UNIVERSITEIT VAN AMSTERDAM

MSc Artificial Intelligence
Master Thesis

Knowledge Generation

by
Florian Wolf
12393339

January 3, 2021

48 Credits
April 2020 - December 2020

Supervisor:
Dr Peter Bloem
Thiviyan Singam
Chiara Spruijt

Asessor:
Dr Paul Groth

Contents

List of Tables

Abstract

We generate Knowledge! [1]

This thesis applies the genius idea of having a VAE learn latent features of the raw data from KGs. Building on successful models of prior work, a linear, a convolutional and a architecture combining both methods are tested on the two most popular KGs. To best possible train out models on sparse graph representation, we implement a permutation invariant loss function. We compare the performance of our models to sate of the art link predictors, as well as link predictors including the variational module. The results compare ??? The model x outperfoms the others, indicating that convolutions are [/not] necessary. Interpolation of the latent space shows that the model learns features??? When generating subgraphs with up to x nodes, we see ??? Finally we filter generated triples for predicates which imply the subject or object to be entity of the class location. x out of these triples adhere to this axiom. Thus we can say that VAE's are to a certain extend able to capture the underlying semantics of a KG.

1 Introduction

Here comes a beautiful introduction. Promise!

1.1 Motivation

Computer vision reached a point, where semantics, entities and relations can be inferred in an simple image. Would it not be fantastic to be able to apply this to text too? Could we train a model to learn the semantics of a KG?

1.2 Expected Contribution

This thesis is aimed to be at fundamental research and provide insight, into if further research in this direction would be meaningful.

1.3 Research Question

`How well can a VAE learn the underlying semantics of a KG?`

Without further ado -

2 Related Work

This section presents previous work which inspired and layed the fundamentals for this thesis. Relevant papers to three topics related to this thesis will be presented. We will present their method and results in the fields of relational graph convolutions, graph encoders, and embedding based link prediction.

2.1 Relational Graph Convolutions

We define a graph as $G = (\mathcal{V}, \mathcal{E})$ with a set of nodes $\mathcal{V}$ and a set of edges $\mathcal{E}$. The set of edges, with each edge connecting node x and y, is defined by $\{(x, y) \mid (x, y) \in \mathcal{V}^2 \wedge x \neq y\}$ while the constrain $x \neq y$ disallows self-connections or self-loops, which is optional depending on the graphs function. Moreover, nodes and edges can have describing features, which contribute additional information about the nodes and their connection. Using graph convolutions, we make use of these properties holding spectral information about their neighboring nodes and relations. The two main standards of evaluate the performance of a neural network on graphs, are

node classification and link prediction. Node classification is a classification problem where the model provides a probability distribution over all classes for each node. During link prediction de model scores a set of one real and corrupted triples and aims to score the highest on the real triple. A more in-depth explanation follows later on in chapter 4.

In Thomas Kipf's and Max Welling's first paper on graph convolutions [2] a novel Graph Convolution Network (GCN) for semi-supervised classification is introduced. This method acts directly on the graph structure and shows to be linearly scalable with the number of nodes. While the authors compare different propagation models for the graph convolutions, their propagating rule using a first-order approximation of spectral graph convolutions, outperforms all other implementations. This so called renormalization trick normalized the adjacency matrix and adds it to an identity matrix of same size, what keeps the eigenvalues in a range between $[0, 2]$ what again leads to a stable training, avoiding numerical instabilities and vanishing gradients during learning. Additionally the feature information of neighboring nodes is propagates in every layer what shows improvement in comparison to earlier methods, where only label information is aggregated. Kipf and Welling perform node classification on the three citation-network datasets, Citeseer, Cora and Pubmed as well as on the KG dataset NELL. In all classification tasks, their results outperform other recently proposed methods in this field and proves computational more efficient than its competition. More details on the implementation of graph convolutions can be found in the next chapter.

In their publication *Modeling Relational Data with Graph Convolutional Networks* Schlichtkrull, Kipf, Bloem, v.d. Berg, Titov and Welling propose a relational graph convolutional network (RGCN) and evaluate it on link prediction on the FB15K-237 and WN18 dataset and node classification on the AIFB, MUTAG, BGS and AM datasets [3]. While the RGCN with its encoder properties, it is used by it self as node classifier, yet for link prediction it is coupled with a DistMult model acting as decoder which scores triples encoded by the RGCN see 2.1. More details on the DistMult can be found in 2.3.

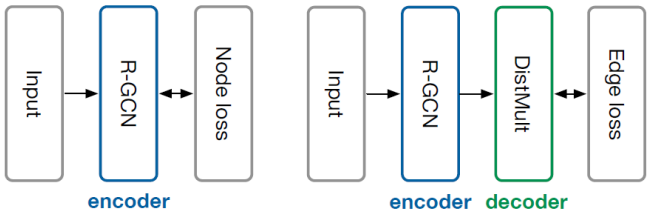


Figure 1: RGCN pipeline for node classification and link prediction experiments. The first pipeline only uses an encoder to classify the input nodes. The second pipeline additionally uses a decoder to score the input and predict the correct link, Source [3].

The RGCN works on graphs stored as dense triples, creating a hidden state for head and tail of each triple. A novel message passing network is layer-wise propagated with the hidden states of the entities. As regularization the authors propose a *basis-* and *blockwise* decomposition. while the first aims at an effective weight sharing between different relation types, the second can be seen as a sparsity constraint on the relation type's weight. The model outperforms embedding based model on the link prediction task on the FB15K-237 dataset and scores competitive on the WN18 dataset. In the node classification task, the model sets state of the art results on the datasets AIFB and AM, while scoring competitive on the rest. The authors conclude, that the model has difficulties encoding higher-degree hub nodes on datasets with many entities and low amount of classes.

2.2 Graph VAE

We have seen how graph convolutional neural networks can be combined to a encoder-decoder architecture, resulting in a generative model suitable for unsupervised learning. We will present three recent publications with different methods and usecases of a graph generative model, in particular a VAE.

In yet another publication of Kipf and Welling introduce the Variational Graph Autoencoder (VGAE), a framework for unsupervised learning on graph-structured data [4]. This generative model uses a GCN as encoder and a simple inner product module as decoder. Similar to th GCN the VGAE incorporates node features, what significantly improves its performance on link prediction tasks compared to related models. The VGAE uses a two-layer GCN to encode the mean and the logvariance of for the stochastic module to sample the latent space representation. The activation of the inner product of this latent vector yields then the reconstruction

of the adjacency matrix. Figure 2.2 shows how the model learns to represent the underlying data structure by grouping the latent representations of the datapoints according to their class, without these labels being provided to the model during training.

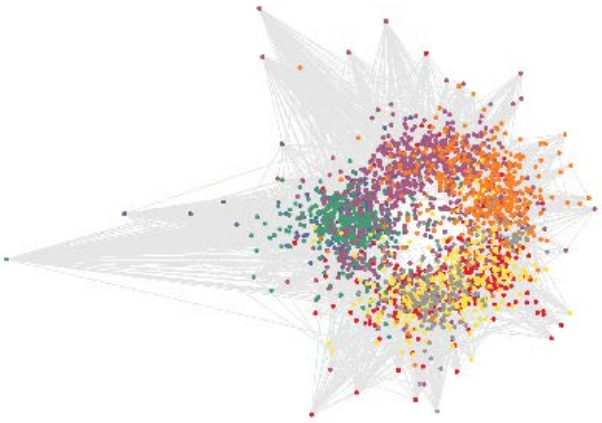


Figure 2: Colorized vizualization of the latent representation of the VGAE trained on the Core citation network with colors differentiating document classes and gray links indicating citations. This shows that the model implies and featurizes the document classes, without them being provided during training. Source [4]

The VGAE with added features outperforms state of the art models in the task of link prediction on the datasets Cora, Citeseer and Pubmed. The authors point out, that a Gaussian prior might be a poor choice combined with the inner-product decoder.

While citation networks represent basic graph structures, there has also been done work on more complex KGs. Simonovsky and Komodakis introduce the GraphVAE, a generative model which outputs a probabilistic fully-connected graph of a predefined maximum size in a one-shot approach [5]. The model includes a standard graph matching algorithm to align the predicted graph to the ground truth. In contrast to the previously presented publications, the input to this model is a threefold and sparse graph, defined as $G = (A, E, F)$ with A being the adjacency matrix, E the edge attribute matrix and F the node attribute matrix, with E ad F being one-hot encoded. Considering that this method lays the foundation for this thesis, we will adopt this notation for our own methods in 4. Figure 2.2 shows the architecture of the GraphVAE. The encoder is a feed forward network with edge-conditioned graph convolutions. After the convolutions the result is flattened and conditioned on the node labels y. A simple neural network then encodes the stochastic latent space, as we know it from previous works. The decoder is again conditioned on the node labels y and in form of a fully-connected neural network reconstructs the graph prediction. The threefold decoder output is matched with the target using graph matching algorithm, which we will discuss further in 3.3. The best matching permutation is then used for the reconstruction loss term of the GraphVAE. Notable is, that the size of the target and prediction graph do not necessarily have to match. While this approach seems promising, the maximum graph size is limited to a node count of 100 by computational memory requirements.

