RWTH Aachen University

Bachelor Thesis

Accelerating KGlove Graph Embedding

Abdulrahman Altabba

Thesis supervisor:
Prof. Dr. Stefan Decker
Thesis advisor:
Dr. Michael Cochez

Chair for
Computer Science 5
Information Systems
RWTH Aachen University
52056 Aachen
Germany

January 25, 2019

Table of Contents

Abstract. The exponential growth of data and the need of processing it requires efficient algorithms that are capable of gradually interpreting this immense amount of new information gathered in all kind of structures. In this thesis we list several approaches to accelerate a recent graph embedding method and deploy it in tuning the hyper-parameters for the whole graph embedding model taking a graph as an input and delivering a set of embeddings for the instances in this graph. We provide some theoretical background and thoughts and explain the details of accelerating the underlying GloVe training model and how to overcome some related issues. Eventually we build a random search model that is used to determine the suitable hyper-parameters for an efficient graph embedding model.

1 Introduction

Graphs are an essential element of recent data modelling and has been a great structure for saving arbitrary data and meta data (e.g. RDFs) in the industry. Analysing the content of these graphs has become recently a substantial task in data driven applications. Particularly translating data represented in graphs into a propositional feature vector representation that can be forwarded to different machine learning tasks (e.g. node classification, node recommendation, link prediction, etc.) has drown the interest of many researchers in a lot of artificial intelligence and data science fields [16][28][7]. Here comes the substantial role of graph embedding, which produce vector representations for every instance of the input, i.e. entity and relation vectors for graphs, many algorithms and methods have been proposed to deliver vectors that preserve semantic and syntactic regularities. In [10] a summarized overview about state of art used graph embedding methods is given with industrial use cases. In this research the focus will be on embedding graphs using the approach proposed in [5] by building a cooccurrence matrix for the graph structure and applying the GloVe [24] algorithm using the produced matrix. The resulting embeddings should have properties such as the vectors of two similar entities are close to each other in the vector space due to the similarity of their neighbourhoods in the graph. These vectors would ease and fasten tasks such as predicting whether a new edge is likely to exist between two entities (e.g., for suggesting new friends in social networks) or to determine a certain characteristic for a subgraph (e.g., revealing spam messages or catching plagiarism). In order to make the best use of the mentioned approach the user has to provide the right parameters for both, the algorithm that creates the cooccurrence matrix out of the graph (KGloVe), and the GloVe algorithm that minimizes a loss function by gradually updating the word and context vectors till they reach a high precision degree. These two phases rely on many different variables and parameters that strongly influence the output vectors, notably a combination of parameters might be suitable for a kind of tasks but not for all tasks. Setting the right parameters for different use cases requires running the model repeatedly with different combinations and evaluating every result. The training phase is, however, an expensive program that consumes a lot of time and memory. In this piece of work we will take a look at parallelized implementations of GloVe and reimplement it using different variants of parallelizing approaches. Finally we will evaluate the complete work and come out with an accelerated version of GloVe runnable on a GPU, it will then be used to find out the best parameters for different kind of machine learning scenarios.
As the GloVe training algorithm relies on optimizing a loss function, it updates the features of the vectors that represent the instances continuously, till they get aligned and keep a certain distance from each other in the euclidian space, which gives the vector an adequate meaning relative to other instances. Updating the vector in a serialized manner causes no collision problem, however when this algorithm is distributed over thousands of threads, one has to consider data races and collisions. Throughout this work we will try reducing the collisions of threads when they operate on the same vector during the training process

and managing the available memory on the GPU efficiently to build the baseline project for a portable accelerated graph embedding model. We will run different implementations of GloVe on a GPU with a theoretical discussion regarding their performance and memory occupancy. After explaining the implementations we will list all the parameters needed for the complete model and implement a random search algorithm that will examine the model using combinations of given parameter ranges.

2 Background and Related Work

Graphs are used in a wide range of fields to denote different kind of information (e.g. maps, communication networks, social networks, word co-occurrence networks, biological interactions etc.) and they are used as databases to store, map and query relationships. Analysing graphs and interpreting them with AI- and ML-algorithms gives a great insight to the information stored in it and makes it possible to conclude facts and make predictions. For example, analysing social networks graphs would help to find relationship kinds between people and suggest new connections rather than observing common interests among some social groups, which could be used to personalize advertisements or to predict a certain behaviour. Moreover, a recent approach have been published by Haofeng Jia and Erik Saule [14] to help researchers to find relevant scientific papers for a given topic, it uses the embeddings of a citation graph for *Citation Recommendation*: "Given a set of seed papers S, return a list of papers ranked by relevance to the ones in S" [14]. The main problem by using graph representations of data is that the instances (i.e. entities and relationships) have no numerical identities that preserve their characteristics and features on the one hand and serve as an input for machine learning models on the other hand. Many scientific works have been published to solve this problem and create a numerical representation (mainly vectors) for the instances of a graph, where the features of every instance are preserved in its vector. Some approaches encode the features of a node (or edge) directly in the vector representation, so a feature could have a direct meaning like the number of incoming/outgoing edges as the case in some Graph Convolutional Network models. Other approaches generate random vectors for the instances and train the vectors using the neighbourhood of the corresponding instance in the graph, producing vectors with semantic meanings demonstrated via vectors operations (+, -, $\langle\rangle$), for example, after creating a vector representation for instances like queen, woman, king, man, the location of the vectors in the vector space would denote semantic meanings of the underlying represented instances and these semantic meanings can be interpreted through by applying mathematical operations on the vectors. So if we subtract the vector $\overrightarrow{man}$ of the vector $\overrightarrow{king}$ the result would be a vector that represents the property of royalty and adding this vector to $\overrightarrow{woman}$ is supposed to result in a vector that is very close to $\overrightarrow{queen}$ in the vector space. We propose in the following paragraphs some recently used approaches for graph embedding:

2.1 TransE

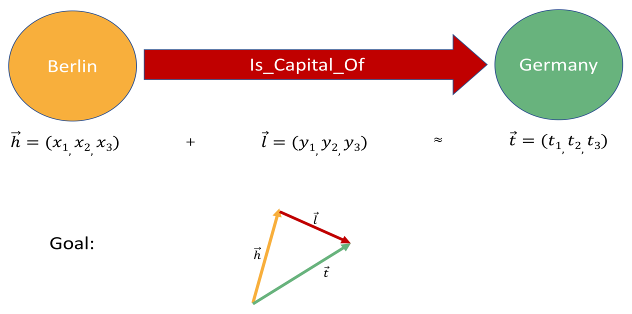

Berlin
Is_Capital_Of
Germany
$\vec{h} = (x_1, x_2, x_3)$
+
$\vec{l} = (y_1, y_2, y_3)$
≈
$\vec{t} = (t_1, t_2, t_3)$
Goal:
$\vec{l}$
$\vec{h}$
$\vec{t}$

Fig. 1: TransE example triplet.

This approach introduced in [2] is one of the first approaches for embedding graphs in vector space directly using the triplet sets of the graph $(head, relation, tail)$. The idea behind it is straight forward:

- generate a random vector with length k for each node and edge in the graph.
- if the a triplet (h, l, t) holds, then minimize the distance (*Manhattan Distance*) between the vectors $\overrightarrow{h + l}$ and $\vec{t}$.
- if the a triplet (h, l, t) does not hold, keep a minimum distance (margin γ) between the vectors $\overrightarrow{h + l}$ and $\vec{t}$.

The loss function is defined as the following:

$$\mathcal{L} = \sum_{(h,l,t)\in S} \sum_{(h',l,t')\in S'_{(h,l,t)}} [\gamma + d(h + l, t) - d(h' + l, t')]_+ \qquad [2]$$

Where $[x]_+ = \max(0, x)$ and $S'_{(h,l,t)} = \{(h', l, t)|h' \in V\} \cup \{(h, l, t')|t \in V\}$.

An important condition is that the L_2 norm of an entity's vector has to be 1, while the relations' vectors are unconstrained. This condition prevents the training process to trivially minimize $\mathcal{L}$ by artificially increasing entity embeddings norms. Notably some relations (e.g. "city_in") are treated as corrupted triplets in some cases in TransE. Let us assume the distance between $\overrightarrow{NewYork} + \overrightarrow{city\_in}$ and $\overrightarrow{USA}$ is being minimized. During the minimization some other distances between other cities in USA will get negatively affected while it should not. Therefore *TransE* lack a proper treatment of reflexive, many-one, one-many and many-many relations, however it still deliver a good result in the overall process.

