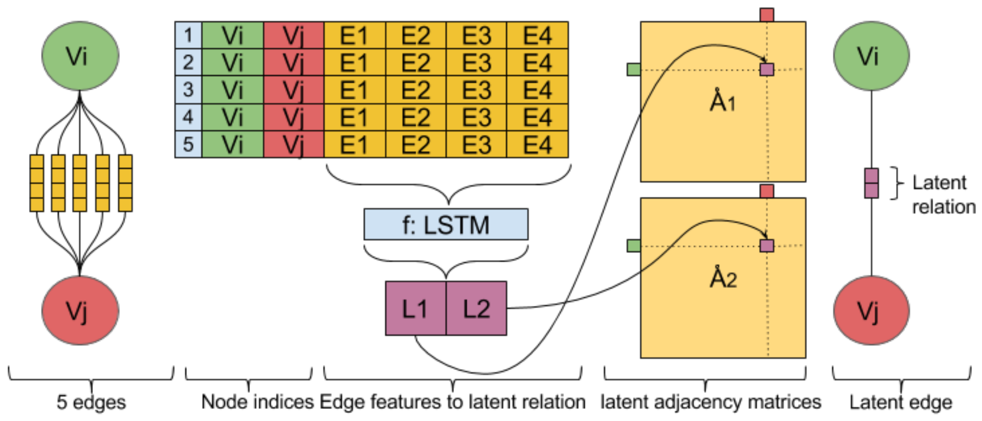

FIGURE 3.2: A schematic depiction of the mapping function from five edges $\{1, 2, 3, 4, 5\}$ with four edge features $E_{1,2,3,4}$ between node v_i and v_j to two latent graphs $\dot{A}_1$ and $\dot{A}_2$ and the corresponding latent relation. Function f is an LSTM neural network layer denoted as LSTM.

This simple example will output a single value or edge weight for each edge based on the three edge features.

In the case of a feature multi graph, such as the original RITA dataset, the edge features are a sequence or set of vectors instead of a single vector, each flight is one edge, this results in thousands of edges between two nodes and each flight has different features. In order to create an end-to-end model where these flights are processed by the model and not aggregated by a hand-crafted function, we need to use something else rather than a fully connected layer. In this case we choose to use an LSTM (Hochreiter and Schmidhuber, 1997) layer as function f to ingest all the raw data points, or flights in this example (figure 3.2). Function f will produce a weight for the latent edge in the latent graph exactly as described earlier. Since the full model is differentiable, the model can be trained end-to-end, which typically yields an improved performance as discussed in section 2.5.

To generalize further, the placeholder function f can be any differentiable function that is suitable for the data that describes the edge. For instance, if a communication network solely relies on images that are sent back and forth between nodes, a convolutional neural network can be used.

3.3 Implementation

As mentioned before, we build on code provided by Kipf and Welling (2016), TensorFlow (Abadi et al., 2015) and Keras (Chollet et al., 2015). In the GCN implementation, sparse matrices are used to implement the algorithm more efficiently and also to alleviate memory issues. These memory issues come from the nature of an adjacency matrix. Adjacency matrices tend to be rather sparse, especially in scale-free networks. When a network contains a thousand nodes, the adjacency matrix, in dense format, contains one million entries denoting if an edge is present between node pairs. In many cases nodes are only connected to a small number of other nodes, and thus the adjacency matrix is storing a lot of information unnecessarily explicit. A sparse matrix only stores non-zero elements, implicitly storing the zero elements. In the example of the CORA dataset, which contains 2708 nodes and 5429 edges, this reduces the number of explicitly stored values from $2708^2 = 7333264$ to 5429. One can easily see that this approach greatly reduces the memory requirement.

In our work we attempt to alleviate the memory issues by using the versatility of the `tf.SparseTensor` class to allow the user to compute the embedding from any kind of data. After preprocessing the data, each edge has an origin and a destination, these are generally encoded as indices in the range $[0, N]$ where N is the total number of nodes. Knowing this allows us to process the edge features in a matrix form rather than a tensor form and carefully placing the resulting edge embedding on the corresponding sparse tensor indices (see Appendix B for an example).

Chapter 4

Experiments & Results

In this chapter we first describe the datasets, architectures and experimental results. Since the L-GCN model is applicable on multiple types of graph networks we split the experiments up in three parts. First, we test the L-GCN model on the feature multi-graph network called RITA. Secondly, we investigate if the L-GCN model is also applicable on undirected graphs without edge features (such as CORA). Lastly, we apply the L-GCN model on two knowledge graphs (AIFB and NELL), the experiment and split for the AIFB dataset is taken directly from Schlichtkrull et al. (2017), where they evaluate the R-GCN model. For the NELL dataset we create our own splits. We evaluate the L-GCN model by accuracy score on a node classification task similar to the experiments in Kipf and Welling (2016) and Schlichtkrull et al. (2017).

4.1 Datasets

4.1.1 RITA: flights dataset

We introduce the RITA dataset, named after the Research and Innovative Technology Administration's Bureau of Transportation Statistics (RITA BTS). The BTS collects data about all transportation in the United States of America. In this dataset, data about flights and airports is combined into a feature multi graph containing node and edge features.

The nodes of the graph network are 283 airports[1] in the United States of America. The edges are all (6965803) flights[2] between those airports during the year 2008. Originally

[1]Airport feature data comes from RITA BTS under: `http://osav-usdot.opendata.arcgis.com/datasets/0e872765538d499a883850e3f2ba0848_0`

[2]Flight data comes from RITA BTS under: `https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236`

the dataset contains 305 airports, but some are removed during preprocessing because the features and/or the target labels of those airports are unknown. The target labels of the airports (or nodes) are their hub-sizes e.g. no-hub, small-hub, medium-hub or large-hub[3].

A full description of the dataset can be found in the appendix A. This dataset is available on Github[4].

4.1.2 CORA: citation network

The CORA dataset (Sen et al., 2008) contains 2708 nodes that each represent a scientific paper. The nodes are connected through citations between the papers, forming a graph network. Each paper belongs to one of seven classes: Neural Networks, Rule Learning, Reinforcement Learning, Probabilistic Methods, Theory, Genetic Algorithms or Case Based. The citation network consists of 5429 edges. The content of the papers and thus their feature vectors is encoded as a binary valued word vector indicating the presence or absence of a word from the dictionary. After preprocessing the dictionary contains 1433 unique words. The goal is to classify each node into one of the seven classes.