The model is trained on the QM9 dataset, containing the graph structure of 134k organic molecules with experiments on latent space dimension in the range of $[20, 80]$. On the free generation task, about 50% of the generated molecules are chemically valid and thereof remarkably 60% are not included in the trainings dataset. When testing the model for robustness, it showed little disturbance when adding Gaussian noise to the input graph G. The authors conclude that the problem of generating graphs from a continuous embedding was addressed successfully and that the GraphVAE performs better on small molecules, thus worse on larger graphs.

In a little sidestep, we present the idea of supervised graph generation in an autoregressive fashion by Belli and Kipf and their publication *Image-Conditioned Graph Generation for Road Network Extraction* [6]. While we will focus on the generative model, their contribution ranges wider, namely the introduction of the graph-based roadmap dataset *Toulouse Road Network* and the task specific distance metric *StreetMover*. The authors propose the Generative Graph Transformer (GGT) a deep autoregressive model that makes use of attention mechanisms on images, to tackle the challenging task of road network extraction from image data. The GGT has a encoder-decoder architecture, with a CNN as encoder, taking the grayscale image as input signal and predicting a conditioning vector. The decoder is a self-attentive transformer, which takes as input the encoded condition vector and a hidden representation of the adjacency matrix A and feature vector X of

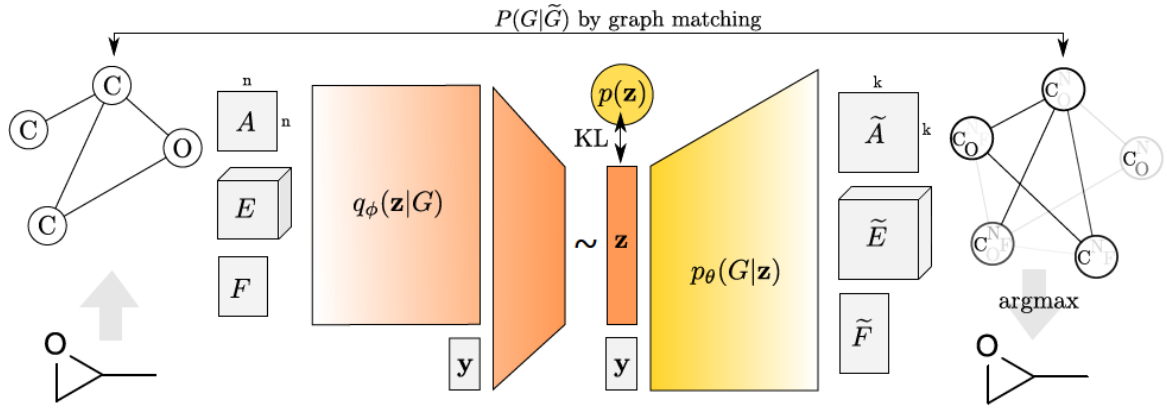


Figure 3: Model architecture of the GraphVAE. The target graph with n nodes is encoded and conditioned on the node labels y. The KL divergence ensures a Gaussian prior to the decoder, which reconstructs the latent representation to a graph with k nodes. Target and prediction graphs are matched and permuted before the reconstruction loss. To sample, the argmax is taken directly from the prediction. Source [5].

the previous step. The adjacency matrix here indicates the links between steps and the features are normalized coordinates. A multi-head operator outputs the hidden representation of A and X which finally are decoded by a MLP to the graph representation. For the fist step, a empty hidden representation is fed into the decoder. The model terminates the recurrent graph generation by predicting a end-of-sequence token, which signalizes the end of the graph. During learning, the generated graphs are matched to the target graphs using the *StreetMover* metric, based on the Sinkhorn distance. The authors attribute *StreetMover* as a scalable, efficient and permutation-invariant metric for graph comparison. The successful results of the experiments performed, show that this novel approach is suitable for the task of road network extraction and could yield similar success in graph generation task of different fields. While this publication does not directly align with the previously presented work, we find it of added value to present alternative approaches on our topic.

2.3 Embedding Based Link Prediction

Finalizing this chapter, we will look at embedding based methods on KGs. These approach was inspired by the success of word-embeddings in the field of NLP. Compared to the previously presented research, embedding models have a much simpler architecture and can be trained computationally very efficiently on large graphs. Embedding based models can only operate on simple triples, meaning a KG is represented as set of triples with indices pointing to the unique entity and relation in the graph. In disregard of their simplicity, they achieve great results on relational prediction tasks such as link prediction. In particular, this means to rank a correct triple the highest between a set of corrupted triples. In order to generate corrupt triples, each triple from the test set is modified by replacing the head or tail with all remaining entities occurring in the KG. Link prediction will be explained in more detail in 4.

Already in 2013 Bordes et al. introduced in their paper *Translating Embeddings for Modeling Multi-relational Data* the low-dimensional embedding model TransE [7]. The core idea of this model is that relations can be represented as translations in the embedding space. Entities are encoded to a low-dimension embedding space and the relation is represented as vector between the head and tail entity. The assumption is, that correct triples have a shorter relational vector than corrupted triples, thus, are closer together in embedding space. The loss function for learning of the model takes a set of corrupted triples for every triples in the training set and subtracts the translation vector of the corrupted triple in embedding space from the translation vector of the correct triple with added margin. To minimize the loss, the model has to place entities of correct triples closer together in embedding space. We think of a triple as (s, r, o) and the bold notation its embedded representation, $d()$ the distance measure, γ the positive margin and S and S' as sets of correct and corrupt triples, the loss function of TransE is

$$\mathcal{L} = \sum_{S} \sum_{S'} [\gamma + d(\boldsymbol{s} + \boldsymbol{r}, \boldsymbol{o}) - d(\boldsymbol{s}' + \boldsymbol{r}, \boldsymbol{o}')] \tag{1}$$

The model is trained on the two KGs Freebase and Wordnet, which will also be the source for the datasets used in this thesis. TransE's link prediction results on both head and tail outperformed other competing methods of the time, such as RESCAL [8].

In 2015, Yang et al. proposed a similar, yet better performing KG embedding method [9]. Their model DistMult captures relational semantics by matrix multiplication of the embedded entity representation and uses a bilinear learning objective. The main difference o TransE is the bilinear scoring function $d_r^b()$, with bilinear meaning the score invariance of swapping head and tail entity. For the embedding space representation of subject and object $\boldsymbol{s}$ and $\boldsymbol{o}$ and a diagonal matrix $\mathbf{M}_r \in R^{n \times n}$ as embedding of r, the scoring function is

$$d_r^b(\boldsymbol{s}, \boldsymbol{o}) = \boldsymbol{s}^T \mathbf{M}_r \boldsymbol{o} \tag{2}$$

The publication goes on to explore the options of embedding based rule extraction from KGs. Concluding, the authors state that that embeddings learned from the bilinear objective not only outperform the state of the art in link prediction but can also capture compositional semantics of relations and extract Horn rules using compositional reasoning.

In a more recent publication by Ruffinelli et al. outdated KG embedding models such as TransE and DistMult are re-trained with state of the art techniques in deep learning. The authors start by pointing out the similarities and differences of most models. While all methods share the same embedding approach, they differ in their scoring function and their original hyperparameter search. The authors perform a quasi-random hyperparameter search on the five models RESCAL, TransE, DistMult, ComplEx and ConvE using the two datasets FB15K-237 and WN18. For evaluation the MRR and Hits@10 are used.Since these metrics nad datasets will be used later on our research, those metrics are explained in 4. The tuned models report a up to 24% higher MRR score, compared to their first reported performance. The authors conclude that simple KG embedding methods can show strong performance when trained with state-of-the-art techniques and score competitive to or even outperformed more recent architectures. The improved model configurations were found by exploring relatively few random samples from a large hyperparameter space.

3 Background

In this section we will go over related work and relevant background information for our model and experiments. The depth of the explanation is adopted to the expected prior knowledge of the reader. The reader is supposed to know the basics of machine learning and deep learning, including probability theory and basic knowledge on neural networks and their different architectures. Basic principles such as forward pass, backpropagation and convolutions are expected to be understood. Further the use and functionality of deep learning modules such as the model, the optimizer and the terms target and prediction should be known. This also includes being familiar with the training and testing pipeline of a model in deep learning.

Should all these boxes be checked, then we can expect to get a deeper understanding of the magic behind the VAE and its differences to a normal autoencoder. After that we will present how convolutional layers can be used on graphs. Of course where there is a layer there is a model, thus we are presenting the graph convolutional network (GCN). Closing the circle we show how we can adopt the VAE to graph convolutions. Wrapping things up, we present the state of the art algorithms for graph matching, which will be util to allow permutation invariance when matching prediction and target graph [10].

3.1 Knowledge Graph

Knowledge graph has become a popular key phrase. Yet, the term is so broad, that it can have various definitions. In this thesis we are going to focus on KGs in the context of relational machine learning.