2.2 TransG

Han Xiao, Minlie Huang and Xiaoyan Zhu have presented lately in their paper [29] a generative model for knowledge graph embedding. The speciality of this

paper that it addresses the issue of ***multiple relation semantics*** assuming that the meaning of a relation $\boldsymbol{r}$ in a triplet (h, r, t) could eventually be different from its meaning in another triplet (h', r, t') in the graph. For example the relation HasPart has at least two latent semantics: composition-related as (Table, HasPart, Leg) and location-related as (Atlantics, HasPart, NewYorkBay). Another example in Freebase, (Jon Snow, birth place, Winter Fall) and (George R. R. Martin, birth place, U.S.) are mapped to schema /fictional universe/fictional character/place of birth and /people/person/place of birth respectively, indicating that birth place has different meanings [29]. One reason of this phenomenon in knowledge bases is *artificial simplification*, this means in the most if not all knowledge bases similar relations are abstracted into one relation that reflects the average meaning of these relations. Second reason is *the nature of knowledge*, because ambiguous words exist already profusely in a language. An example for that is "Expert", it could refer to an expert doctor, an expert engineer or an expert writer etc.
As long as every relation has one vector translation in the vector space, different semantics of the relation will not be distinguishable. Therefor *TransG generates multiple translation components for a relation* [29]. For example, the two different meanings HasPart.1 and HasPart.2 will have two different components in the embedding model. *The possible semantics are automatically clustered to represent the meaning of associated entity pairs.* In [29], they propose to use Bayesian non-parametric infinite mixture embedding model (Griffiths and Ghahramani, 2011). In TransG they use the normal distribution as a vector distributor instead of random distribution as previously done in TransE. The approach works as the following:

- For every entity $e \in E$:
  – *Draw each entity embedding mean vector from a standard normal distribution as a prior:*
  $$u_e \sim \mathcal{N}(0,\ 1)$$
- For a triplet $(h, r, t) \in \Delta$:
  – Draw a semantic component from Chinese Restaurant Process for this relation:
  $$\pi_{r,m} \sim \mathcal{CRP}(\beta)$$
  – Draw a head entity embedding vector from a normal distribution::
  $$h \sim \mathcal{N}(\mu_h, \sigma_h^2 E)$$
  – Draw a tail entity embedding vector from a normal distribution:
  $$t \sim \mathcal{N}(\mu_t, \sigma_t^2 E)$$
  – Draw a relation embedding vector for this semantics:
  $$u_{r,m} = t - h \sim \mathcal{N}(\mu_t - \mu_h, (\sigma_h^2 + \sigma_t^2)E)$$

where μ_t and μ_h stand for the mean embedding vector for tail and head. The mean embedding vector indicates the peak of the normal distribution. σ_t and σ_h are the variances, $u_{r,m}$ is the translation vector of component m of relation r. $\mathcal{CRP}$ stands for Chinese Restaurant Process, which is an unfair distribution that detects the semantic components of a relation taking in account its triplet and gives a specific semantic component according to the triplet the relation is coming from a higher weight $\pi_{r,m}$ over the others.
So we notice that in every triplet (h, r, t) containing the relation r, one of the semantic components of r has a significant weight over the other components, which represents the right interpretation of r in this specific triplet. The score function looks as the following [29]:

$$\mathbb{P}\{(h, r, t)\} = \sum_{m=1}^{Mr} \pi_{r,m} e^{-\frac{\|\mu_h + u_{r,m} - \mu_t\|_2^2}{\sigma_h^2 + \sigma_t^2}}$$

Mr is the number of semantic components for a relation r, it is learned from the data automatically by the CRP. $\pi_{r,m}$ is is the mixing factor, indicating the weight of m-th component. Other methods like TransE and similar ones aim for a given triplet (h, r, t) to maintain $h + r \approx t$. However TransG has a slight difference because of the multiple semantic components of a relation:

$$h + u_{r,m^*_{(h,r,t)}} \approx t$$

$$m^*_{(h,r,t)} = \arg\max \left(\pi_{r,m} e^{-\frac{\|\mu_h + u_{r,m} - \mu_t\|_2^2}{\sigma_h^2 + \sigma_t^2}} \right)$$

where $m^*_{(h,r,t)}$ is the index of the component with the highest weight, depending on the right interpretation determined by the head.

2.3 Graph Convolutional Networks

The aim in this kind of approaches is to adapt arbitrarily structured data (graphs) to serve as an input for CNNs and RNNs, so that it can be trained using a partial set of target values depending on the application. So the input would be[1]:

- The adjacency matrix of the given graph $A(N \times N)$
- A matrix $X(N \times D)$ containing the feature vectors, where D is the number of input features

Every layer on the convolutional network produces a node-level output matrix $Z(N \times F)$, where F is the number of output features per node. Layers in the network can be defined in a non-linear function:

$$H^{(l+1)} = f(H^{(l)}, A)$$

[1] The explanation is based on the article: https://tkipf.github.io/graph-convolutional-networks/

with $H^{(0)} = X$ and $H^{(L)} = Z$ assuming L is the number of layers. The difference between specific models lies only in the definition and parameterization of $f(.,.)$. A simple and powerful model was introduced by Thomas N. Kipf and Max Welling [19]:

$$f(H^{(l)}, A) = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)})$$

Where $\hat{A} = A+I$ is meant to enforce self loops into the adjacency matrix, so that the input feature vector of a node contains information about the node itself. $\hat{D}$ is the diagonal node degree matrix of $\hat{A}$. By multiplying with A, we sum up for every node all the feature vectors of all neighbouring nodes including the self loop since the identity matrix is added to $\hat{A}$. The symmetric normalization is to avoid unstable gradients calculation during training.

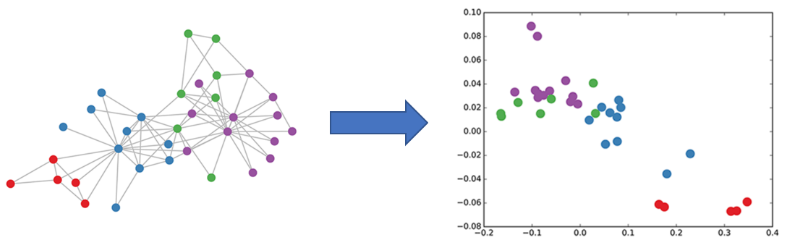


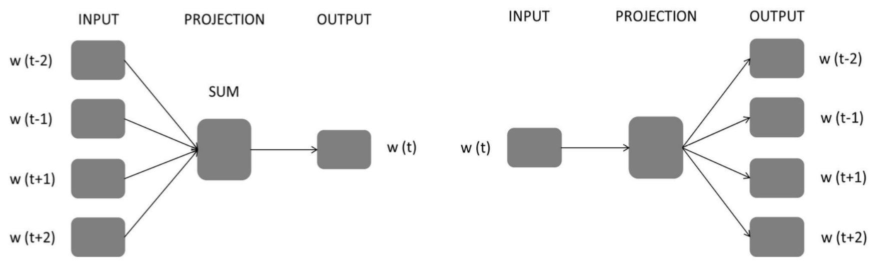
Fig. 2: Embedding the karate club network [3] [19]

One noticeable thing in this approach is that it does not include the edges of the input graph in the feature representation. Similar approach that includes edges in the embedding is Relational Graph Convolutional Networks [27].

2.4 Node2Vec

Interpreting data requires a unique identity of every data instance. For instance, when dealing with complex data like pictures and audio waves, there is enough information encoded in these data instances that gives the machine a relatively clear differentiation between them. However, when dealing with words in NLP tasks, a word string or any mapped id would not give enough information to describe this word since the context, in which the word occur, plays a role in defining this word's essence. Therefore it needs to be embedded into a vector space, where every word is represented in a feature vector that describe this word and express its unique meaning to the machine and word similarities are expressed through vector distance. In Node2Vec [11] and similarly RDF Embedding [4] the approach they use to for embedding a graph is the following three steps:

- Extract Information of the graph:
  Many algorithms were deployed to perform a text extraction from a graph like an edited version of ***Weisfeiler-Lehman graph kernels***, ***Random Graph Walks*** and ***Biased Graph Walks*** [4].
- Summarize the extracted information in a text corpus.
- Use the extracted text as an input for a neural language processing model presented by Mikolov et al. [21].



a) *CBOW architecture* **b)** *Skip-gram architecture*

Fig. 3: NLP Models [4]

The two NLP models used in Word2Vec are presented in fig. 3, where CBOW predicts a word from a given context and Skip-gram predicts a context from a given word. Training Node2Vec on a large graph requires some optimizations due to huge the number of backpropagations needed to update the weight matrices for every single instance. Therefore the optimizations presented in Mikolovś paper can be adapted in this model as well, which are ***Hierarchical Softmax*** or ***Negative Sampling***.