4.1.3 AIFB: knowledge graph

The AIFB dataset is a knowledge graph on four research institutes. It contains 45 relations that describe the relations of researchers and their affiliation with the four institutes or the presence of a feature. In this network there are no predefined node features, to solve this problem we will use a one-hot vector describing the identity of a node as node features. This will effectively result in pooling the node features into the hidden state of the node during the first *localpooling* operation, because you pool in the literals that define the nodes. We copy this approach from Schlichtkrull et al. (2017). The goal of this experiment is to classify which of the four institutes each person is affiliated with. This dataset contains 8285 nodes of which 176 are labeled. This dataset has a predefined train and test splut of 80 and 20 percent, respectively.

4.1.4 NELL: knowledge graph

NELL is a knowledge graph first introduced by Yang et al. (2016). It contains entities from wikipedia as nodes and relations that are learned by the Never-Ending Language

[3]Labels are scraped from wikipedia `https://en.wikipedia.org/wiki/List_of_airports_in_the_United_States`

[4]`https://github.com/wbwvos/`

Learning algorithm created by Mitchell et al. (2015). In total the dataset contains 10325 entities that are linked through 64925 edges which all belong to one of the 415 types of relations. The dataset that is used in Yang et al. (2016) is not usable in our case since they remove the connection between two nodes by creating a node for each relation and by concatenating the edge features to the node features. The node features are wikipedia descriptions of that entity in a 5414 long binary vector denoting the absence of presence of the corresponding word. A relation (v_i, r, v_j) is transformed into (v_i, r_1) and (v_j, r_2), thus making it impossible to trace back where the edge is going. We obtained the original data and preprocessed the dataset ourselves. The remaining dataset, where all nodes have features and the edge features are separated from the node features contains 3875 nodes and 7837 edges. The goal in using this dataset to predict to which of the 210 classes each entity belongs.

Dataset	#Nodes	#Edges	#Node features	#Edge features	#Relations	#Classes
RITA	283	5,279	36	20	-	4
CORA	2,708	5,429	1,433	-	-	7
AIFB	8,285	29,043	-	-	45	4
NELL	3,875	7,837	5,414	-	415	210

Table 4.1: Dataset statistics

4.2 Model architectures

In the experiments we compare different model architectures, which are described in the following sections. For all models we use full-batch gradient descent.

4.2.1 Fully connected neural network

The baseline for the experiments is a small neural network (NN) if applicable. Compared to the other models, this architecture does not use the graph structure in any way. All training and predicting is solely performed on the node features. Unless stated otherwise, the output size of the hidden layer is 16.

$$\begin{aligned} H^{(1)} &= XW^{(1)} + b^{(1)} \\ H^{(1)} &= \text{ReLU}(H^{(1)}) \\ H^{(2)} &= H^{(1)}W^{(2)} + b^{(2)} \\ Z &= \text{softmax}(H^{(2)}) \end{aligned} \tag{4.1}$$

4.2.2 Graph Convolutional Network

This model architecture is the Graph Convolutional Network created by Kipf and Welling (2016). The architecture used in this research is similar to the architecture used in their experiments. This architecture makes use of a single undirected symmetrical adjacency matrix, two *localpooling* layers each followed by a fully-connected layer[5]. Unless stated otherwise, the output size of the hidden layer $H^{(1)}$ is 16. In contrast to the original GCN, we do use a bias term.

$$\begin{aligned}
\hat{A} &= \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}\text{(Kipf and Welling, 2016)} \\
H^{(1)} &= \hat{A}XW^{(1)} + b^{(1)} \\
H^{(1)} &= \text{ReLU}(H^{(1)}) \\
H^{(2)} &= \hat{A}H^{(1)}W^{(2)} + b^{(2)} \\
Z &= \text{softmax}(H^{(2)})
\end{aligned} \tag{4.2}$$

4.2.3 GCN with separated self-links adjacency matrix

This model architecture remains unchanged compared to the previous one except for the separation of the self-links. In the original model the adjacency matrix is combined with a diagonal identity matrix, $\tilde{A} = A + I$. This creates a single elegant adjacency matrix, but this can reduce prediction accuracy in some situations. Therefore, we separate the diagonal identity matrix I from the adjacency matrix. Since a dot product of I and X results in X (for square matrices) we remove the identity matrix I from the equation. The output size of the hidden layer $H^{(1)}$ is 16.

$$\begin{aligned}
\hat{A} &= D^{-\frac{1}{2}}AD^{-\frac{1}{2}} \\
H^{(1)} &= \begin{bmatrix}\hat{A}X & X\end{bmatrix} W^{(1)} + b^{(1)} \\
H^{(1)} &= \text{ReLU}(H^{(1)}) \\
H^{(2)} &= \begin{bmatrix}\hat{A}H^{(1)} & H^{(1)}\end{bmatrix} W^{(2)} + b^{(2)} \\
Z &= \text{softmax}(H^{(2)})
\end{aligned} \tag{4.3}$$

Square brackets containing scalars $\begin{bmatrix}a & b & c\end{bmatrix}$ denote a vector. Square brackets containing matrices $\begin{bmatrix}X_1 & X_2 & X_3\end{bmatrix}$ denote matrix concatenation.

[5]In the implementation the *localpooling* and the dense layer are combined into one layer

4.2.4 GCN with normalized edge features as adjacency matrix

In this set-up multiple adjacency matrices are used, their weights are determined by the edge features E. This architecture is similar to the R-GCN model architecture proposed in Schlichtkrull et al. (2017). The only difference is that in our experiments we allow each adjacency matrix to contain edge features as well as different relation types.

Edge features are similar to node features in structure but in this case they describe the nature of the edge between two nodes (e.g. in the RITA dataset each edge has several features, which are the features of the flights from node i to node j and from node j to node i). An edge feature vector can be described as a simple vector $E_{ij} = \begin{bmatrix} e_1 & e_2 & \cdots & e_n \end{bmatrix}$ with n features.

Relations are encoded in a similar fashion as edge features, the only difference is that the edge feature vectors now only contain binary values denoting if the corresponding relation is present or absent. This vector can be described as $E_{ij} = \begin{bmatrix} r_1 & r_2 & \cdots & r_n \end{bmatrix}$ with n types of relations.