A KG is a database and as all other databases it is used to store data. The main difference to tabular databases is that KGs store data in a relational fashion. A standard KG structure, introduced by the semantic-web community, is the Resource Description Framework (RDF). It is a so called schema-based approach, meaning that every entity has a unique identifier and all possible relations are stored in a vocabulary. The opposite schema-free approach is used in OpenIE models for information extraction. Here any type of triple can be

extracted, eg. (Michelangelo, painted, Sixtine Chapel). Where as a triple in RDF format from Freebase, one of the largest open-source KGs, has the form

$$(/m/02mjmr, /people/person/born-in, /m/03gh4)$$

$$(s, r, o)$$

with:

- Subject $s : /m/02mjmr$ Barack Obama
- Relation/Predicate $r : /people/person/born-in$
- Object $o : /m/03gh4$ Hawaii

For all triples s and o are part of a set of so called entities, while r is part of a set of relations. This is enough to define a basic KG.

Schema-based KG can include type hierarchies and type constrains. Classes group entities of the same type together, based on common criteria, e.g all names of people can be grouped in the class 'person'. Hierarchies define the inheriting structure of classes and subclasses. Picking up our previous example, 'child' would be a subclass of 'person' and inherit its properties. At the same time the class of an entity can be key to a relation with type constrain, since some relations can only be used in conjunction with entities fulfilling the constraining type criteria.

These schema based rules of a KG are defined in its ontology. Here properties of classes, subclasses and constrains for relations and many more are defined. Again, we have to differentiate between KGs with open-world or closed-world assumption. In a closed-world assumption all constrains must be sufficiently satisfied before a triple is accepted as valid. This leads to a huge ontology and makes it difficult to expand the KG. On the other hand open-world KGs such as Freebase, accept every triple as valid, as long as it does not violate a constrain. This again leads inevitably to inconsistencies within the KG, yet it is the preferred approach for large KGs. In context of this thesis we refer to the ontology as semantics of a KG, we research if our model can capture the implied closed-world semantics of an open-world KG [11].

Lastly, we point out one major difference between KGs, namely their representation. RDF KGs are represented as set of triples, consisting of a unique combination of numeric indices. Each index linking to the corresponding entry in the entity and relation vocabulary. This is called dense representation and benefits from fast computation due to an optimized use of memory.

In contrary the dense representation of a triple is the sparse representation. Here a binary square matrix also called the adjacency matrix, indicated a link between two entities. To identify the node, each node in the adjacency matrix has a one-hot encoded entity-vocabulary vector. All one-hot encoded vectors are concatenated to a node attribute matrix. In simple cases, like citation networks this is a sufficient representation. In the case of Freebase, we need an additional edge-attribute matrix, which indicates the relation of each link. The main benefits of this method are the representation of subsets of triples, also subgraphs, with more than one relation and the computational possibility to perform graph convolutions.

3.2 Graph VAE – one shot method

3.2.1 VAE

The VAE as first presented by [12] is an unsupervised generative model in form of an autoencoder, consisting of an encoder and a decoder. Its architecture differs from a common autoencoder by having a stochastic module between encoder and decoder. The encoder can be represented as recognition model with the probability $p_\theta(\mathbf{z} \mid x)$ with x being the variable we want to inference and z being the latent representation given an observed value of x. The encoder parameters are represented by θ. Similarly, we denote the decoder as $p_\theta(\mathbf{x} \mid z)$, which given a latent representation z produces a probability distribution for the possible values, corresponding to the input of x. This will be the base architecture of all our models in this thesis.

The main contribution of the VAE is the so called reparameterization trick. When sampling from the latent prior distribution, we get a stochastic module inside our model, which can not be backpropagated and

makes learning impossible. By placing the stochastic module outside the model, we can again backpropagate. We use the predicted latent space as mean and variance for a Gaussian normal distribution, from which we then sample ϵ, which acts as external parameter and does not need to be updated.

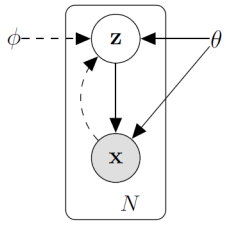


Figure 4: Representation of the VAE as bayesian network, with solid lines denoting the generator $p_\theta(z)p_\theta(\mathbf{x} \mid z)$ and the dashed lines the posterior approximation $q_\phi(\mathbf{z} \mid \mathbf{x})$ [12].

In 4 we see, that the true posterior $p_\theta(\mathbf{z} \mid \mathbf{x})$ is intractable. Thus, we make the assumption that the prior to the decoder is Gaussian with an approximately diagonal covariance, which gives us the approximated posterior.

$$\log q_\phi \left(\mathbf{z} \mid \mathbf{x}^{(i)} \right) = \log \mathcal{N} \left(\mathbf{z}; \boldsymbol{\mu}^{(i)}, \boldsymbol{\sigma}^{2(i)} \mathbf{I} \right) \tag{3}$$

Now variational inference can be performed, which allows both θ the generative and ϕ the variational parameters to be learned jointly. Using the Monte Carlo estimation of $q_\phi(\mathbf{z} \mid \mathbf{x})$ we get the so called estimated lower bound (ELBO):

$$\mathcal{L} \left(\theta, \phi; \mathbf{x}^{(i)} \right) = -D_{KL} \left(q_\phi \left(\mathbf{z} \mid \mathbf{x}^{(i)} \right) \| p_\theta(\mathbf{z}) \right) + E_{q_\phi \left(\mathbf{z} \mid \mathbf{x}^{(i)} \right)} \left[\log p_\theta \left(\mathbf{x}^{(i)} \mid \mathbf{z} \right) \right] \tag{4}$$

We denote the first term the regularization term, as it forces the model into using a Gaussian normal prior. The second term represents the reconstruction loss, matching the prediction with the target.

While we use a discrete input space, the output space is a continuous probability. To generate final result, the prediction is used as binomial probability distribution, from which we then sample. Once training of a VAE is completed, the Decoder can be used on its own to generate new samples by using latent input signals [12].

3.2.2 MLP

The Multi-Layer Perceptron (MLP) is the vanilla basemodel of all neural networks. Its properties as universal approximator has been discovered and widely studied since 1989. The innovation it brought to existing models was the hidden layer between the input and the output.

The mathematical definition of the MLP is rather simple. It takes linear input vector of the form $x_1, ..., x_D$ which is multiplied by the weight matrix $\mathbf{w^{(1)}}$ and then transformed using a non-linear activation function $h(\dot{})$. Due to its simple derivative, mostly the rectified linear unit (ReLU) function is used. This results in the hidden layer, consisting of hidden units. The hidden units get multiplied with the second weight matrix, denoted $\mathbf{w^{(2)}}$ and finally transformed by a sigmoid function $\sigma(\dot{})$, which produces the output. Grouping weight and bias parameter together we get the following equation for the MLP

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma \left(\sum_{j=0}^{M} w_{kj}^{(2)} h \left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i \right) \right) \tag{5}$$

for $j = 1, \ldots, M$ and $k = 1, \ldots, K$, with M being the total number of hidden units and K of the output.

Since the sigmoid function returns a probability distribution for all classes, the MLP can have the function of a classifier. Instead of the initial sigmoid function, it was found to also produce good results for multi label

classification transforming the output with a softmax function instead. Images or higher dimensional tensors can be processed by flattening them to a one dimensional tensor. This makes the MLP a flexible and easy to implement model [13].

3.2.3 Graph convolutions

Convolutional neural nets (CNN) have the advantage to be invariant to permutations of the input. Convolutional layers exploit the property of datapoints which are close to each other and thus, have a higher correlation. CNNs have shown great results in the field of images classification and object detection. Neighboring pixel contain information about each other, thus by detecting local features, the model can then merged those to high-level features, e.g a face in an image [13]. Similar conditions hold for graphs. Neighboring nodes contain information about each other and can used to infer local features.

Let us shortly go over the definition and math behind graph convolutions. Different approaches have been published on this topic, here we will present the graph convolution network (GCN) of [2]. We consider $f(X, A)$ a GCN with an undirected graph input $G = (\mathcal{V}, \mathcal{E})$, where $v_i \in \mathcal{V}$ is a set of N nodes and $(v_i, v_i) \in \mathcal{E}$ the set of edges. The input is a sparse graph representation, with X being a node feature matrix and $A \in R^{N \times N}$ being the adjacency matrix, defining the position of edges between nodes. In the initial case, where self-connections are not considered, the adjacency's diagonal has to be one resulting in $\vec{A} = A + I_N$. The graph forward pass through the convolutional layer l is then defined as

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right). \tag{6}$$

Here $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ acts as normalizing constant. $W^{(l)}$ is the layer-specific weight matrix and contains the learnable parameters. H returns then the hidden representation of the input graph.

The GCN was first introduced as node classifier, meaning it returns a probability distribution over all classes for each node in the input graph. Assuming that we preprocess $\vec{A}$ as $\hat{A} = \tilde{D}^{-\frac{1}{2}}\vec{A}\tilde{D}^{-\frac{1}{2}}$, the equation for a two-layer GCN for z classes is

$$Z = f(X, A) = \text{softmax}\left(\hat{A}\,\text{ReLU}\left(\hat{A}XW^{(0)}\right)W^{(1)}\right). \tag{7}$$

3.2.4 Graph VAE

Now we have understood all the principles needed for a Graph VAE. While there are many variations, not only in architecture but also in graph representation, namely sparse or dense, we will focus here on the graph VAE presented in [5]. A sparse graph model with graph convolutions.