2.5 KGloVe

GloVe [24] is another model designed for NLP tasks, the concept behind it is to study the relationship between two selected words by analysing their relevance to another set of words. An example presented in the original paper is analysing the relationship between *ice* and *steam* by checking their relevance to other words (*solid*, *gas*, *water*, *fashion*). The relevance is measured with the ratio of the probability that a word k occurs in the context of the word i to the probability that k occurs in the context of the word j, demonstrating this on the given

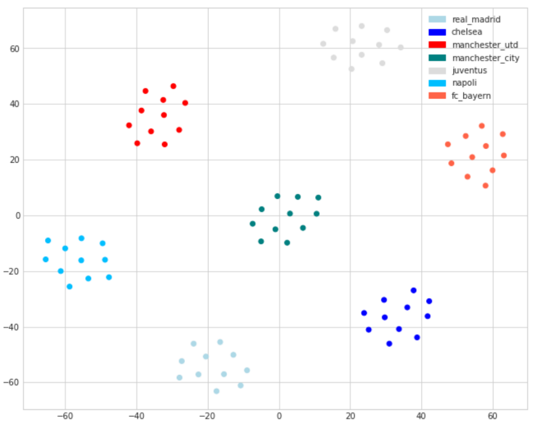


Fig. 4: An experiment done by Elior Cohen using Node2Vec to embed the graphs of football teams from a FIFA dataset on Kaggle [6]

example would give:

$$\frac{P_{ik}}{P_{jk}} >> 1 \qquad \text{for}\{i = ice, j = steam, k = solid\}$$

solid is relevant to ice but not steam

$$\frac{P_{ik}}{P_{jk}} << 1 \qquad \text{for}\{i = ice, j = steam, k = gas\}$$

gas is relevant to steam but not ice

$$\frac{P_{ik}}{P_{jk}} \approx 1 \qquad \text{for}\{i = ice, j = steam, k = fashion\}$$

fashion is irrelevant to ice nor steam

By creating a model that realizes this characteristic among the vector representations of the words, they start by assuming that a function f exists, which takes the vectors w_i, w_j and $\widetilde{w}_k$ as input (w for word vector, $\widetilde{w}$ for context vector) and delivers the wanted ratio $\frac{P_{ik}}{P_{jk}}$ and they add some constraints till they derive a loss function J. So the overall process consists of the following:

- Gather statistics of word occurrences in a given text.
- Create for every word two vectors, one that represents the word itself and one that represents its context.
- Train the vectors to obtain a semantic meaning based on the probability of a word occurring in the context of another word.

In the first phase of *GloVe* the cooccurrences of the words in a text are gathered based on the given text, we create a Co-Matrix $X(N \times N)$ with N the number of different words in the text, and scan the text word by word, if a word

j occurs in the context of a word i we increment X_{ij}. Determining the value by which the entries are incremented can be tuned for direct neighboured words to be larger than indirect neighboured words in the window size. Further more hyperparameters tunings can be applied regarding the shape and size of the window, vector length and iterations number. After gathering the cooccurrences we initialize $(2 \times N)$ vectors with random values and train them to obtain the following property:

$$w_i^T \widetilde{w}_k = \log p_{ik}$$

which means: the inner product of the word vector for a word i (w_i) and the context vector for a word k $(\widetilde{w}_k)$ should deliver the logarithm of the probability of the word k occurring in the context of the word i . Then we could have the probability simply by applying $\exp(\log p_{ik})$.

To give the embeddings this characteristic we minimize the loss function J defined in [24] using the calculated cooccurrences:

$$J = \sum_{i,j=1}^{V} f(X_{ij})(w_i^T \widetilde{w}_j + b_i + \widetilde{b}_j - \log X_{ij})^2$$

The damping function f is added to limit the influence of high occurring words (e.g. *"the"*) to preserve the rare words in the vector space. f is defined in the original paper as the following:

$$f(x) = \begin{cases} (x/x_{max})^{\alpha} & x < x_{max} \\ 1 & \text{otherwise} \end{cases}$$

For training the vectors the author of GloVe Pennington et al. uses the AdaGrad optimzer with default parameters:

- $VectorSize = 50$
- $x_{max} = 100$
- $LearningRate = 0.05$
- $\alpha = 0.75$

In the paper [24], some experiments have been done to find the best vector size and context size using one dataset and running several evaluations on the trained vectors. The following figure from [24] shows the results for word analogy tests, where questions like "Paris to France is like Rome to ___?" are queried. The window size in (a) is 10 and the vector size in (b) and (c) is 100.

The results in fig. fig. 5 and other evaluations in [24] provide a good visualization for the parameter set regarding NLP tasks. As making the window of the gathered cooccurrences very small would produce vectors that are useful for syntactic correctness checking, however widening the window helps by checking semantic correctness of the sentences.

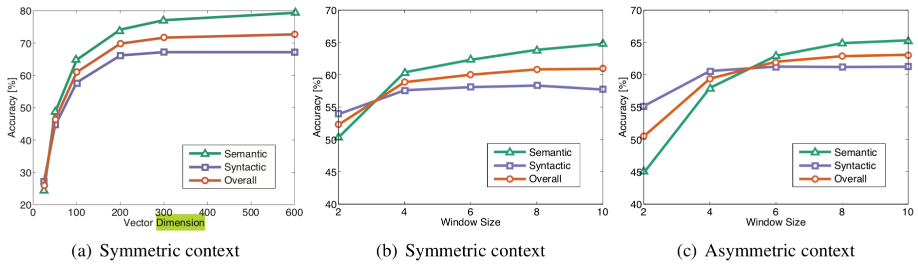


Fig. 5: Effect of changing the window shape and embedding size on the overall accuracy [24].

Graph Embedding using GloVe: In [5] a new Graph Embedding approach was introduced using the GloVe algorithm. It uses the same optimization model starting from a cooccurrence matrix that was created from a Graph instead of a text corpus. The entities of the graph are treated as words and the nearby entities and/or edges are considered to be the context surrounding the focus node. In Node2Vec [11] and RDF2Vec [4] they used graph kernels and random graph walks to extract a text consisting of the labels of the graph and then feed this text into the NLP model. In KGlove [5] there is no text extracting step, the co-occurrence matrix that summarizes the relatedness of the entities has to be extracted directly from the graph. The naive way of creating a cooccurrence matrix for a graph would be to perform breadth-first search for each node with a defined depth and consider the reachable nodes as the context nodes, where the cooccurrence rate is higher when the number of the hops between the context node and the original node is lower. There are several issues that are not handled in such an approach like 1) reaching the same node through several paths with different depth level 2) having loops in the graph 3) overrating of further many nodes over few close nodes. A possible solution would be to use the PageRank algorithm which is used originally for ranking webpages according to the outgoing edges they contain (references to another URLs). To build up the co-occurrence matrix we would need to compute the PPR for every single node of the graph, this would generate a very dense matrix with huge amount of unnecessary data that plays no role in the training. therefore in [5] they adapt the Bookmark-Coloring Algorithm (BCA) algorithm [1] to create the co-occurrence matrix in a fast and scalable way. The BCA algorithm's idea is to inject a fixed amount of paint into the focus node, and start pumping this paint through the connected edges to the neighbouring nodes, where α-portion of the paint is retained in the node itself and $(1-\alpha)$ is pumped out uniformly. This is repeated recursively for every node that received paint. If a node has no outgoing edges the outgoing paint is discarded. For the determinism of the repeating process another parameter is added ϵ, when a node has amount of paint that equals or less then ϵ it stops distributing paint. Beside that when a node retain α-portion of paint, this paint becomes dry so it can not be calculated in future distributions of paint in the case of a cycle. For each walk starting from nominated focus node, the paint can at most flow n hops far, where n can be calculated as:

$$(1-\alpha)^n \approx \epsilon \implies n \approx \frac{log(\epsilon)}{log(1-\alpha)}$$

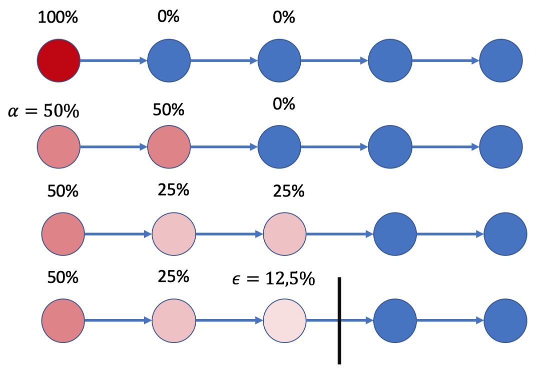


Fig. 6: Demo of BCA for $\alpha = 50\%$ and $\epsilon = 12,5\%$

In fig. 6 is a demonstration regarding BCA computation for one node, this computation has to be performed for all nodes, however this would cost too much computations for large graphs. To make this scalable we start with a set of significant nodes of the graph and add all nodes that are painted to this set every iteration. Moreover not all nodes of this set are chosen every iteration, but nodes with more paint are prioritized over others with little amount of paint, this guarantees to choose more influential nodes for the computations. More details are explained in [5] and [1].