It is also possible to combine these two options into an edge feature vector that contains binary values for the different relations and continuous values for the different edge features. In the original GCN, where only undirected single graphs are used, the E_{ij} vectors contain one feature which is a 1 if there is an edge $A_{ij} = E_{ij} = \begin{bmatrix} 1 \end{bmatrix}$ and 0 otherwise $A_{ij} = E_{ij} = \begin{bmatrix} 0 \end{bmatrix}$.

Taking an approach such as this, where edge features are directly used as weights, will increase the number of adjacency matrices by the number of edge features as discussed in section 2.2. Each edge feature e between node i and node j is placed on its corresponding adjacency matrix A_{eij}. Each adjacency matrix is then normalized with the re-normalization trick (Kipf and Welling, 2016). The output size of the hidden layer $H^{(1)}$ is 16.

$$\begin{aligned}
\hat{A}_e &= D_e^{-\frac{1}{2}} A_e D_e^{-\frac{1}{2}} \\
H^{(1)} &= \begin{bmatrix} \hat{A}_1 X & \hat{A}_2 X & \cdots & \hat{A}_n X & X \end{bmatrix} W^{(1)} + b^{(1)} \\
H^{(1)} &= \text{ReLU}(H^{(1)}) \\
H^{(2)} &= \begin{bmatrix} \hat{A}_1 H^{(1)} & \hat{A}_2 H^{(1)} & \cdots & \hat{A}_n H^{(1)} & H^{(1)} \end{bmatrix} W^{(2)} + b^{(2)} \\
Z &= \text{softmax}(H^{(2)})
\end{aligned} \tag{4.4}$$

4.2.5 Latent-Graph Convolutional Network

In the final model architecture, the edge feature vector is mapped to a latent relation as described in 3.1. The latent relation can be described as $E_{ij} = \begin{bmatrix} l_1 & l_2 & \cdots & l_n \end{bmatrix}$ with a latent relation length of n = L. We experiment with several latent relation lengths, the parameters of the best model will be listed at the results. As in the previous experiment, the self-links are added to the model without being modified. We use two different functions for the placeholder function f: 1. a fully connected layer as discussed in section 3.1, and 2. an LSTM layer as discussed in 3.2. The output size of the hidden layer is 16. Each edge feature vector will be mapped to a latent relation in the same way. This model is an end-to-end model, thus weights for the embedding are learned directly within the network from the node classification task. Each latent adjacency matrix A_l is normalized with the same re-normalization trick. In equation 4.5 we show the equation for the fully connected layer, extending this to an LSTM layer is trivial within the Keras framework (Chollet et al., 2015).

$$\begin{aligned} f(E_{ij}) &= \mathrm{ReLU}(E_{ij}W + b) \\ \dot{A}_{lij} &= f(E_{ij})_l \\ \dot{A}_l &= D_l^{-\frac{1}{2}} \dot{A}_l D_l^{-\frac{1}{2}} \\ H^{(1)} &= \begin{bmatrix} \dot{A}_1 X & \dot{A}_2 X & \cdots & \dot{A}_n X & X \end{bmatrix} W^{(1)} + b^{(1)} \\ H^{(1)} &= \mathrm{ReLU}(H^{(1)}) \\ H^{(2)} &= \begin{bmatrix} \dot{A}_1 H^{(1)} & \dot{A}_2 H^{(1)} & \cdots & \dot{A}_n H^{(1)} & H^{(1)} \end{bmatrix} W^{(2)} + b^{(2)} \\ Z &= \mathrm{softmax}(H^{(2)}) \end{aligned} \tag{4.5}$$

4.3 Results

In this section the experimental results are described for the classification tasks on the datasets described in section 4.1.

4.3.1 RITA

The experiments with the RITA dataset serve as a proof of concept where we show the versatility of the model. The L-GCN model outperforms most of the baselines and other configurations by a significant margin. All configurations are run with similar hyper-parameters: a learning rate of 0.01 for the Adam (Kingma and Ba, 2014) optimizer, two *localpooling* operations followed by two fully connected layers. The first one with 16 hidden nodes and the second one with four output nodes, corresponding to the four target labels. A dropout rate of 0.5, l2-regularization with a strength of $5e^{-4}$, and Glorot uniform weight initialization (Glorot and Bengio, 2010) in the two fully connected layers.

The L-GCN-FC architecture uses this set of hyper-parameters, but additionally the L-GCN model has some unique hyper-parameters as well: a latent relation length L of 2, and a single fully connected layer, which maps the edge features E_{ij} to two latent-graphs. The weights in the fully connected layer within function f is initialized with a random uniform initializer and it uses a ReLU activation function.

The L-GCN-LSTM architecture uses exactly the same hyper-parameters as the L-GCN-FC.[6] The only difference is that this model uses an LSTM layer instead of a fully connected layer in function f. In the other models, all the flight data is averaged before before it is used as input to the model. For the L-GCN-LSTM model, the flight data is averaged in twelve bins of one month and used as input in chronological order.

Each architecture is trained on 50% of the RITA dataset and tested on the other 50%, from the train split we use 10% for early stopping[7]. The average test accuracy and standard error is calculated on 10 runs, The standard error is defined as: $SE = \frac{s}{\sqrt{n}}$, where s is the standard deviation over the n runs.

After initial experiments, we found that the NN model outperformed all other models, as can be seen in the second column in table 4.2. After some investigation, we found that some features (e.g., enplanements, arrivals, and passengers) were highly correlated with the target label (hub-size), resulting in the high accuracy score for the plain neural network. Intuitively, our explaination for the big different between the NN and the GCN score is that naively pooling the features from neighbours is not beneficial in this case. The reason being that if a small airport (hub-size=0) only has one connection to another airport, which is a large airport (hub-size=3), the resulting number of enplanements will be somewhere in the middle between the small and the large airport's number. This in turn makes it look like a medium sized airport (hub-size=2). Even though the experiments on the RITA dataset show no increase in performance for the L-GCN model, it does show that it is more robust against these kind of correlated datasets than the GCN model.

Since we introduced the RITA dataset ourselves and we wanted to create a dataset that makes use of the graph structure rather than relying on highly correlated features, we took the liberty to remove some of these features, resulting in the RITA-hard dataset (see

[6] All other settings are the default settings used in Keras.