The encoder $q_\phi(\mathbf{z} \mid G)$ takes a graph G as input, on which graph convolutions are applied. After the convolutions the hidden representation is flattened and concatenated with the node label vector y. A simple MLP encodes the mean and logvariance of the latentsapce distribution. Using the reparametrization trick we sample the latent representation.

For the decoder $p_\theta(\mathbf{x} \mid G)$ the latent representation is again concatenated with the node labels. The decoder architecture for this model is a reverse MLP, which outputs a flat prediction of G, which is split and reshaped in to the sparse matrix representation.

3.2.5 One Shot vs. Recursive

The concept of the VAE has been used to generate data for various usecases. When using the VAE as generator, by sampling from the approximated posterior distribution $q_\phi(\mathbf{z})$, we can reconstruct the data in a singular run or recursive manner.

The one shot method is the used in the popular example of the VAE generator on the MNIST dataset [12]. each sample is independent from each other.

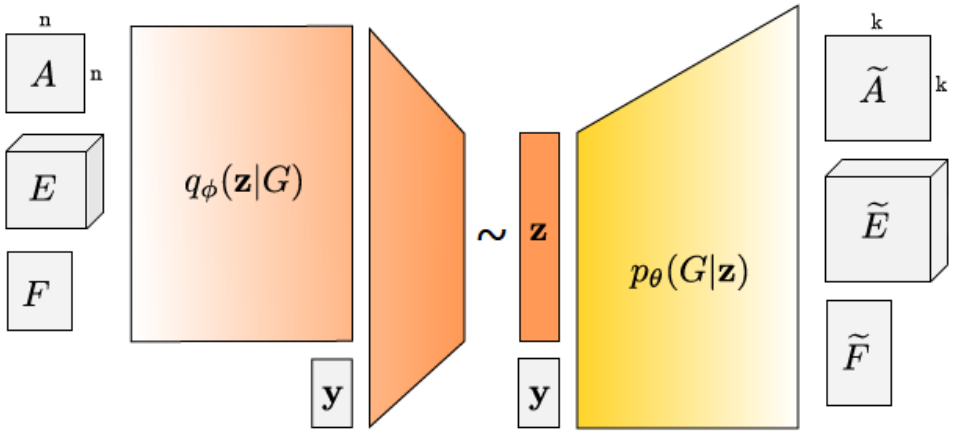


Figure 5: Illustration of the GraphVAE architecture as presented in [5]. The decoder combines graph convolutions and a MLP with concatenated labels y. The decoder reconstruct G from the latent representation and the concatenated node labels y.

Recursive methods take part of the generated datapoint as input for the next datapoint, thus continuously generating the sample. This has been applied to generate audio and to reproduce videogame environments [14]. In [6] variation of the GraphVAE has been used to recursively construct a vector roadmap, which has been presented in [link to related work].

For this thesis, we will use the one-shot method, predicting each datapoint independent from each other. The datapoints will be sparse subgraph with n nodes. A single triple being $n = 2$ and a subgraph representation $2 < n < 100$.

3.3 Graph Matching

In this subsection we will explain the term permutation invariance and present an algorithm to find the optimal permutation in order to match two graphs. We then derive the loss term of the graph VAE 3.2.4 applying the optimal matching matrix to the models prediction.

3.3.1 Permutation Invariance

Permutation invariance refers to the invariance of a permutation of an object. An visual example is the the image generation of numbers. If the loss function of the model would not be permutation invariant, the generated image could show a perfect replica of the input number but being translated by one pixel the loss function would penalize the model. Geometrical permutations can be translation, scale or rotation around any axis.

In the context of sparse graphs the most common, and relevant permutation for this thesis, is be the position of a link in the adjacency matrix. By altering its position with help of a permutation matrix, the link will connect different nodes. When matching graphs with more than two nodes, we can partially match the graphs by permuting only a subgraph instead of the full graph. This way also graphs of different sizes, can be matched.

A model is permutation invariant, when it includes a function to detect and match such permutations between target and prediction.

3.3.2 Max-Pool Graph matching algorithm

While there are numerous graph matching algorithms, we will focus on the max-pool algorithm, which can be effectively implement and used in the VAE setting. First presented in [15] in the context of computer vision and successfully trained to match graphs from feature points in an image. It uses a max-pooling graph matching

approach, which is resilient to deformations and highly tolerant to outliers. The output is a reliable cost matrix. Notable is, that also graphs with different number of nodes can be matched.

Let us introduce a new sparse representation for a sampled subgraph, where the discrete target graph is $G = (A, E, F)$ and the continuous predicted graph $\widetilde{G} = (\widetilde{A}, \widetilde{E}, \widetilde{F})$. The A, E, F are store the discrete data for the adjacency, for node attributes and the node attribute matrix of form $A \in \{0,1\}^{n \times n'}$ with n being the number of nodes in the target graph. $E \in \{0,1\}^{n \times n' \times d_e}$ is the edge attribute matrix and a node attribute tensor of the shape $F \in \{0,1\}^{n \times d_n}$ with d_e and d_n being the size of the entity and relation dictionary. For the predicted graph with k nodes, the adjacency matrix is $\widetilde{A} \in [0,1]^{k \times k}$, the edge attribute matrix is $\widetilde{E} \in R^{k \times k \times d_e}$ and the node attribute matrix is $\widetilde{F} \in R^{k \times d_n}$.

Given these graphs the algorithm aims to find the affinity matrix $S : (i,j) \times (a,b) \to R^+$ where $i, j \in G$ and $a, b \in \widetilde{G}$. The affinity matrix expresses the similarity of all nodepairs between the two graphs and is calculated

$$S((i,j),(a,b)) = \left(E^T_{i,j,\cdot}, \widetilde{E}_{a,b,\cdot}\right) A_{i,j} \widetilde{A}_{a,b} \widetilde{A}_{a,a} \widetilde{A}_{b,b} [i \neq j \wedge a \neq b] + \left(F^T_{i,\cdot} \widetilde{F}_{a,\cdot}\right) \widetilde{A}_{a,a} [i = j \wedge a = b] \tag{8}$$

where the square brackets define Iverson brackets [5].

The next step is to find the similarity matrix $X^* \in [0,1]^{k \times n}$. Therefore we iterate a fist-order optimization framework and get the update rule

$$\mathbf{x}_{t+1} \leftarrow \frac{1}{\|\mathbf{S}\mathbf{x}_t\|_2} \mathbf{S}\mathbf{x}_t. \tag{9}$$

To calculate Sx we find the best candidate $x_{i,a}$ from the possible pairs in the affinity matrix. Heuristically, taking the argmax over all neighboring nodepair affinities yields the best result. Other options are sum-pooling or average-pooling, which do not discard potentially irrelevant information, yet have shown to perform worse. Thus, using the max-pooling approach, we can pairwise calculate

$$\mathbf{S}\mathbf{x}_{ia} = \mathbf{x}_{ia} \mathbf{S}_{ia;ia} + \sum_{j \in N_i} \max_{b \in N_a} \mathbf{x}_{jb} \mathbf{S}_{ia;jb}. \tag{10}$$

Depending on the matrix size, the number of iterations are adjusted. The resulting similarity matrix $X*$ yields a normalized probability of matching nodepairs. In order to get a discrete translation matrix, we need to find the optimal match for each node.

3.3.3 Hungarian algorithm

Picking up on the nodepair probabilities X^* of last chapter, we reformulate it as a linear assignment problem. Here comes into play an optimization algorithm, the so called hungarian algorithm. It original objective is to optimally assign n resources to n tasks. The cost of assigning task $i \in R^n$ to $j \in R^n$ is stored in x_{ij} of the quadratic cost matrix $x \in R^{n \times n}$. By assuming tasks and resources are simple nodes and taking $C = 1 - X^*$ we get the cost matrix C for the optimal translation. This algorithm is has a complexity of $O\left(n^4\right)$, thus is not applicable to complete KGs but only to subgraphs with limited number of nodes per graph [16].

The core of the hungarian algorithm consist of four main steps, initial reduction, optimality check, augmented search and update. The now presented algorithm is a slight alternative of the original algorithm and improves the complexity of the update step from $O\left(n^2\right)$ to $O\left(n\right)$ and thus, reduces the total complexity to $O\left(n^3\right)$. It takes as input a bipartie graph $G = (V, U, E)$ and the cost matrix $C \in R^{n \times n}$ where $V \in R^n$ and $U \in R^n$ are sets of nodes and $E \in R^{n \times n}$ the set of edges between the nodes. The algorithm's output is a discrete matching matrix M. To avoid two irrelevant pages of pseudocode, the steps of the algorithm are presented in the following short summary [**mills2007dynamic**].

1. Initialization:

(a) Initialize the empty matching matrix $M_0 = \emptyset$.