In the previous illustration of the BCA algorithm we assumed that the paint is pumped equally through the out-edges, however, the pumping happens according to an importance factor so that some edges (or adjacent nodes) receive bigger amount of paint. Calculating the importance of an edge is the task of the weighting function. As shown in [4] there are twelve different approaches to assign weights to the edges in the graph, each with its different characteristics and influence on the resulting embeddings.

Possible weighting functions:

Uniform Approach

1. Uniform = Object Frequency Split

Edge-centric approaches

2. Predicate Frequency
3. Inverse Predicate Frequency
4. Predicate- Object Frequency
5. Inverse Predicate- Object Frequency

Node-centric object freq. approaches

6.Object Frequency
7.Inverse Object Frequency
8.Inverse Object Frequency Split

Node-centric PageRank approaches

9. PageRank
10. Inverse PageRank
11. PageRank Split
12. Inverse PageRank Split

After weighting the edges, pumping the paint out of an edge v that has d out edges would be biased towards more important nodes according to the formula:

$$Pr[follow\ edge\ v_{ol}] = \frac{Weight(v_{ol})}{\sum_{i=1}^{d} Weight(v_{oi})}$$

After we apply the algorithm on the nominated nodes of the graph according to their importance, we fill the cooccurrence matrix $X(N \times N)$, where N is number of nodes that got nominated for the BCA algorithm, with the amount of paint that reaches a node j by applying BCA on node j. The user could choose to normalize he values and to take only entities in consideration during the walks. So overall the user get to choose the following set of parameters for the creation of the co-occurrence matrix:

– Weighers (12×12) effecting the symmetry of the walks
– α and ϵ effecting the window size
– Normalize the values
– Taking only entities

The result of this process is a co-occurrence matrix ready to serve as an input for the GloVe training model, taking in consideration that some parameters like *x-max* has to be changed since the values of a matrix created from the graph differs from those created from a text corpus.

3 Implementation

In this thesis we throw an implementation for training the vectors in the KGlove embedding model. We basically use the CUDA API provided by Nvidia with

C++ and the code is inspired from the GloVe code[1]. The overall graph embedding model consists of many stages that depend on each other. Since the focus in this thesis is on the implementation of the training phase and the previous parts of the model are adopted from paper [4] we will assume having already created the cooccurrence matrix. The dependancies of the GloVe training model are:

1. Hardware capacity
   (a) Host memory
   (b) Device memory
2. Vector size
3. Batch size
4. Optimizer

3.1 Optimizers

Optimizing a function using its gradients is a widely researched topic with plenty different methods. Considering a machine learning problem like graph embedding, the trainable instances are usually sparse and high dimensional. In such a case the normal gradient descent optimizer is not very applicable to find the global minima (or maxima) of a function, mainly because it handles frequent features and infrequent features equally, which causes the effect of informative infrequent features to get subdued by the effect of frequent ones. In the coming sections we will explain briefly the two popular optimizers Adagrad [8] and Adam [17] which claim to suit this exact use case. The originally used optimizer in [24] is adagrad and in this thesis we experiment an implementation of GloVe with Adam optimizer. The used optimizer has an effect on the amount of required memory during the training. As we are using the optimzers to find the global minima of the GloVe loss function, we can directly calculate the gradients of it:

$$
\begin{aligned}
J &= \sum_{i,j=1}^{V} f(X_{ij})(w_i^T \widetilde{w}_j + b_i + \widetilde{b_j} - \log X_{ij})^2 \\
\frac{\partial J}{\partial w_i} &= f(X_{ij})\tilde{w_j}(w_i^T \widetilde{w}_j + b_i + \widetilde{b_j} - \log X_{ij}) \\
\frac{\partial J}{\partial \tilde{w_j}} &= f(X_{ij})w_i(w_i^T \widetilde{w}_j + b_i + \widetilde{b_j} - \log X_{ij}) \\
\frac{\partial J}{\partial b_i} &= f(X_{ij})(w_i^T \widetilde{w}_j + b_i + \widetilde{b_j} - \log X_{ij}) \\
\frac{\partial J}{\partial \tilde{b_j}} &= f(X_{ij})(w_i^T \widetilde{w}_j + b_i + \widetilde{b_j} - \log X_{ij})
\end{aligned}
$$

[1] based on the code in: https://github.com/stanfordnlp/GloVe

We notice that the core of all gradients is the calculation of $(w_i^T \widetilde{w}_j + b_i + \widetilde{b}_j - \log X_{ij})$ and this will be needed for any optimzer that is based on calculating the gradients.

Adagrad [8]
is an edition of adaptive optimizers that adapt their learning rates to every single feature of a vector so that infrequent instances get updated with larger steps depending on the sum of the previous squared gradients of the particular feature. Quote: "Informally, our procedures give frequently occurring features very low learning rates and infrequent features high learning rates, where the intuition is that each time an infrequent feature is seen, the learner should "take notice." Thus, the adaptation facilitates finding and identifying very predictive but comparatively rare features." The updating rule of Adagrad is:

$$x_{t+1,i} = x_{t,i} - \frac{\alpha}{\sqrt{\Sigma_{\tau=1}^{t-1} g_{\tau,i}^2}} g_{t,i}$$

The drawback noticed by Adagrad is that the learning rates converge to zero, so after a certain period of training the improvement becomes negligible.

Adam [18] optimizer combines the the advantage of AdaGrad to deal with sparse gradients, and the property of RMSProp to deal with non-stationary objectives, so it computes adaptive learning rates of the features based on two components, an exponentially decaying average of past squared gradients v_t and an exponentially decaying average of past gradients m_t:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \end{aligned}$$

Since m_t and v_t are initialized with zeros they bias towards zero, therefore the authors of Adam compute bias-corrected first and second moment estimates:

$$\begin{aligned} \hat{m_t} &= \frac{m_t}{(1 - \beta_1^t)} \\ \hat{v_t} &= \frac{m_t}{(1 - \beta_2^t)} \end{aligned}$$

so the update rule would be as the following:

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{\hat{v_t}} + \epsilon} \hat{m_t}$$

The authors of Adam propose the values for the parameters $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

3.2 Re-implementation of GloVe

The GloVe algorithm was originally programmed in C by the authors of [24]. In this work we conceive an inspired implementation with C++ and Cuda explaining in details how the conceptual model is structured and highlighting some differences. A cooccurrence type is defined as the following:

```
typedef struct cooccur_rec {
        int word1;
        int word2;
        double val;
} CREC;
```

In a GPU implementation, the reason behind most of the latency is usually transferring data, therefore training the model directly from the cooccurrence file on the hard disk, as the original c-glove does, is extremely inefficient. In our cuda-glove model we overcome this issue by calculating a sufficient batch size and dividing the whole cooccurrence matrix into number of batches that is a factor of number four(explanation comes later). The batch size depends on the host memory, the device memory and the vector size. A batch generator takes the cooccurrences binary file that contains the records and splits it into several batch files gathered in one subdirectory, during generating the batches the largest index will be saved and later on it will be considered as the vocabulary size. This way of getting the vocabulary size reduces the need of the vocab-file, which will be just needed in the end to map the vectors to their instances in the graph. The glove class is responsible during the training for copying the batches from the generated batch files to the host memory and then to the gpu memory. The second part that occupies a significant amount of memory are the vectors, in c-glove they are initialized on the host memory with random values sequentially. However, in cuda-glove they reside during the training in the device memory. Initializing them on the host memory and transferring them is not a good idea due to the latency caused by sequential initializing and transfer operation beside reserving memory on the host that will not be used during the training process. In cuda-glove we alternately allocate and initialize the vectors directly on the device.

3.3 Allocating memory

After determining the batch size that fits on the device memory given the vector size and the vocabulary size, the glove model is ready to be initialized with the following memory allocations:

1. Words' vectors $SizeOfReal \times VectorSize \times VocabularySize$
2. Contexts' vectors $SizeOfReal \times VectorSize \times VocabularySize$
3. Biases $SizeOfReal \times 2 \times VocabularySize$
4. An extra variable for every vector's feature and bias to store the sum of previous squared gradients which are used for the adaptive updates described in the Adagrad optimizer section.

Adagrad: $SizeOfReal \times (VectorSize + 1) \times VocabularySize \times 2$
For Adam we have to store the sum of past gradients m_t in addition to the sum of past squared gradients:
Adam: $SizeOfReal \times (VectorSize + 1) \times VocabularySize \times 4$

5. Batches $sizeOfRecord \times batchSize \times 2$
6. Total Cost $sizeOfReal$

This is the total memory needed for the GloVe model on the GPU if the Adagrad optimizer is intended to be used. As a demonstration of this, we take the vocabulary size of the DBpedia knowledge graph that we use later for the evaluations.