[7] We assigned 50% of each class to either the training or test set, this is saved in the rita_tts.content file, where the last column denotes which split each sample belongs to (1=train, 2=test)

Appendix A.2 for the cross-correlation matrix we based this decision upon). The results on the RITA-hard dataset are more in line with the expectations, the performance of the NN dropped drastically and the L-GCN model outperforms all the other models, effectively using the graph structure and the edge features.

Model	**RITA**	**RITA-hard**
NN	**76.69** $\pm$ 0.72	57.39 $\pm$ 0.50
GCN	64.08 $\pm$ 1.21	65.00 $\pm$ 1.21
GCN-SSL	73.45 $\pm$ 0.87	67.82 $\pm$ 0.72
GCN-NEF	69.22 $\pm$ 1.79	66.55 $\pm$ 1.61
L-GCN-FC (Ours)	75.49 $\pm$ 0.84	69.44 $\pm$ 1.15
L-GCN-LSTM (Ours)	73.31 $\pm$ 1.35	**69.93** $\pm$ 1.11

TABLE 4.2: Experimental results of different model architectures on the RITA dataset. NN denotes Neural Network described in section 4.2.1. GCN denotes Graph Convolutional Network described in section 4.2.2 (Kipf and Welling, 2016). GCN-SSL denotes GCN-Separated Self Links described in section 4.2.3. GCN-NEF denotes GCN-Normalized Edge Features described in section 4.2.4. L-GCN-FC and L-GCN-LSTM denote L-GCN architectures with a fully connected and LSTM layer for function f, respectively.

4.3.2 CORA

The CORA dataset is used to investigate whether the L-GCN model can also improve the performance on several datasets where no edge features are available. To achieve this we add a two-hot vector to the edge features, this two-hot vector contains zeros except for the two indices of the nodes that are on each side of the edge. The intuition behind this is that some nodes are more discriminative than other nodes, which means that they should be pooled with a higher or lower weight. In the example of the ING transaction graph, where companies are connected to other companies through their transactions, the node representing ING itself is not discriminative at all. This is the case because all nodes are likely to be connected to this node, thus pooling the features of the ING node yields no gain in predictive performance. Unfortunately, we fail to show that this two-hot approach is beneficial for the model.

Our intuition was that this approach failed on the semi-supervised classification task is that a large number of weights in function f for the unseen nodes are still unchanged after the training phase, resulting in random noise for those edges. In an attempt to solve this problem, we experimented with increasing the percentage of seen target labels during training time. As you can see in table 4.3, the performance of the L-GCN-2H increases relative to the GCN accuracy when more target labels are added, but it still fails to outperform it.

Model	**6%**	**20%**	**40%**	**60%**	**80%**
NN	57.49	68.82	73.89	77.13	76.25
GCN	80.57	85.30	86.92	88.23	89.41
L-GCN-2H	77.49	84.47	86.60	87.35	89.41

TABLE 4.3: This table shows the influence of the percentage of the data that is being used during training on the different models on the CORA dataset. We list the average accuracy over 5 runs. The models are trained on random training splits of {6,20,40,60,80} percent, 10% is used for early stopping and remainder for testing. NN denotes Neural Network described in section 4.2.1. GCN denotes Graph Convolutional Network described in section 4.2.2 (Kipf and Welling, 2016). L-GCN-2H denotes the our Latent-Graph Convolutional Network with two-hot vectors as edge features.

In this experiment we use a latent-relation of $L = 1$, which means that we create one latent-graph. The other hyper-parameters are identical to the hyper-parameters used in the RITA experiments.

4.3.3 AIFB

The experiment done on the AIFB dataset is copied directly from the one done by Schlichtkrull et al. (2017). We use their train and test split and we use most of their hyper-parameter choices, which are similar to the hyper-parameters listed in section 4.3.1, allowing us to directly compare our results to theirs. The baselines for the experiment are the results listed in Schlichtkrull et al. (2017). A latent relation of length $L = 2$ is used for the AIFB experiment. We train the model for 50 epochs. We list the average accuracy and standard error over 10 runs, calculated as described in section 4.3.1. The AIFB experiment has a wall-clock run time of approximately 12 seconds per epoch. We achieve state-of-the-art results on the AIFB dataset as shown in table 4.4.

As we mentioned in the introduction, our intuition was that relations that are similar ought to be similar in the latent space. Since we used a latent relation of length 2 we are able to plot the embeddings for the different relations in the AIFB dataset (figure 4.1). The right bottom section contains the relations that are important for latent graph 1. The left top section contains the relations that are important for latent graph 2. The right top section contains relations that are important in both latent graphs, such as *publication*. The left bottom section contains the relations that are considered unimportant for either latent graph. Although interpretation is rather difficult, there is some evidence that latent graph 1 is grouping the people or entities such as *member*, *author* and *financedBy*, whilst latent graph 2 has a preference for literals such as *homepage*, *isAbout*, and *abstract*. It would be interesting to see if adding a l1-normalization to the output of function f will result in more interpretable embeddings where relations are only allowed to be important in one of the latent graphs.

Model	**AIFB**
WL	80.55 ± 0.00
RDF2Vec	88.88 ± 0.00
R-GCN	95.83 ± 0.62
L-GCN-FC (Ours)	**96.94** ± 0.61

TABLE 4.4: Knowledge graph results on AIFB dataset. WL denotes a Weisfeiler-Lehman kernel as described in Shervashidze et al. (2011) and de Vries and de Rooij (2015). RDF2Vec denotes the algorithm introduced by Ristoski and Paulheim (2016). R-GCN is the Relational-Graph Convolutional Network as introduced in Schlichtkrull et al. (2017). L-GCN-FC denotes our Latent-Graph Convolutional Network with a fully connected layer as function f.

4.3.4 NELL

The NELL dataset was first introduced in Yang et al. (2016), but since it was not suitable for our use case we preprocessed the knowledge graph in a slightly different way. We created a 80/20 train/test split[8], similar to the experiments on the other knowledge graphs. We use similar baselines to the baselines in the RITA experiments. In order to examine the difference between the number of weights we also introduce an NN and a GCN configuration with 32 hidden nodes, denoted as NN-32 and GCN-32, respectively. These additional configurations allow us to eliminate the possibility that the difference in test accuracy does not come from the mere increase of weights, but rather the improved model. All models are trained for 50 epochs. Again, we report average accuracy and standard error over 10 runs. We use a latent relation of length $L = 1$.