(b) Assign α_i and β_i as follows:

$$\begin{aligned} &\forall v_i \in V, && \alpha_i = 0 \\ &\forall u_i \in U, && \beta_j = \min_i (c_{ij}) \end{aligned}$$

2. Loop n times over the different stages:

(a) Each unmatched node in V is a root node for an Hungarian tree with when completed results in an augmentation path.

(b) Expand the Hungarian trees in the equality subgraph. Store the indices i of v_i encountered in the Hungarian tree in the set $I*$ and similar for j in u_j and the set J^*. If an augmentation path is found, skip the next step.

(c) Update α and β to add new edges to the quality subgraph and redo the previous step.

$$\begin{aligned} \theta &= \frac{1}{2} \min_{i \in I^*, j \notin J^*} (c_{ij} - \alpha_i - \beta_j) \\ \alpha_i &\leftarrow \begin{cases} \alpha_i + \theta & i \in I^* \\ \alpha_i - \theta & i \notin I^* \end{cases} \\ \beta_j &\leftarrow \begin{cases} \beta_j - \theta & j \in J^* \\ \beta_j + \theta & j \notin J^* \end{cases} \end{aligned}$$

(d) Augment M_{k-1} by flipping the unmatched with the matched edges on the selected augmentation path. Thus M_k is given by $(M_{k-1} - P) \cup (P - M_{k-1})$ and P is the set of edges of the current augmentation path.

3. Output M_n of the last and n^{th} stage.

3.3.4 Graph Matching VAE Loss

Coming back to our generative model, we now explain how the loss function needs to be adjusted to work with graphs and graph matching, which results in a permutation invariant graph VAE.

The normal VAE maximizes the evidence lower-bound or, in a practical implementation, minimizes the upper-bound on negative log-likelihood. Using the notation of 3.2.1 the graph VAE loss is

$$\mathcal{L}(\phi, \theta; G) = \quad = E_{q_\phi(\mathbf{z}|G)} \left[-\log p_\theta(G \mid \mathbf{z}) \right] + \mathrm{KL} \left[q_\phi(\mathbf{z} \mid G) \| p(\mathbf{z}) \right] \tag{11}$$

The loss function $\mathcal{L}$ is a combination of reconstruction term and regularization term. The regularization term is the KL divergence between a standard normal distribution and the latent space distribution of Z. This term does not change when adopting to graphs. The reconstruction term is the binary cross entropy between prediction and target, which in the sparse graph representation are threefold with A, E, F.

The predicted output of the decoder is split in three parts and while $\tilde{A}$ is activated through sigmoid, $\tilde{E}$ and $\tilde{F}$ are activated via edge- and nodewise softmax. The graph matching permutation for A is applied to the target $A' = XAX^T$ and for E and F on the prediction, $\widetilde{F}' = X^T \widetilde{F}$ and $\widetilde{E}'_{\cdot,\cdot,l} = X^T \widetilde{E}_{\cdot,\cdot,l} X$, l being the one-hot encoded edge attribute vector which is permuted. These permuted subgraphs are then used to calculate the maximum log-likelihood estimate [5]

$$\begin{aligned} \log p\left(A' \mid \mathbf{z}\right) = & 1/k \sum_a A'_{a,a} \log \widetilde{A}_{a,a} + \left(1 - A'_{a,a}\right) \log \left(1 - \widetilde{A}_{a,a}\right) + \\ & + 1/k(k-1) \sum_{a \neq b} A'_{a,b} \log \widetilde{A}_{a,b} + \left(1 - A'_{a,b}\right) \log \left(1 - \widetilde{A}_{a,b}\right) \end{aligned} \tag{12}$$

$$\log p(F \mid \mathbf{z}) = 1/n \sum_{i} \log F_{i,\cdot}^{T} \widetilde{F}'_{i,} \tag{13}$$

$$\log p(E \mid \mathbf{z}) = 1/\left(\|A\|_1 - n\right) \sum_{i \neq j} \log E_{i,j,}^{T}, \widetilde{E}'_{i,j,\cdot} \tag{14}$$

Note that k can be equal n is target and prediction graphs have the same number of nodes. The normalizing constant $1/k(k-1)$ takes into account the no self-loops restriction, thus an edge-less diagonal. In the case of self loops this constant would be $1/k * K$. Similar $1/\left(\|A\|_1 - n\right)$ for $\log p(E \mid \mathbf{z})$ where $-n$ accounts for the edge-less diagonal and in case of self-loops would be discarded leaving us with $1/\left(\|A\|_1\right)$.

3.4 Ranger Optimizer

Finalizing this chapter, we introduce the novel deep learning optimizer Ranger. An optimizer, which in 2020 placed itself on the top of 12 FastAI leaderboards. Ranger combines Rectified Adam (RAdam), lookahead and optionally gradient centralization. let us shortly look into the different components.

RAdam is based on the popular adam optimizer. It improves the learning by dynamically rectifying Adam's adaptive momentum. This is done by reducing the variance of the momentum, which is especially large at the beginning of the training. Thus, leading to a more stable and accelerated start [17].

The Lookahead optimizer was inspired by recent advances in the understanding of loss surfaces of deep neural networks, thus proposes an approach where, a second optimizer 'looks ahead' on a set of parallel trained weights. while the computation and memory cost are negligible, it learning improves and the variance of the main optimizer is reduced [18].

The last and most novel optimization technique, Gradient Centralization, acts directly on the gradient by normalizing it to a zero mean. Especially on convolutional neural networks, this helps regularizing the gradient and boosts learning. This method can be added to existing optimizers and can be seen as constrain of the loss function [19].

Concluding we can say that Ranger is a state of the art deep learning optimizer with accelerating and stabilizing properties, incorporating three different optimization methods, which synergize with each other. Considering that generative models are especially unstable during training, we see Ranger as a good fit for this research.

4 Methods

This section describes the methods used in the experiments of this thesis. We will begin with data formatting and preprocessing, then the presentation the model, mainly the encoder and decoder and their variations. Moving on we explain our implementation of graph matching the loss function of the model and its evaluation metrics. To end of this chapter we describe the link prediction pipeline which is the main experiment to evaluate our model. Our implementation is written in Python using PyTorch, a high-performace deep-learning library [**pytorch**]. All experiments are aimed to be fully reproducible and the model meant to be used in future work, thus the code is openly available on Github [1].

4.1 Knowledge graph data

All our presented methods will operate on KG data. While data from other graph domains is possible, this work focuses solely on datasets in triple format. We will explain the sparse graph representation, which is the input format for our model and how to preprocess the original KG triples to match that format.

[1] `https://github.com/INDElab/rgvae`

4.1.1 Sparse representation

In this work, we opt for the sparse graph representation $G(A, E, F)$, where A denotes the adjacency matrix, E the edge feature matrix and F the node feature matrix. This allows models architectures as discussed in 3.2.4. The graph is binary and each matrix is stored in a separate tensor.

The adjacency matrix A takes the shape $n \times n$ with n being the number of nodes in our graph/subgraph. While most of previous work would only allow edges on the upper triangular adjacency matrix and fill the diagonal with ones, we chose a less constrained representation, which we assume is a better fit for representing KGs. In particular, we allow self-loops, meaning a triple where object and subject are the same entity and our relations are directed and can be inverted. Thus A can have a positive signal at any position $A_{i,j}$ $i, j \in R^{n \times n}$, indicating a directed edge between node of index i and node of index j, while $A_{i,j}$ differs from $A_{j,i}$.

The edge attribute matrix E takes the shape $n \times n \times n_e$ with n_e being the number of unique entities in our dataset. For each possible edge in the adjacency matrix we have a one hot encoded vector pointing to the unique relation in the dataset. Stacking these vectors leads to the three dimensional matrix E.

The shape of node attributes matrix F is $n \times n_e$ with n_e being the number of node attributes describing the nodes. Considering that we will split the KG in subgraphs, we use the entity index as node attribute, making it possible to assign every node in a subgraph to the entity in the full KG. Thus, the number of node attributes n_e equals the number unique entities in our dataset. Again the node attributes are one hot encoded vectors, which stacked result in the two dimensional F matrix.

4.1.2 Preprocessing

Our datasets consist of three tab separated value files full of triples for training, evaluation and final prediction.The preprocessing steps do not only account for generating subgraphs in the right format but also to ensure valuable research by withhold the triples in the test set until the final run.

From all three sets, we create a set of all occurring entities and similar set for the relations. Now we can define our dimensions n_e and n_r. For both sets we create two dictionaries *index-2-entity* and *entity-2-index* which map back and forth between numerical index and the string representation of the entity (similar for the relation set). These dictionaries are used to create a train and test set of triples with numeric indices. Depending on if we are in the final testing stage or not we include all triples from the training and evaluation file in the training set and use the triples in the testing file as test set, or we ignore the triples in the test file and use the evaluation file triples as test set.

Further we create two dictionaries, *head* and *tail* which for all occurring subject and relation combination, contain all entities which would complete it to a real triple in our dataset (similar for all relation and object combinations). This will allows us to filter true triples, an important part of link prediction and helpful in graph generation.