For $VocabSize = 8,876,675$ and using $float$ as real type with size 4, and assuming the vector size is 50, we end up using approximately 7.24 GB without the batches' memory. The memory usage can be calculated using the formula(using Adagrad): Required Memory=

$$SizeOfReal \times (4 \times (VocabSize(VectorSize + 1)) + 1) \text{ Bytes} \tag{1}$$

The left memory is a maximum for the batch size of the data used to train the vectors.

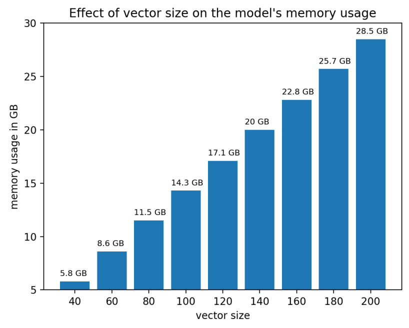


Fig. 7: vector size vs memory usage for DBPedia

3.4 Initializing

After allocating the memory needed for the model we have to initialize the values of the vectors, biases and the squared gradients. As these components are residing on the device memory, it is only possible to set their values with cudaMemset that sets the value of every byte for the given size to $v \in [0, 255]$ or to use a kernel for bigger datatypes. However, the word and context vectors are initialized originally with random values serially using the internal time as

a seed. On the device the initialization happens in parallel nearly at the same time, therefore taking the time as a seed for randomization is going to generate the same values in this case. For this reason the seed for every thread has to be unique in order to generate random values. The first idea is to take the unique ID of every thread as a seed for the random function:
`threadIdx.x + blockIdx.x * blockDim.x`
However, these seeds are arranged successively (0,1,2, ... , n) which will cause the generated values to be arranged successively as well. In our implementation we use the approach from [25] by using a light hash function created by Thomas Wang that takes the ID of a thread and returns a hash value, which is used as a seed for the random function.

```
 uint wang_hash(uint seed)
{
    seed = (seed ^ 61) ^ (seed >> 16);
    seed *= 9;
    seed = seed ^ (seed >> 4);
    seed *= 0x27d4eb2d;
    seed = seed ^ (seed >> 15);
    return seed;
}
```

Testing this method with a *"bitmap with one random bit per pixel, black or white"* it shows fairly good pseudorandom pixels where a human eye can not recognize any patterns.

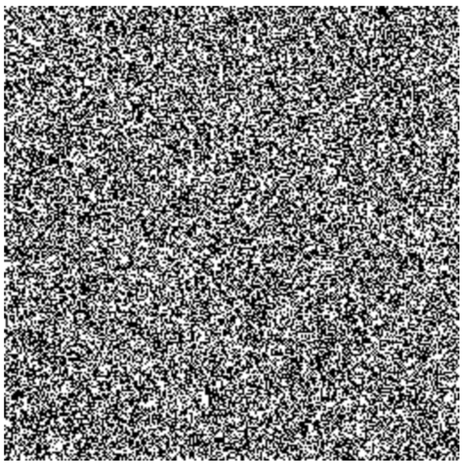


Fig. 8: [25]

We define a cuda kernel that uses this method to initialize the vectors and biasses with random values between $(-1, 1)$ and a second kernel that intitializes the squared gradient descents with initial value 1.

3.5 Generating batches

After the graph walks are completed and the cooccurrences are calculated and stored in a cooccurrences binary file with chunks of CRECs, glove takes this file as an input and iterates over the records using them to train the vectors. As we mentioned before, instead of copying every cooccurrence from the disk to the RAM and then to the device, we transfer them in batches depending on the left device memory after allocating the vectors and other components. We need two things to determine the maximum size of a batch, the result of (eq. (1)) and the device's capacity which can be known by calling *cudaGetDeviceProperties* and passing the address of a struct instance *cudaDeviceProp* [23]. Substituting the used memory from *totalGlobalMem* gives us the available memory. The max_batch_size would be the available memory (one could subtract some addition space to leave some free memory in order to avoid overfilling the memory) divided by 2 due to compatibility with the training patterns we are going to explain. After determining this we call the function `void writeBatches(const std::string co_filename, const std::string outPath)`. This function reads from the given file a record and writes it to a batch file (as long as the first word_index is not different) then it checks if the size of the written elements has exceeded the max_batch_size. This way we guarantee that the cooccurrences of a certain word do not land in two different batches. The *outPath* arg is where the batch files are stored.

3.6 Training

After we have all the components ready, we start the training by calling `void glove::train(unsigned nIterations, int save_every)`. This function is responsible for organizing data transfer between the Host (CPU) and the Device (GPU) and kernel launches. As shown in [20] the trivial processing flow in a GPU consists of:

– Copying data from CPU memory to GPU memory
– Launching a GPU Kernel
– Copying data from GPU memory to CPU memory

In the glove algorithm it is a bit different, in a training iteration the data (cooccurrences) are copied to the GPU, cuda kernels are launched to train the vectors using the data, and the data is dismissed in the end. After all the iterations (or somewhen in between when the vectors shall be written to a file) the vectors will be copied to the CPU memory and then written to a file. As mentioned before the cooccurrences file could not always fit on the GPU memory, therefore it will be split into several batches. So the processing flow in GloVe after allocating and initializing would be as the following:

– For every Iteration
  • Copying Cooccurrences matrix from CPU memory to GPU memory
  • Launching a GPU Kernel
– Copy the vectors from GPU to CPU and write them to a file.

3.7 Optimizing GPU Utilization:

There is generally two factors that mostly contribute to maximizing the utilization of a GPU.

- Choosing the right number of blocks and threads per block for the kernel launch.
- Minimizing the overhead of memory transfer between CPU and GPU, and inside the GPU.

A trivial implementation of GloVe would be to carry out the operations sequentially. copy batch, train, copy second batch etc. Looking at the processing flow in fig. 9, a dataset with a cooccurrences matrix consisting of six batches is being trained. The brown slices represent the periods when the GPU is waiting for data transfer. In this example one data transfer slice takes about $0,83$ seconds. For six batches it means around 4,8 seconds for data transfer.

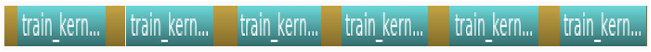


Fig. 9: sequential

Pinned memory [12] The default allocated memory on the host is allocated as pageable memory, this means it can be paged in or out between the ram and some other storage device like an SSD for memory managing purposes. It is however not possible for the GPU to access data directly from pageable memory space, therefore by an invocation of a data transfer from pageable memory space on the host to the device memory, the CUDA driver allocates first a temporary page-locked, or "pinned" host array, then it copies the data to the pinned array, and then it can transfer the data from the pinned array to the global device memory, as shown in fig. 10.

Transferring the batches to the GPU using pinned memory in our training model has dropped the time consumption per batch transfer to $0,17$ seconds. This means $(0,83 - 0,17) \times 6 = 3,96$ seconds for 6 batches.

Overlapping 2x Another technique to keep the GPU fully occupied with kernel launches is to overlap memory transfer operations so they happen during a kernel launch to avoid making the kernel wait for the data transfer. Recent GPUs provide several streams which can work asynchronously without blocking each other, we use these streams to achieve overlapped memory transfer. If we want to perform a batch copy while training on another batch the batch size has to be the half of the memory designated for uploading the batches, we preserve at least one batch already uploaded and start a kernel and a copying operations at the same time and switch the passed address pointing to the start point of the

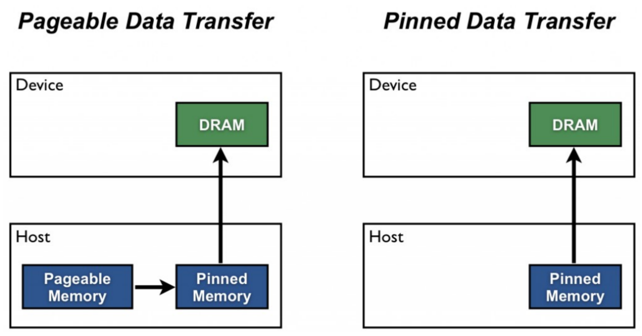


Fig. 10: pinned vs pageable memory transfer [12]

batch memory on the GPU. Moreover we have to make sure that the batches are not overwriting each other before the kernel that is training on one of them is completed, this way we guarantee the consistency of the training. According to [20] the default stream is always synchronized with all other streams, therefore we avoid using it throughout the whole training in order to maintain the asynchrony of the operations. [20] *Operations within the same stream are ordered (FIFO) and cannot overlap*, copying memory in the same direction can be carried out by only one stream at a time, `cudaStreamSynchronize(stream)` is used to wait for a stream till it finishes all queued jobs. In this approach we use 2 streams, one designated for copying data and the other one for launching kernels. The train function in pseudo code is:

```
cudaMemset(total_cost_on_device, 0);
counter := 0;
for batch in batches
{
  SwitchBatchLocation(location);
  ReadBatchIntoRAM(batch_file);
  cudaMemcpyAsync(batch, location, async_copy_stream);
  if (counter > 0)
  {
    cudaStreamSynchronize(async_kernel_stream);
  }
  if (counter == 0)
  {
    cudaStreamSynchronize(async_copy_stream);
  }
  train_kernel<<<BLOCKS, THREADS, 0, async_kernel_stream
      >>>(location);
  counter++;
}
```

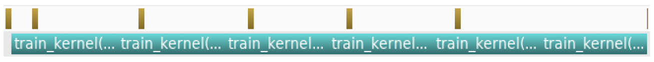


Fig. 11: overlap data transfer

A second model for the pipeline of transferring data and launching kernels concurrently is to use two streams as well where every stream copies data and run the kernel alternately. This way we can make sure without any additional if-statements that no batch is overwriting the other because the jobs in one stream are carried out in FIFO manner. The size of free memory on the GPU after allocating the model has to be at least 2× batchSize as well. Theoretically the processing flow has to be as shown in fig. 12.