As you can see in table 4.5, the NN model benefits greatly from the additional weights, for the GCN this is not the case. One explanation can be that the node features, which are a bag-of-words representation of the entity, are fairly good node features for predicting the corresponding target label. By using the GCN's *localpooling* operation, all the features of neighbouring nodes are mixed with the node's own features, resulting in an indistinguishable mixture of features. The GCN-SSL results are supporting this claim, because in this case the node's features are separated from the mixture that is formed by the features of the neighbours. This allows the model to choose if it wants to use the features of the neighbours or not, in this situation is chooses a nice combination because it outperforms the NN-32 model.

The L-GCN model outperforms the GCN model by a large margin. Intuitively, this is the result of selecting which neighbours it is beneficial to pool from, but if we look at the accuracy score for the GCN-SSL model, which has a similar score, perhaps it is

[8]We assigned 80% of each class to the training set and the remainder to the test set, this is saved in the nell_tts.content file, where the last column denotes which split each sample belongs to (1=train, 2=test)

the case that separating the node's own features from the neighbouring features is most important for this dataset.

Model	**Test accuracy**	**Standard error**	**#Parameters**
NN-16	66.93	±0.28	89,224
NN-32	71.88	±0.32	178,296
GCN-16	52.06	±0.31	89,224
GCN-32	49.50	±0.30	178,296
GCN-SSL	77.16	±0.22	178,280
L-GCN-FC (Ours)	**77.55**	±0.22	178,705

Table 4.5: Experimental results of different model architectures on the NELL dataset. NN denotes Neural Network described in section 4.2.1, NN-16 and NN-32 have 16 and 32 hidden nodes, respectively. GCN denotes Graph Convolutional Network described in section 4.2.2 (Kipf and Welling, 2016), GCN-16 and GCN-32 have 16 and 32 hidden nodes, respectively.. GCN-SSL denotes GCN-Separated Self Links described in section 4.2.3. L-GCN-FC denotes Latent-GCN with a fully connected layer as mapping function described in section 4.2.5.

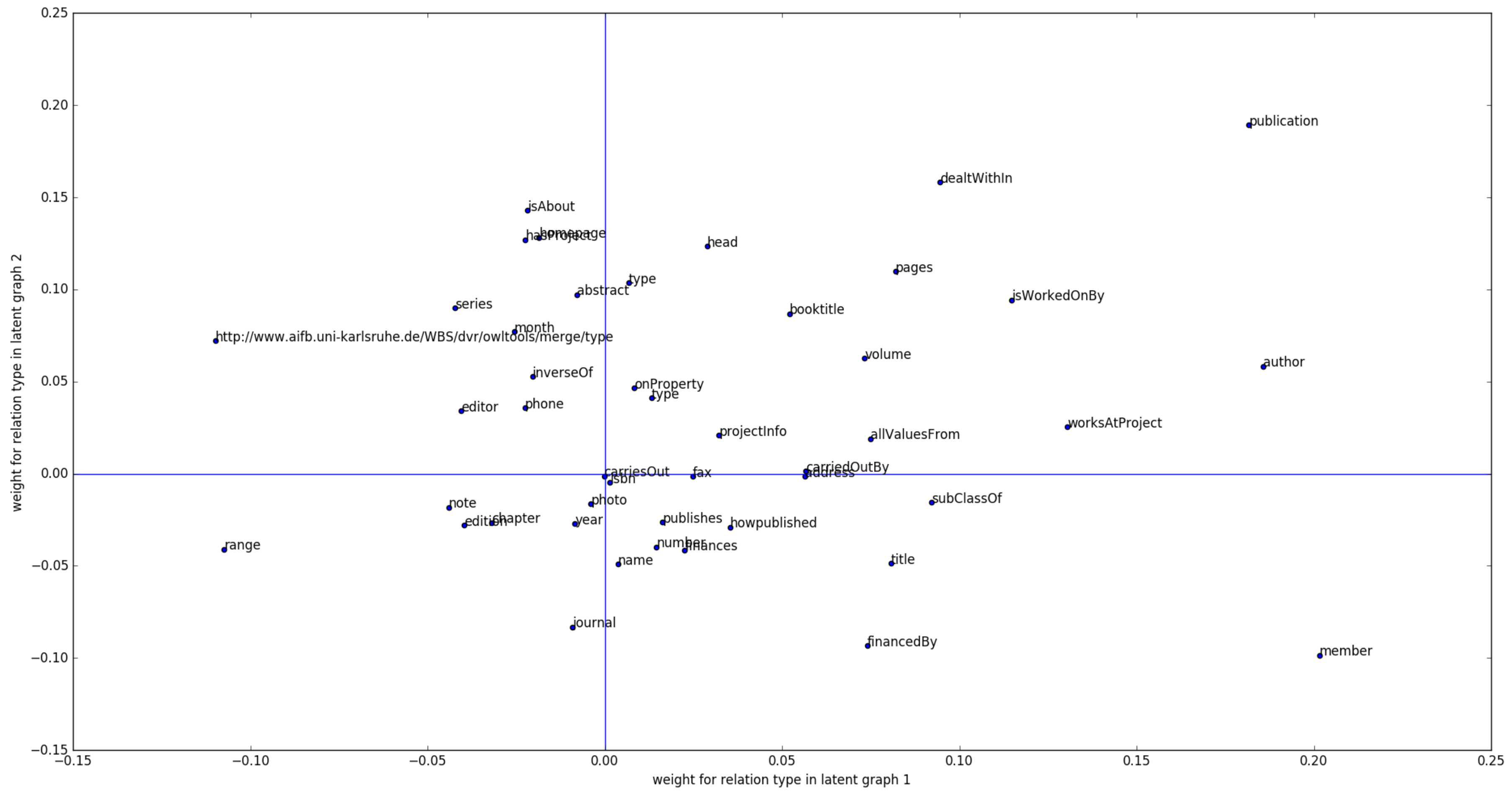

Figure 4.1: Latent relation embeddings for the 45 relations in the AIFB dataset. The blue lines indicate the cut-off enforced by the ReLU in function f.