The final step of preprocessing is a function, which takes a batch of numerical triples and converts them to a batch of binary, multidimensional tensors A, E and F. While this might sound easy for only one triple per graph, it proves more complex for graphs with $n > 2$ facing exemption cases such as self loops or an entity occurring in two triples. We solve this by creating a separate set for head and tail entities, then storing the indices od both in a list, starting with the subject set and finally using this list as keys for a dictionaries with values in the range to n. In both edge cases, this results in an adjacency matrix with a rank lower than n. A similar approach, with less edge cases to consider, is used to apply the inverted translation from tensor matrix to triple.

4.2 RGVAE

The principle of a graph VAE has been explained in 3.2.4, what also covers the foundation of our model, the Relational Graph VAE (RGVAE). Therefore we will focus on the implementation as well as parameter and hyperparameter choice. Since this work is meant to be a prove of concept rather than aimed at outperforming the state of the art, our model is kept as simple as possible and only as complex as necessary. Our appr For the encoder we implemented two variations, a fully connected and a convolutional, while for the decoder we opted

for a single fully connected network.

4.2.1 Initialization

The RGVAE is initialized with a set of hyperparameter, which define the input shape. Table 4.2.1 shows a complete list of those parameters and their default values. It is left to mention that we use the Xavier uniform method with a gain of $0,01$ to initialize the weight parameter.

Hyerp.	Default	Description
n	2	Number of nodes
n_e	-	Total number of entities
n_r	-	Total number of relations
d_z	100	Latent space dimension
d_h	1024	Hidden dimension
dropout	0.2	Dropout
β	1	β value for regularization
perminv	**True**	Permutation invariant loss function
clipgrad	**True**	Learning w/ gradient clipping
encoder	**MLP**	Learning w/ gradient clipping

Table 1: The inital hyperparameter of the RGVAE with default value and description.

4.2.2 Encoder

The prove-of-concept encoder is a MLP as described in 3.2.2, which takes the flattened concatenated threefold graph $x = G(A, E, F)$ as batch input. We use the initial parameters to calculate the input

$$d_{input} = n * n + n * n * n_r + n * n_e \tag{15}$$

The main encoder architecture is a 3 layer fully connected network, with both layers using ReLU as as activation function. The choice for two hidden layers is based on the huge difference between d_{input} and d_z. The first layer has a dimension of $2 * d_h$ and the option to use dropout, which by default is set to 0.2. The second (hidden) layer has the dimension d_h which is by default set to 1024. After the second ReLU activation, the encoder linearly transforms the hidden state to an output vector of $2 \times d_z$. This vector is split and makes the mean and log-variance of size d_z for the reparametrization trick. Sampling ϵ from an autonomous module, we get the latent representation z of x, the final output of the encoder.

The second option for our RGVAE encoder is a GCN as described in 3.2.3. We adopt the architecture from [2] namely two layers of graph convolutions with dropout in between. To match the encoder output to the base model, we then add a flattening and a final linear transformation layer and to substitute the feature matrix used in Kipf's work, we reduce the edge attribute matrix E by one dimension and concatenate it with F resulting in $x_{GCN} \in R^{n \times (d_e + n * d_r)}$. We forward pass the adjacency matrix A and x_{GCN} through the first GCN layer with a hidden dimension of d_h and ReLU as activation function, followed by a dropout layer. It should be mentioned that dropout is only applied during learning, not on evaluation. The second GCN layer takes the hidden state and again A as input the two dimensional output from the previous layer. Now, instead of having the GCN predict on a number of classes, we have it output a logits vector of dimension $2 * d_z$. Therefore we pass the GCN output through a flattening and a linear transformation layer. Similar to above described encoder we use the reparametrization trick to output the latent reparametrization z.

Table comparing both architectures!!!

4.2.3 Decoder

For our RGVAE decoder, we use the same minimal approach as in [5], namely an MLP with inverted dimensions. The decoder architecture is similar to the one described in COMPARING TABLE for the MLP encoder version. Since we are decoding the latent space, the input dimension is d_z and the output dimension is d_{input} as

calculated in equation 15. The flat logits output tensor is split threefold and reshaped to the original input shape of $G(A, E, F)$.

To sample from the generated graph we apply the Sigmoid activation function to the logits of the first matrix and use the normalized output as weights for binomial distributions, from which we can sample the discrete $\tilde{A}$. For $\tilde{E}$ and $\tilde{F}$ we take the argmax on the last dimension of both matrices, Each node and edge can have only one attribute, referring to its index in $\mathcal{E}$ and $\mathcal{V}$, thus only the highest predicted value is relevant. The generated sample is a discrete graph $\tilde{G}(\tilde{A}, \tilde{E}, \tilde{F})$.

4.2.4 Limitations

The main limitation of the RGVAE is the parabolic increase of model parameters with the increase of nodes per input graph $\mathcal{O}(n^2)$. The number of parameters to train is directly linked with the GPU memory requirements. Even more computationally expensive is the use of permutation invariant graph matching, with a complexity of $\mathcal{O}(n^3)$. Thus, we propose this model only for generating small graphs with $n < 30$.

4.3 RGVAE learning

In this section we will present our implementation of how to fit the model to the data. We follow best practices of academical research as much as possible. Learning a model on data is a mostly standardized procedure, which includes training and evaluation per epoch. During training, the model forward passes the data, computes the loss, then does a backwards pass and updates its parameters. During evaluation, it is presented a split of the dataset unseen during training. Only the forward pass is done and the loss tracked during evaluation. Up to here the RGVAE does not differ from the vanilla VAE training. Special is the graph matching function which is applied to the predicted graph and the loss function which takes into account the optimal permutation. Thus, we will look deeper into graph matching and derive RGVAE loss. The trainig and all experimnets are performed on the GPU cluster LISA on a single node. The GPU equipped on this node is a Nvidia titan RX 25GB. To log our experiments results, we use *Weights Biases*, a cloud-based experiment tracking tool [20].

4.3.1 Max pooling graph matching

While the pseudocode presented in [15] is simple and straight forward, it proves complicated to implement this in algorithms for batches and thus, without looping over the indices. Yet, our batch implementation solves these challenges and is compared more efficient than the direct implementation, which we use for validating our results. Given the target graph G and the predicted graph $\tilde{G}$, the algorithm can be divided in three steps, calculating the five dimensional affinity matrix (the first being the batch dimension), max-pool matching the soft (continuous) similarity matrix X^* and discretizing X^* to our final permutation matrix X.

We use equation 8 for the first step but instead of adding the two terms to a single output, we return S twofold as S_r, five dimensional holding the information of edge affinity and S_e, three dimansional with the affinity information of the nodes. In a preprocessing step we zero out the diagonal of A, $\tilde{A}$ and for E and $\tilde{E}$ the diagonal of the second and third dimension, to compilie with the constrain $[i \neq j \wedge a \neq b]$ of the first term. For the second term we only take into account the diagonal of $\tilde{A}$ to compile with the constrain $[i = j \wedge a = b]$. Pseudocode 1 shows the implementation, here *diag()* stands for a vector with only the diagonal entries. For the dot product of E and $\tilde{E}$ over the last dimension we implement our own version of `torch.matmul()` to cope with higher dimensions. The operator $\odot$ denotes element-wise matrix multiplication.

The next step is the graph matching algorithm is the max-pool loop presented in [15]. We initialize the similarity matrix as ones $X^* \in 1^{bs \times n \times n}$ with bs denoting the batch size. For a certain number of iterations, Cho proposes 40 but the number should be adjusted to the number of nodes in the graph, we multiply X^* with a reduced version of S and use its Frobenius norm as normalizer. The algorithm 2 shows our implementation for batches.

To the best of our knowledge, this is the first time this algorithm is implemented in batch style. Thus, we would like to believe that laying out the implementation in detail will contribute to the academic value of this thesis.

Algorithm 1 Batch implementation for the affinity between two graphs

Input: $G(A, E, F$ and $\tilde{G}(\tilde{A}, \tilde{E}, \tilde{F})$

First term: $[i \neq j \wedge a \neq b]$

$E_{term1} = E^T \tilde{E}$ ▷ Dot product over the last dimension

$A_{term1} = A\,.\,\text{unsqueeze}(-1)^T(\tilde{A} \odot (\tilde{A}\tilde{A}^T))\,.\,\text{unsqueeze}(-1)$ ▷ Dot product over the last (empty) dimension

$S_r = E_{term1} \odot A_{term1}$

Second term: $[i = j \wedge a = b]$

$A_{term2} = \text{ones\_like}(\text{diag}(\tilde{A}))^T\,\text{diag}(\tilde{A})$

$F_{term2} = F^T \tilde{F}$ ▷ Dot product over the last dimension

$S_e = F_{term2} \odot A_{term2}$

return (S_r, S_e)

Algorithm 2 Max-pool graph matching for batches

Input: (S_r, S_e)

Init $X^* \in 1^{bs \times n \times n}$

for *iteration* $= 1, 2, \ldots$ **do**

  $S_{max} = \text{sum}(\max(S_r \odot X^*\,.\,\text{unsqueeze}([1, 1])))$ ▷ Sum and max over the last dimension. Unsqueeze two times on the second dimension

  $X^* = X^* \odot S_e + S_{max}$

  $X^* = X^* / \,\text{frobenius\_norm}(X^*)$

end for

return X^*

The last step in the graph matching pipeline is the discretization of X^*. We chose the Hungarian algorithm as presented in 3.3.3. To our disapointment and resulting in a bottleneck, no batch nor tensor implantation of the named algorithm has been published so far. Thus, we convert X^* to `numpy.array()` format and make use of the *Scipy* package [21]. First we create the cost matrix $X_{cost} = 1 - X^*$ and then, iterating over the batch size, use `scipy.optimize.linear_sum_assignment` which returns the optimal assignment matrix. This is our final permutation matrix X, indicating the best match between target and prediction. If no permutation is needed, X is the identity matrix of A.