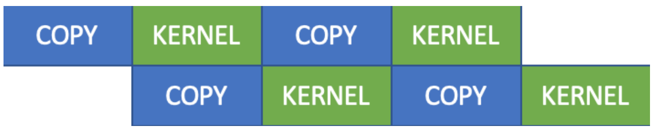


Fig. 12: theoretically

The runtime of the kernel takes longer than the data transfer operations when the GPU is sufficiently utilized. fig. 13 shows two profiled iterations, if an output of the cost has to be printed, then we would need a cudaSynchronize statement after completing an iteration and a cudaMemCpy to copy the cost from the device to the host since variables on the device can not be printed directly.

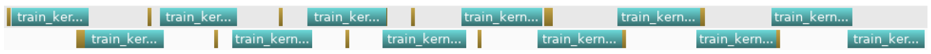


Fig. 13: profiled

kernel analysis In the kernel we implement the code that every thread has to run on his own inspired from the implementation of GloVe by Pennington et al., the cuda kernel receives pointers referring to the vectors and training data. Common cuda style is to use a thread for every single element, this style is called by Mark Harris *monolithic kernel* [13], however this is only applicable when the number of available threads larger than or equal the number of the elements to process. Therefore we use in our implementation the *grid-stride loop* as Harris M. advices in the his article ”*Grid-stride loops are a great way to make your CUDA kernels flexible, scalable, debuggable, and even portable*”.

```
for(int i = blockIdx.x * blockDim.x + threadIdx.x;
        i < n;
        i += blockDim.x * gridDim.x)
```

The number of elements is n, we notice here that if the number of threads exceeds the number of elements, it would lead to idle threads, and if the number of elements exceeds the number of threads, the threads will loop over the rest of the elements.
The first thing we do in the loop is to copy the variables we need throughout the computations to locally defined variable, thus we try to restrict the operations to be carried out on local memory, which is considered to be the fastest memory among global, shared and local memory of a graphic card. In order to train the vectors w_i and $\tilde{w_j}$ using a co-occurrence X_{ij} and the loss function defined by GloVe, first we calculate the gradients from the formula showed previously in section 3.1.

Second we update the parameters $\{w_i, \tilde{w_j}, b_i, \tilde{b_j}\}$ according to the rule of the used optimizer. For AdaGrad we accumulate the squared gradient for the next update and for Adam we accumulate additionally the gradients. After we calculate all needed gradients we have to update the parameters stored in the global memory, for this we either use atomic operations provided by the GPU `atomicAdd(addr, value)`, or we use our alternating approach introduced in section 4.2 to reduce collisions.

3.8 Parallel Training Issue:

For an efficient training, a good implementation would utilize as many threads as possible to use the most capabilities offered by the hardware. By minimizing the loss function of the GloVe model, an epoch means a complete one iteration over the cooccurrence matrix entries, where the word vector w_i and the context vector $\widetilde{w}_j$ are updated for every single cooccurrence X_{ij}. A trivial approach to parallelize this process is to distribute the cooccurrences among the available threads and let them work together. In the HOWILD paper [22] they assume that all the operations are atomic and let the threads overwrite each other. They have proven that by a sparse problem the collisions have an intangible effect on the result with a great win in performance.
They ran some numerical experiments on different machine learning tasks, the comparison was against a round-robin approach proposed in [30] notated as RR, and another model called AIG. This works very similar to HOGWILD except that it uses locks when updating the variables. Their experiments demonstrate that even this fine-grained locking induces undesirable slow-downs see fig. 14.
"All of the experiments were coded in C++ are run on an identical configuration: a dual Xeon X650 CPUs (6 cores each x 2 hyperthreading) machine with 24GB of RAM and a software RAID-0"
In their experiments they have used Xeon CPUs with maximum 32 threads.

Returning to our specific cause of collision when implementing a multithreaded GloVe training model, a collision could occur when two cooccurrences on the

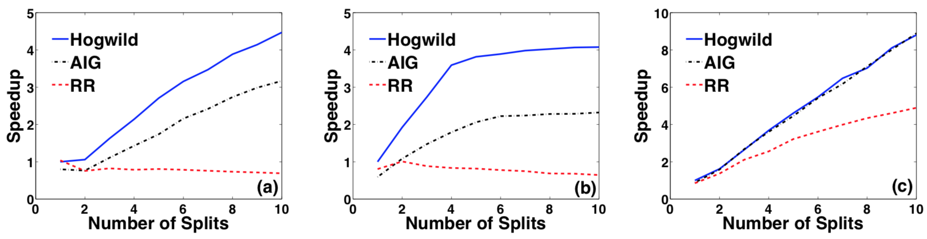


Fig. 14: Total CPU time versus number of threads for (a) RCV1, (b) Abdomen, and (c) DBLife.

same row, or on the same column of the matrix X, are being handled by two different threads at the same moment, either both of them will update the same context vector when the cooccurrences are on the same column, or both of them will update the same word vector when the cooccurrences are on the same row.

In this thesis we are trying to reach a highly scalable implementation that is able to run on a GPU with thousands of threads. First, atomic operations are more expensive on a device with this number of threads, second if we demonstrate the probability of collisions according to the number of running threads we will notice that it grows exponentially by increasing the number of threads. This can be estimated using a formula similar to the birthday paradox:

$$P(t, v) = 1 - \frac{v!}{v^t(v-t)!}$$

Where t denotes the number of threads and v the number of vectors in the model, this function computes the probability of a collision to occur in one iteration. The results are shown in fig. 15.

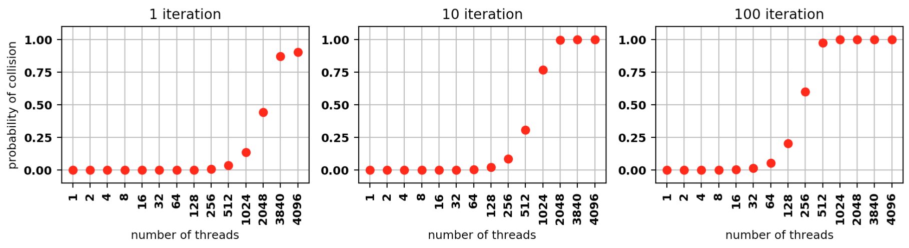


Fig. 15

The probability of collisions becomes very high (almost certain to happen at every iteration) when the number of threads is large. This means that optimizing the loss function will somewhen get very chaotic and at some point will stop approaching the global minimum.

3.9 GPU Atomics

CUDA provides atomic operations for consistent memory writing among threads, these operations have been improved on newer GPUs to compete the normal operations in performance. For this reason we have done a little experiment to compare the performance of the normal addition $+=$ operation and the `atomicAdd(addr, value)` operation provided by cuda. We use the Nvidia Titan Xp with compute capability 6.1 and we repeat the experiment for different datatypes. We define two kernels as the following:

```
__global__
void atomic_addition(real* a)
{
  atomicAdd(a,1);
}
__global__
void normal_addition(real* a)
{
  *a += 1;
}
```

And we run these two kernels with 1024 blocks $\times$ 1024 threads per block, we observe the following results:

```
type: int
Expected result: 1048576
...Atomic...
Value: 1048576
Time: 0.070528 milliseconds
...Normal ...
Value: 25
Time: 0.456 milliseconds
```

```
type: float
Expected result: 1048576
...Atomic...
Value: 1.04858e+06
Time: 0.151392 milliseconds
...Normal ...
Value: 25
Time: 0.652768 milliseconds
```

```
type: double
Expected result: 1048576
...Atomic...
Value: 1.04858e+06
Time: 3.9711 milliseconds
```

```
...Normal ...
Value: 26
Time: 0.48928 milliseconds
```

The results show that the atomic addition overperforms the normal addition when using integer and 4 byte float numbers, whereas normal addition is still faster when using 8 bytes float numbers (double)[1].