Chapter 5

Conclusion

In our experiments we have shown that the Latent-Graph Convolutional Network is an interesting extension of the GCN algorithm. We achieve state-of-the-art on the AIFB dataset, surpassing the R-GCN algorithm. Next to that, the L-GCN approach is competitive compared to the GCN and other baselines such as RDF2Vec (Ristoski and Paulheim, 2016) and Weisfeiler-Lehmann kernels (de Vries and de Rooij, 2015; Shervashidze et al., 2011). Our approach is originally designed for feature multi graphs, but as we have shown, it can be successfully applied on any kind of graph that contains any kind of node or edge features.

On the RITA dataset we show the ability to train the network end-to-end, effectively using the edge features to learn a latent-graph. After removing highly correlated node features in the RITA dataset, we see that the L-GCN algorithm outperforms all the other baselines since it has the ability to learn from the graph structure, and especially from the edge features. The L-GCN-LSTM model proves that the L-GCN model can be used as an end-to-end model for learning from node and edge features. In our L-GCN-LSTM experiment we did average edges by hand, but since they are binned by month rather than by year, one can see that it is trivial to extend this to a fully end-to-end model where flights are directly used as input. Additionally, the L-GCN-FC and L-GCN-LSTM models show that function f can be comprised of any differentiable function. It might be interesting to try other types of neural network layers such as a convolutional layer used for image processing. Applications that use feature multi-graph datasets, such as the RITA dataset, can benefit from using the L-GCN model over the original GCN model.

An unexpected outcome was that the L-GCN was unable to improve the accuracy score on the CORA dataset. The L-GCN without the two-hot vector works identical to the GCN algorithm, thus it can still be used for datasets such as CORA, but it yields no performance increase compared to the original GCN. Intuitively, adding the two-hot vector makes sense, but if we look at how the weights in function f are trained, it is logical that it does not work as intended because if weights are never updated in function f, they only introduce random noise. Further research is needed to see if this never works, or that the problem of random noise can be solved by different weight initialization or masking methods.

On the AIFB dataset we achieve state-of-the-art performance, making this model interesting for knowledge graphs. On the other hand, the AIFB dataset might not be the best dataset to test on, because it has a very small number of labeled nodes. The latent relation embeddings are interesting to look at but interpretation is difficult. Further research is needed to improve the interpretability of these latent relations.

Initially, we attempted to perform all the classification experiments that have been performed by Schlichtkrull et al. (2017) (AIFB, MUTAG, BGS, and AM), but during the attempt to run the experiment on the MUTAG dataset, we found that the L-GCN, in its current implementation, is unable to process such large graphs. We found that the MUTAG experiment takes approximately 1100 seconds per epoch, rendering the experiment unfeasible. The increase in running time originally comes from the quadratic nature of the adjacency matrix and some limitation within the TensorFlow library. Most of the L-GCN model is implemented with sparse matrices, similar to the original GCN, but due to limited support for sparse matrix multiplication within the TensorFlow library it is impossible to keep the matrices sparse throughout the whole pipeline. A more efficient implementation is left to future work.

Another improvement could be to eliminate the L-GCN's full-batch nature, which is imminent when working with adjacency matrices in this way. This forces us to perform training and testing on the full-batch, which in turn limits us in the distributability of the algorithm. One solution could be a graph sub sampling approach as used by Hamilton et al. (2017). In the L-GCN algorithm this would mean that the node features and adjacency matrices should be split up into smaller parts containing subgraphs.

The NELL algorithm shows us that the L-GCN, although it only uses one latent graph, can outperform the GCN by a large margin. The question remains if this is the result of carefully selecting which relations are important and which are not, or if this is the result of separating a node's features from the mixture of features of its neighbours as in the GCN-SSL model.

Appendix A

RITA dataset description

A.1 File description

The RITA dataset contains a predefined train and test split of 50-50. Each class contributes 50 percent of their examples to the train set and 50 percent to the test set. This split can be found in the last column of the rita.content file, 1 denoting the training split, 2 denoting the test split.

rita.content contains 283 nodes (airports) and 36 features for each node. See table A.1.

rita.cites contains 5279 edges, these are the average values for all flights on that edge concatenated with the total amount of flights on that edge. See table A.2.

	Feature name	Description (quantity)	Type	NaNs	Min	Max
1	AirportId	Airport id	int	0	0	282
2	Lat	Latitude (degrees)	float	0	-165.445	-64.8017
3	Lon	Longitude (degrees)	float	0	17.7016	71.2849
4	Elevation	Elevation (feet)	float	0	3	7837.9
5	Acres	Area size (acres)	int	0	0	33531
6	hasNOTAMse	Notice to airmen security system	Bool	0	0	1
7	hasCustoms	Customs landing rights airport	Bool	7	0	1
8	isMilCivJo	Joint use military and civil	Bool	12	0	1
9	hasMilLand	Military landing rights	Bool	9	0	1
10	hasAirFram-Ma	Airframe repair service	Bool	8*	0	1
11	hasAirFram-Mi	Airframe repair service	Bool	8*	0	1
12	hasPowerPl-Ma	Power plant (engine) repair	Bool	7*	0	1
13	hasPowerPl-Mi	Power plant (engine) repair	Bool	7*	0	1
14	Bottled02-H	Type of bottled oxygen available	Bool	12*	0	1
15	Bottled02-HL	Type of bottled oxygen available	Bool	12*	0	1
16	Bottled02-L	Type of bottled oxygen available	Bool	12*	0	1
17	Bulk02-H	Type of bulk oxygen available	Bool	37*	0	1
18	Bulk02-HL	Type of bulk oxygen available	Bool	37*	0	1
19	Bulk02-L	Type of bulk oxygen available	Bool	37*	0	1
20	hasTower	Control tower on the airport	Bool	0	0	1
21	SingleEngC	Based single engine aircraft	int	0	0	482
22	MultiEngCo	Based multi engine aircraft	int	0	0	70
23	JetEngCoun	Based jet engine aircraft	int	0	0	203
24	HeloCount	Based helicopters	int	0	0	54
25	GlidersCou	Based gliders	int	0	0	10
26	MilitaryCr	Based military aircraft	int	0	0	200
27	Ultralight	Based ultralight aircraft	int	0	0	3
28	Commercial	Commercial services	int	0	0	740363
29	Commuter	Commuter/Cargo carriers	int	0	0	185204
30	AirTaxi	Air taxi operations	int	0	0	361134
31	Local	Local operations	int	0	0	225814
32	Itinerant	Itinerant operations	int	0	0	195870
33	Military	Military aircraft operations	int	0	0	533635
34	Enplanemen	Enplanements	int	0	4220	45941440
35	Passengers	Passengers on flights	int	0	4220	46277043
36	Arrivals	Arrival flights	int	0	292	415532
37	Departures	Departure flights	int	0	295	415683
38	HubSize (label)	The Hubsize of the airport	int	0	0	3