4.3.2 Loss function

The RGVAE uses the ELBO loss from equation 4, consisting of two terms, the regularization loss and the reconstruction loss. We will present our implementation of both loss terms with graph matching and an alternative loss without graph matching for comparative evaluation of our results

We implement $\log p\,(A' \mid \mathbf{z})$ from equation 12 with the second normalizing constant as $1/k * K$ since we allow self-loops. The permutation matrix X is applied to to the target adjacency, resulting in A'. For $\log p\,(E' \mid \mathbf{z})$ and $\log p\,(F \mid \mathbf{z})$ the permutation is applied to the prediction, what in the case of E' requires our own implementation of matrix multiplication of $d > 2$. Taking into account self-loops we change the normalization constant of $\log p\,(E' \mid \mathbf{z})$ to $1/\,(\|A\|_1)$. It is left to mention that when implementing $\sum_{i \neq j} \log E^T_{i,j,}, \widetilde{E}'_{i,j,\cdot}$ in matrix multiplication style, we have to account for the zero values before taking the logarithm. We implement `torch.sum(torch.sum(torch.log(no_zero(`E `* `$\hat{E}$`)),-1)-1)` with `no_zero()` being a function which replaces 0 values with 1. This implementation of the loss function can be backpropageted with exception of the graph matching part, where the `numpy` implementation of the hungarian algorithm prevents backpropagation.

The regularization loss is given by the KL divergence between the approximated posterior $\log q_\phi\,(\mathbf{z} \mid \mathbf{x})$ and the Gaussian prior $p(\mathbf{z})$. The only modification we make to the original loss, is adding a β parameter which in values $100 < \beta < 500$ has shown great results in factorizing the latent space [**higgins**]. By setting $\beta = 1$ we return to the original loss function. This hyperparameter will be explored in the experiments.

Alternativly and as ground truth we implement the VAE loss 4 for graphs without graph matching. The reconstruction loss of the adjacency, we use calculate the binary cross entropy (BCE) and categorical cross entropy (CE) for rhe attribute matrices. The regularization loss is similar to the above presented and also

includes the hyperparameter β. The ELBO then is 16 with $\sigma()$ indicating Sigmoid activation.

$$\mathcal{L}(\phi, \theta : G) = BCE(A, \sigma(\tilde{A})) + CC(E, \tilde{E}) + CC(F, \tilde{F}) - D_{KL}\left(q_\phi\left(\mathbf{z} \mid G\right) \| p_\theta(\mathbf{z})\right) \tag{16}$$

We train the model on the negative ELBO. Further we use *Ranger* presented in 3.4 as optimizer combining three learning optimization methods. Out of the various optimization parameters, we achived good performance with the default values and only adjusted the learning rate and the number of lookahead steps. Missing a publication to cite, we refer to the source code *Ranger* [2]

4.4 Link prediction and Metrics

Our main experiment will be link prediction. It is intended as proof of concept rather than an attempt to set the state of the art. The results will let us draw conclusions on the impact and function of different parameters. Besides the final link prediction experiment, we will let the model perform link prediction on a randomly drawn small subset of the testset during trainig. This gives us a broader view on the models performance, which otherwise would only evaluated by the ELBO loss.

Link prediction on multi-relational KGs is the task of predicting unobserved triples, based in the information accurired during traiing. To evaluate a model on this task, the most common method is entity ranking in the form of triple completion of unseen triples from the testset. Given a KG $G(\mathcal{E}, \mathcal{V})$ we want our model find the right entity out of $\mathcal{E}$ which completes the unseen triple $(s, r, ?)$ or $(?, r, o)$ for heads or tail prediciton. Thus, the model scores the triple for all possible combination with the entities from $\mathcal{E}$. The rank of the true triple, in descending order, defines the performance of the model [22].

In the preprocessing step we created a dictionary with all occurring combinations for all possible triples with missing head or tail. This dictionaries we use to filter out real triples from the scoring. Unfiltered scores are referred to as *raw scores*. Per link prediction run, the model has to score the number of triples in the test set times the d_e the number of entities in $\mathcal{E}$ times two for head and tail, which mostly results in a number much larger than the size of the actual dataset.

Finally, the metrics for link prediction are the mean reciprocal rank (MRR) of the score for the true triple and the average HITS@k with $k \in [1, 3, 10]$. We denote $\mathcal{K}_{test}$ the unseen test set and $|\mathcal{K}_{\text{test}}|$ the number of triples in the test set. The operator rank() returns the in descending order the by the model scored position of the true triple. Head prediction of a triple given relation and object is denoted $(s|r, o)$ and likewise $(o|s, r)$ for tail prediction. The Iverson brackets $[\text{rank}(s \mid r, o) \leq k]$ return 1 if the scored rank is equal or lower than k, else o.

$$\begin{aligned} \text{MRR} &= \frac{1}{2\,|\mathcal{K}_{\text{test}}|} \sum_{(s,r,o)\in\mathcal{K}_{\text{lest}}} \left(\frac{1}{\text{rank}(s \mid r, o)} + \frac{1}{\text{rank}(o \mid s, r)}\right) \\ \text{Hits@ } k &= \frac{1}{2\,|\mathcal{K}_{\text{test}}|} \sum_{(s,r,o)\in\|_{\text{lest}}} \left(1[\text{rank}(s \mid r, o) \leq k] + 1[\text{rank}(o \mid s, r) \leq k]\right) \end{aligned} \tag{17}$$

4.5 Variational DistMult

In the context of link prediction, we implement control model for better interpretability of our results. From the wide range of embedding based models, DistMult reports both good results as well as an efficient architecture. Its bilinear property aligns with the permutation invariance of the RGVAE. additional to the original model we implement a variational version to isolate the effect of variational inference on multi-relational link prediction.

The DistMult encoder has a linear embedding layer for both sets of entities and relations. The embedded or latent representation is passed through the bilinear scoring function, which is equation 2. During training the model scores the triples in the training set among a number of corrupted triples. Loss function and optimizer are tunable hyperparameter. Table REF shows the optimal hyperparameter settings by Ruffellini at al. for the FB15k-237 dataset.

[2] `https://github.com/lessw2020/Ranger-Deep-Learning-Optimizer`

TODO DistMult parameter table

The model implementation is adopted from Peter Bloem's work [3]. Additionally a variational module is implemented, which uses the embedding vector representation as mean and logvariance for a latent distribution from where the latant representation is sampled using the reparametrization trick. When this method is not selected, the stochastic coeficient ϵ is set to 1 rather than a random sample from the standard normal distribution. Lastly we implement the ELBO loss, which adds a regularization term, including β parameters to the BCE loss, thus we can only use the variational DistMult (VDistMult) in combination with BCE loss. The scoring function of the VDistMult stays identical to the original.

5 Experiments & Results

This section presents the experiments and results aimed at evaluating our proposed graph generative model. First we run a grid-search on the hyperparameter space to find the optimal configuration of the RGVAE. We use the ELBO and MRR as evaluation metric. The best configurations are used to perform link prediction. Here we compare the model performance with vs. without convolutions. We use the VDistMult as control model for link prediction. Finally we run two proof-of-concept experiments. The first generating triples and filter on a entity class constraining relation, thus we get an insight of how much percent of the generated triples are valid. Secondly we analyze the results of a RGVAE trained on subgraphs with $n = 10$. For the experiments we use two multi-relational KG datasets.

5.1 Data

For this sake of comparison with state of the art results, we chose the two most popular dataset used in this field of KG link prediction, FB15k237 and WN18rr.

Fb15k-237

6.69423in

This dataset is a subset of the FreeBase KG (cite).[Explain Free Base] The original Fb15k had redundant triples, creating ambiguity during training. In the updated version, Fb15k-237, these triples were removed what led to a more robust dataset, which is commonly used as benchmark for several NLP tasks.

A subset from one of the first and largest KGs, called FreeBase. In the first version of this dataset, it was possible to infer most of the test triples by inverting triples in the trainset, Thus the latest 237 version filtered these triples out. The dataset contains $14,951$ entities and $1,345$ different relations.

WN18rr

Based on the wordnet KG.