Table A.1: Feature description of rita.content. Features are a mix of continuous (type is int or float) and categorical (type is Bool). All NaNs are filled with zeros. Categorical features are one-hot encoded (Ma=Major, Mi=Minor, H=High, HL=High/Low, L=Low). * Since these features are one-hot encoded, missing values are the same amount for each feature originating from the same categorical feature. Full description can be found here: `http://www.fgdl.org/metadata/fgdl_html/airports_2015.htm`

Feature name	Description	Quantity	Type	NaNs	Min	Max
Origin	Id of origin		int	0	0	282
Destination	Id of destination		int	0	0	283
Flights	Number of flights on that edge		int	0	1	13788
DepTime	Actual departure time	hhmm	float	10	0000	2400
CRSDepTime	Scheduled departure time	hhmm	float	0	0000	2400
ArrTime	Actual arrival time	hhmm	float	166	0000	2400
CRSArrTime	Scheduled arrival time	hhmm	float	0	0000	2400
ActualElapsedTime	Actual elapsed time	minutes	float	166	22	640
CRSElapsedTime	Scheduled elapsed time	minutes	float	8	8	645
AirTime	Flight time	minutes	float	166	9	609
ArrDelay	Arrival delay	minutes	float	166	-46	575
DepDelay	Departure delay	minutes	float	10	-36	587
Distance	Distance	miles	float	0	11	4962
TaxiIn	Wheels down to arrival at gate	minutes	float	166	2	48
TaxiOut	Departure at gate to wheels off	minutes	float	10	3	104
Cancelled	Cancelled ratio		float	0	0	1
Diverted	Diverted ratio		float	0	0	1
CarrierDelay	Carrier delay	minutes	float	248	0	545
WeatherDelay	Weather delay	minutes	float	248	0	210
NASDelay	National Airspace System delay	minutes	float	248	0	257
SecurityDelay	Security delay	minutes	float	248	0	11
LateAircraftDelay	Late aircraft arrival delay	minutes	float	248	0	170

TABLE A.2: Feature description of rita.cites. All NaN values are filled by the mean for that feature. Types are all continuous. Negative delay means the flight arrived or departed early. Origin and Destination are equal to AirportId in **rita.content**. Full description can be found here: `http://www.stat.purdue.edu/~lfindsen/stat350/airline2008_dataset_definition.pdf`

A.2 Cross-correlation matrix

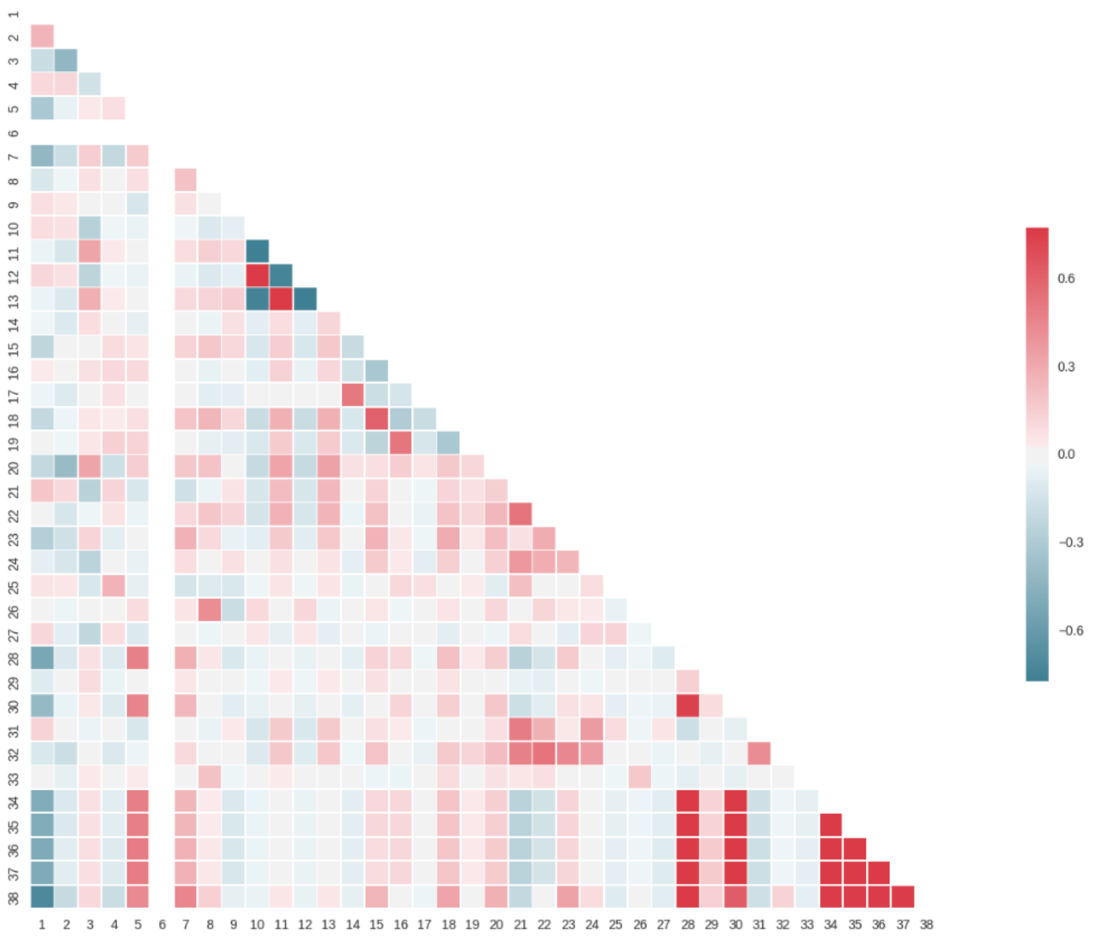

FIGURE A.1: Cross-correlation of node features in RITA dataset. The indices correspond to the indices in A.1. Best viewed digital or in color print.

Appendix B

Sparse adjacency matrix

B.1 Motivation

The main reason why we process the edge features in a matrix form rather than a tensor form is the computational complexity, in this fashion no unnecessary computations are performed whilst calculating the edge embeddings. This can be important since the number of edges can be enormous. Another reason is the limited support for sparse tensor multiplication within TensorFlow.

In the original GCN a sparse matrix from the SciPy library (Jones et al., 01) is used. A custom Keras input layer is used to process this sparse matrix. This works well for single and multiple adjacency matrices but it fails when the adjacency matrices are built during the TensorFlow computation graph.

B.2 Example

This example should give an idea on how the `tf.SparseTensor` is used to combat the memory issues.

We have edge features stored in a file as shown below:

```
0,1,a,b,c
0,2,a,b,c
0,4,a,b,c
1,0,a,b,c
1,4,a,b,c
```

```
2,3,a,b,c
2,4,a,b,c
3,1,a,b,c
3,4,a,b,c
4,0,a,b,c
4,2,a,b,c
4,4,a,b,c
```

The origin and the destination of the edges are denoted in the first and second column, respectively. The following columns are the distinct features for each edge. In the original GCN, the adjacency matrix is built by placing ones in a sparse matrix at the indices corresponding to the first and second column. In the GCN-Normalized Edge Features version (as we introduced in section 4.2.4), the ones are replaced with the normalized values for each row. Since in this example we have three edge features (columns after the first two), we create three adjacency matrices $A_0, A_1 \& A_2$. The value under `a` in the first row will be placed on row 0 and column 1 in the sparse adjacency matrix.

In the case of the edge embeddings, this works in the same fashion, only we first process the edge features into edge embeddings, in this example we will set the embedding length to two, resulting in two adjacency matrices for the embedded edges: $A_0 \& A_1$. The rest of the process is done in the same fashion.

Appendix C

L-GCN as Message Passing Neural Networks

Within the Neural Message Passing Framework (section 2.4.4) the message passing function M_t for the introduced L-GCN is defined as

$$M_t(h_i^t, h_j^t) = (\deg(i)\deg(j))^{-\frac{1}{2}} f(E_{ij}) h_j^t, \tag{C.1}$$

and the node update function U^t as

$$U_i^t(h_i^t, m_i^{t+1}) = ReLU(W^t m_i^{t+1}). \tag{C.2}$$

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Battaglia, P., Pascanu, R., Lai, M., Rezende, D. J., et al. (2016). Interaction networks for learning about objects, relations and physics. In *Advances in Neural Information Processing Systems*, pages 4502–4510.

Berg, R. v. d., Kipf, T. N., and Welling, M. (2017). Graph convolutional matrix completion. *arXiv preprint arXiv:1706.02263*.

Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022.

Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., et al. (2016). End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*.

Bruna, J., Zaremba, W., Szlam, A., and LeCun, Y. (2013). Spectral networks and locally connected networks on graphs. *CoRR*, abs/1312.6203.

Chollet, F. et al. (2015). Keras. `https://github.com/fchollet/keras`.

de Vries, G. K. D. and de Rooij, S. (2015). Substructure counting graph kernels for machine learning from rdf data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 35:71–84.

Defferrard, M., Bresson, X., and Vandergheynst, P. (2016). Convolutional neural networks on graphs with fast localized spectral filtering. *CoRR*, abs/1606.09375.

Dumais, S. T. (2004). Latent semantic analysis. *Annual review of information science and technology*, 38(1):188–230.

Duvenaud, D. K., Maclaurin, D., Iparraguirre, J., Bombarell, R., Hirzel, T., Aspuru-Guzik, A., and Adams, R. P. (2015). Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, pages 2224–2232.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. (2017). Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256.

Hamilton, W. L., Ying, R., and Leskovec, J. (2017). Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Jones, E., Oliphant, T., Peterson, P., et al. (2001–). SciPy: Open source scientific tools for Python. [Online; accessed ¡today¿].

Kearnes, S., McCloskey, K., Berndl, M., Pande, V., and Riley, P. (2016). Molecular graph convolutions: moving beyond fingerprints. *Journal of computer-aided molecular design*, 30(8):595–608.

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907.

Le, Q. and Mikolov, T. (2014). Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1188–1196.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. (2015). Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*.

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

Mitchell, T., Cohen, W., Hruschka, E., Talukdar, P., Betteridge, J., Carlson, A., Dalvi, B., Gardner, M., Kisiel, B., Krishnamurthy, J., Lao, N., Mazaitis, K., Mohamed, T., Nakashole, N., Platanios, E., Ritter, A., Samadi, M., Settles, B., Wang, R., Wijaya, D., Gupta, A., Chen, X., Saparov, A., Greaves, M., and Welling, J. (2015). Never-ending learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI-15)*.

Perozzi, B., Al-Rfou, R., and Skiena, S. (2014). Deepwalk: Online learning of social representations. *CoRR*, abs/1403.6652.

Ristoski, P. and Paulheim, H. (2016). Rdf2vec: Rdf graph embeddings for data mining. In *International Semantic Web Conference*, pages 498–514. Springer.

Schlichtkrull, M., Kipf, T. N., Bloem, P., Berg, R. v. d., Titov, I., and Welling, M. (2017). Modeling relational data with graph convolutional networks. *arXiv preprint arXiv:1703.06103*.

Schütt, K. T., Arbabzadah, F., Chmiela, S., Müller, K. R., and Tkatchenko, A. (2017). Quantum-chemical insights from deep tensor neural networks. *Nature communications*, 8:13890.

Sen, P., Namata, G., Bilgic, M., Getoor, L., Gallagher, B., and Eliassi-Rad, T. (2008). Collective classification in network data. Technical report.

Shervashidze, N., Schweitzer, P., Leeuwen, E. J. v., Mehlhorn, K., and Borgwardt, K. M. (2011). Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep):2539–2561.

Yanardag, P. and Vishwanathan, S. (2015). Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1365–1374. ACM.

Yang, Z., Cohen, W. W., and Salakhutdinov, R. (2016). Revisiting semi-supervised learning with graph embeddings. *arXiv preprint arXiv:1603.08861*.