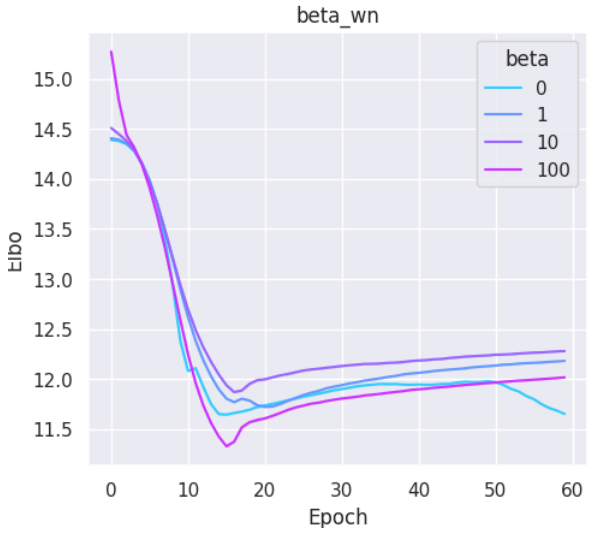
5.2 Hyperparameter Tuning

In this section we run a grid search for the three hyperparameters β, d_z and d_h for a set of contrastive values. To reduce the computational expenses we train each model for 60 epochs and evaluate link prediction on a subset of 50 triples.

Empirically we set the learning rate to $3e-5$ and the batchsize to the maximum fir on the GPU memory. For d_h we did not see any significant changes for higher values, thus we choose a lower number to reduce the total model parameters. The remaining hyperparameter did influence and the optimal setting vary for each dataset, table **??**hows the results of our hyperparameter tuning.

We compare the values of $\beta \in [0, 1, 10, 100]$

[3] `https://github.com/pbloem/embed`



RGVAE: hidden dimensions latent dimensions beta

RGCVAE

5.3 Link Prediction

We try with and without permutation

We try the model as encoder only

We use 1/3 of the test set only, randomly drawn. Run 3 times? Final models only 60 epochs

Different models Report: loss converges MRR stays almost zero

With vs. w/o graph matching

5.3.1 DistMult lp

To find out why our model fails to perform lp we run the original DistMult with optimal parameters from [22] and a variational DistMult version, which samples from a latent distribution, just like the VAE. Give implementation link.

Answer question:Link prediction with control model: Distmult vs Var Distmult

We see that the variational part messes everything up.

Graphs: MRR + Hits@all + Loss

5.4 Impact of permutation

Graph of permutation during training loss with vs without

5.5 Interpolate Latent Space

We take two random triples and interpolate the latent space of these two triples. The interpolations result in: HERE AN EXAMPLE.

Further we go ahead and test what happens if we modify one latent dimension of a triple. HERE AN EXAPMPLE.

Can the model assign logical features to latent dimensions?

5.6 Syntax coherence

Check model trained with vs without perm invariance.

Check if generated triples follow basic logic.

- Generate triples by random signals
- Filter these triples on a certain relation
- Check if the entities are part of the linked class

Since there is no preset for how to check the semantics of a KG, we will use simple basic logical criteria. The generated triples are filtered for the relation 'is capital of', thus the subject entity should be a city and he object entity member of the class 'country', Hope this gives good results.

5.7 Subgraph Generation

Until now our model trained on only one triple per sparse graph. What will happen if we train it on more than one triples?

6 Discussion & Future Work

We will discuss certain aspects of our results and give advice on further research.

6.1 Discuss Aspect 1

Well well well

6.2 Future Work

There are many avenues to follow for future work.

Basing on the believe that the experiments were successful, we recommend:

- Improve the model (deeper)
- link the latent space to word signals e,g text
- prior not normal, e.g NF
- try a GAN

Good luck amigos!

References

1. Kipf, T., van der Pol, E. & Welling, M. Contrastive Learning of Structured World Models. *arXiv:1911.12247 [cs, stat].* arXiv: 1911.12247. http://arxiv.org/abs/1911.12247 (2020) (Jan. 5, 2020).
2. Kipf, T. N. & Welling, M. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv:1609.02907 [cs, stat].* arXiv: 1609.02907. http://arxiv.org/abs/1609.02907 (2020) (Feb. 22, 2017).
3. Schlichtkrull, M. *et al.* in *The Semantic Web* (eds Gangemi, A. *et al.*) Series Title: Lecture Notes in Computer Science, 593–607 (Springer International Publishing, Cham, 2018). ISBN: 978-3-319-93416-7 978-3-319-93417-4. http://link.springer.com/10.1007/978-3-319-93417-4_38 (2020).
4. Kipf, T. N. & Welling, M. Variational Graph Auto-Encoders. *arXiv:1611.07308 [cs, stat].* arXiv: 1611.07308. http://arxiv.org/abs/1611.07308 (2020) (Nov. 21, 2016).
5. Simonovsky, M. & Komodakis, N. GraphVAE: Towards Generation of Small Graphs Using Variational Autoencoders. *arXiv:1802.03480 [cs].* arXiv: 1802.03480. http://arxiv.org/abs/1802.03480 (2020) (Feb. 9, 2018).
6. Belli, D. & Kipf, T. Image-Conditioned Graph Generation for Road Network Extraction. *arXiv:1910.14388 [cs, stat].* arXiv: 1910.14388. http://arxiv.org/abs/1910.14388 (2020) (Oct. 31, 2019).
7. Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J. & Yakhnenko, O. in *Advances in Neural Information Processing Systems 26* (eds Burges, C. J. C., Bottou, L., Welling, M., Ghahramani, Z. & Weinberger, K. Q.) 2787–2795 (Curran Associates, Inc., 2013). http://papers.nips.cc/paper/5071-translating-embeddings-for-modeling-multi-relational-data.pdf (2020).
8. Nickel, M., Tresp, V. & Kriegel, H.-P. A Three-Way Model for Collective Learning on Multi-Relational Data, 8.
9. Yang, B., Yih, W.-t., He, X., Gao, J. & Deng, L. Embedding Entities and Relations for Learning and Inference in Knowledge Bases. *arXiv:1412.6575 [cs].* arXiv: 1412.6575. http://arxiv.org/abs/1412.6575 (2020) (Aug. 29, 2015).
10. Paulheim, H. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic Web* **8** (ed Cimiano, P.) 489–508. ISSN: 22104968, 15700844. https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/SW-160218 (2020) (Dec. 6, 2016).
11. Nickel, M., Murphy, K., Tresp, V. & Gabrilovich, E. A Review of Relational Machine Learning for Knowledge Graphs. *Proceedings of the IEEE* **104,** 11–33. ISSN: 0018-9219, 1558-2256. arXiv: 1503.00759. http://arxiv.org/abs/1503.00759 (2020) (Jan. 2016).
12. Kingma, D. P. & Welling, M. Auto-Encoding Variational Bayes. *arXiv:1312.6114 [cs, stat].* arXiv: 1312.6114. http://arxiv.org/abs/1312.6114 (2020) (May 1, 2014).
13. Bishop, C. M. *Pattern recognition and machine learning* CERN Document Server. ISBN: 9781493938438 9780387310732 Publisher: Springer. https://cds.cern.ch/record/998831 (2020).
14. Ha, D. & Schmidhuber, J. World Models. *arXiv:1803.10122 [cs, stat].* arXiv: 1803.10122. http://arxiv.org/abs/1803.10122 (2020) (Mar. 28, 2018).
15. Cho, M., Sun, J., Duchenne, O. & Ponce, J. *Finding Matches in a Haystack: A Max-Pooling Strategy for Graph Matching in the Presence of Outliers* in *2014 IEEE Conference on Computer Vision and Pattern Recognition* 2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (IEEE, Columbus, OH, USA, June 2014), 2091–2098. ISBN: 978-1-4799-5118-5. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6909665 (2020).
16. Date, K. & Nagi, R. GPU-accelerated Hungarian algorithms for the Linear Assignment Problem. *Parallel Computing* **57,** 52–72. ISSN: 0167-8191. http://www.sciencedirect.com/science/article/pii/S016781911630045X (2020) (Sept. 1, 2016).
17. Liu, L. *et al.* On the Variance of the Adaptive Learning Rate and Beyond. *arXiv:1908.03265 [cs, stat].* arXiv: 1908.03265. http://arxiv.org/abs/1908.03265 (2020) (Apr. 17, 2020).
18. Zhang, M. R., Lucas, J., Hinton, G. & Ba, J. Lookahead Optimizer: k steps forward, 1 step back. *arXiv:1907.08610 [cs, stat].* version: 1. arXiv: 1907.08610. http://arxiv.org/abs/1907.08610 (2020) (July 19, 2019).
19. Yong, H., Huang, J., Hua, X. & Zhang, L. Gradient Centralization: A New Optimization Technique for Deep Neural Networks. *arXiv:2004.01461 [cs].* arXiv: 2004.01461. http://arxiv.org/abs/2004.01461 (2020) (Apr. 7, 2020).
20. Biewald, L. *Experiment Tracking with Weights and Biases* Software available from wandb.com. 2020. https://www.wandb.com/.

21. Virtanen, P. *et al.* SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* **17,** 261–272 (2020).

22. Ruffinelli, D., Broscheit, S. & Gemulla, R. *You CAN Teach an Old Dog New Tricks! On Training Knowledge Graph Embeddings* in. International Conference on Learning Representations (Sept. 25, 2019). `https://openreview.net/forum?id=BkxSmlBFvr` (2020).