4 Conceptual Approaches

The first approach to overcome the collisions that would happen when two different threads are updating the same vector according to two different cooccurrences is to distribute the cooccurrences among the threads in such a way that would not lead to a collision between threads. There is probably several distribution manners that satisfy our constraint, therefore this is in fact an open question, for whom we will provide one possible good distribution.

4.1 Rotating Approach

As shown in Figure fig. 16, every yellow region is the portion of cooccurrences assigned to one thread and the four subfigures demonstrate four serial sub-iterations completing one epoch of training. n threads will divide a single epoch into n serial sub-iterations, in each iteration the threads are assigned new regions of X in a rotating style till all the entries are handled. Theoretically this approach would cause no collisions at all, because when a thread is responsible for a cooccurrence X_{ij} it would be the only candidate to update the word and context vectors w_i and w_j since no other thread is handling any cooccurrences lying on the same row i or column j. We notice that if the number of threads was increased the number of serial sub-iterations will be increased as well.

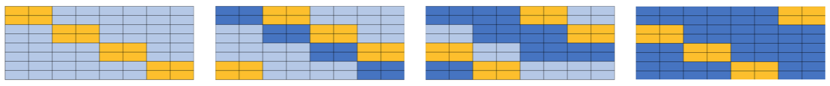


Fig. 16: Rotating Approach

4.2 Alternating-optimization Approach

In this approach we follow a straight distribution of the words over the threads with an additional modification of the updating privileges. We split a training

[1] Some further speculation on this unexpected behavior can be found from https://devtalk.nvidia.com/default/topic/524652/-slower-than-atomicadd-is-there-an-alternate-method-/

epoch into two sub-iterations, in the first iteration the threads are allowed only to update the word vectors, while in the second one they are only allowed to update the context vectors. At first glance it is confusing to understand why this would prevent collisions, but if we dig the cause of a collision we will find the following two causes:

- Two threads handling two cooccurrences X_{ik} and X_{jk} lying on the same column k. Both could collide when updating the context vector $\widetilde{w}_k$.
- Two threads handling two cooccurrences X_{ki} and X_{kj} lying on the same row k. Both could collide when updating the word vector w_k.

So organising the updating step of the threads in a way that prevent them from updating the same word/context vector at the same phase, taking in consideration the distribution manner, will guarantee no collisions. When we assign a thread number of words to handle the cooccurrences lying on their rows, it means in the first phase when only word vectors are being updated, all the cooccurrences lying on the same row, that could cause a collision in this phase, will fall into the responsibility of this one thread. The same applies for the second phase, all the cooccurrences lying on the same column will fall into the responsibility of one thread as shown in fig. 17.

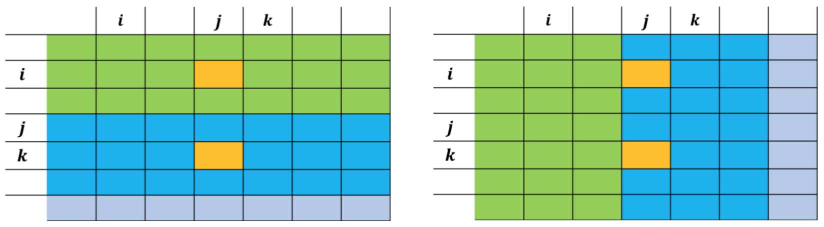


Fig. 17: Alternating-optimization Approach, Thread t_1 with green region t_2 with blue region, left matrix during phase one, right matrix during phase 2.

This approach will always split one epoch into two sub-iterations, no matter how many threads are invoked. The learning rate has to be calibrated in order to avoid biased behaviour during the training. One could split the learning step into two sub steps.

5 Time Evaluation

We have performed a time evaluation for the GloVe model developed based on this thesis meant to run on GPU and another time evaluation for the CPU version. The input used for both training is a set of 9 cooccurrence matrices extracted from the DBpedia data set using the tuned BCA algorithm approach with different parameters. The GPU we use for our experiments is the **NVIDIA TITAN Xp(compute capability 6.1)**, while the CPU is **Intel(R) Xeon(R)**

Gold 5122 CPU @ 3.60GHz with 32 threads. In fig. 18 we see that the GPU version has gained almost 1.71 speed up for one training iteration compared to the CPU version.

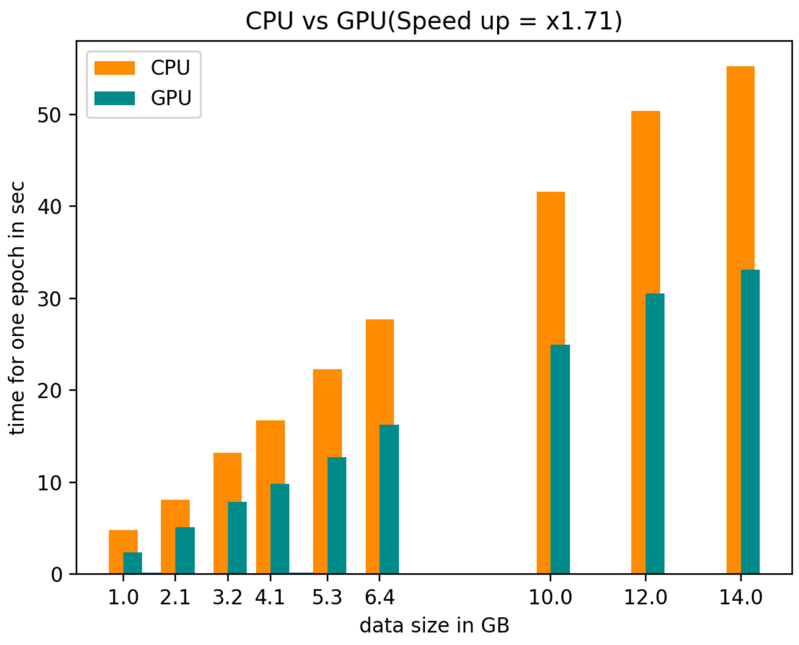


Fig. 18

6 Tuning hyperparameters

The main use case of the acceleration in this thesis is to run as many different experiments as possible in order to develop an understanding of how every parameter of the Graph Embedding process would affect the out-coming embeddings and eventually determine the best practice parameters for a various amount of scenarios. As we have seen in previous sections there are plenty a lot of parameters summarized in fig. 19

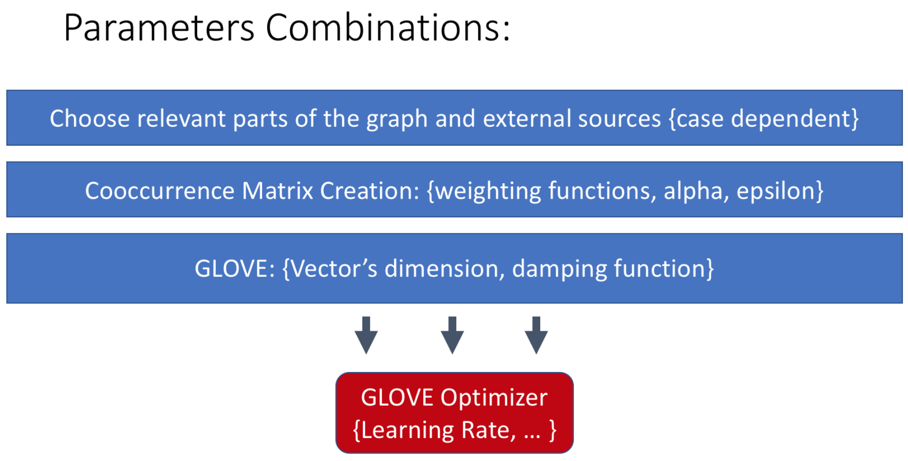

Parameters Combinations:
Choose relevant parts of the graph and external sources {case dependent}
Cooccurrence Matrix Creation: {weighting functions, alpha, epsilon}
GLOVE: {Vector's dimension, damping function}
GLOVE Optimizer
{Learning Rate, … }

Fig. 19

In Graph Embedding we encounter three levels of dependant input parameters. The first dependency is the relevant parts or structure of the graph that has to be included when extracting the cooccurrence matrix. Choosing relevant parts of the graph is mainly dependent on the study case (i.e. different approaches for including the attributes). Second level of dependencies is the level where the parameters for the extraction of cooccurrence matrix are given. For this level there is a huge amount of different parameters combinations that can be experimented, hence one could settle on some best practices for choosing these. The third layer of parameter set contains the vector size and the damping function. In our case we will try to tune the second layer (weighting function, alpha, epsilon) and include some plots in the evaluations that give an approximate description of the affect of each one. First we define the intervals from which the values of the parameters are tested:

- $\alpha \in \{0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95\}$
- $\epsilon \in \{10^i \text{ for } i \in \{-6, -5, -4, -3\}\}$
- Weighting function (forward, reverse): $|\text{Weighers}| \times |\text{Weighers}| = 12 \times 12$

6.1 Grid Search

Grid Search is the first intuitive concept that could serve us finding the suitable hyperparameters for our model, in which we set up the complete parameter combinations set and train the model on each combination individually, after all we compare the evaluations of the resulted trained vectors for every parameters combination. The evaluation has to be based on a different data set from that which was used for the training. The advantage of this method that it gives a

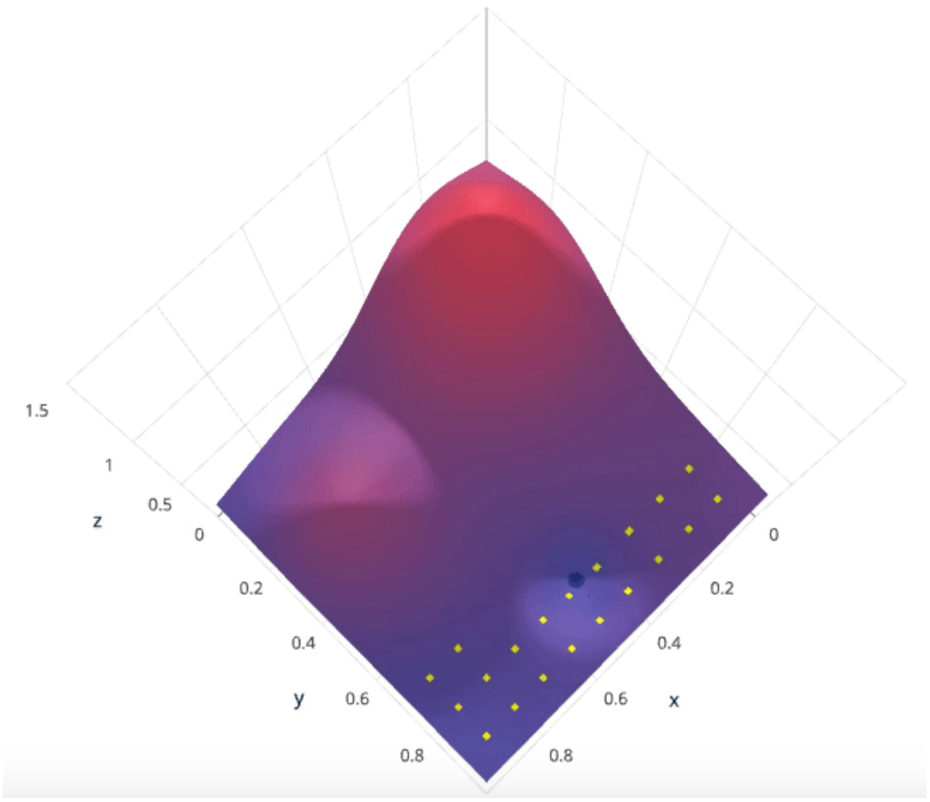


Fig. 20: Example of grid search from [15]

guaranteed result, however the inefficiency by this method is noticeable, since we would have to do:

$$|\alpha| \times |\epsilon| \times |\text{Weighers}| \times |\text{Weighers}| \times |\text{normalized}| \times |\text{only-entities}|$$
$$= 12 \times 6 \times 12 \times 12 \times 2 \times 2 = 41472$$

different walks, trainings and evaluations.

It is infeasible to search for the optimal combination having this number of different settings. Therefore we try another approach.

6.2 Random Search

In the random search tuning method, one provides a distribution for every parameter, from which a sample will be chosen, we create as many random sampled parameter combinations as our experiments can cover, then we run the experiments and try to have an intuition about the effect of every single parameter.

In fig. 20 and fig. 21 an example for finding the optimal hyperparameters for a problem set. We notice that iterating in a grid manner to find the global minima is pretty exhaustive for complex problem sets, whereas the random search is more applicable with some accuracy tradeoff. One could combine random with grid search, in which a convenient set of settings will be chosen based on the random search and a further investigation can be performed using this the chosen set and tuning every parameter on his own while fixing the others.

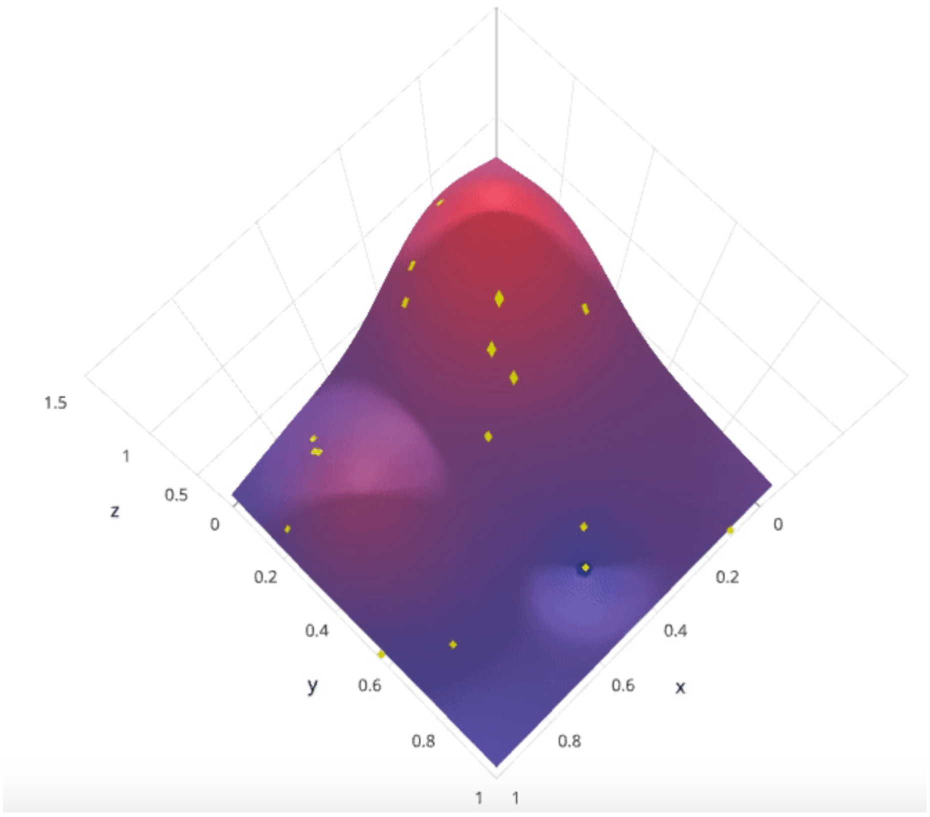


Fig. 21: Example of random search from [15]

7 Experiments

In our investigation for the optimal hyperparameters we performed 105 experiments with randomly chosen values for the walkś parameters, every experiment consists of three stages:

- Graph walks using BCA applied on all data of DBpedia 2016.
- Training using our GPU version of GloVe with 250 iterations.
- Evaluation of vectors regarding Classification and Regression tasks.

We evaluate the trained vectors with the *Graph Embedding Evaluation Framework* [9] developed mostly by Maria Angela Pellegrino and Martina Garofalo. The **Datasets** [26] used for the evaluation as golden sets are gathered by official observations linked to DBpedia. We use the following collections:

- Cities
- Metacritic movies
- Metacritic albums
- Forbes
- AAUP (only salary information)

So the embeddings for the whole DBpedia graph are computed with:

- vector size = 50×2 (50 for word vector and 50 for context vector)
- learning_rate = 0.01
- $X_{max} = 0.6$
- original damping function of GloVe with alpha = 0.75

and then the vectors are piped into the evaluation process which repeat every evaluation 10 times delivering twelve different measurements for each data collection:

- 10×Classification C45 accuracy
- 10×Classification KNN K=3 accuracy
- 10×Classification NB accuracy
- 10×Classification SVM C=0.001 accuracy
- 10×Classification SVM C=0.01 accuracy
- 10×Classification SVM C=0.1 accuracy
- 10×Classification SVM C=1.0 accuracy
- 10×Classification SVM C=10.0 accuracy
- 10×Classification SVM C=100 accuracy
- 10×Classification SVM C=1000 accuracy
- 10×Regression KNN K=3 root mean squared error
- 10×Regression LR root mean squared error

We take the average among the ten repeated evaluations for each experiment (hyperparameter combination), then we take the average result of each measurement among the five data collections. In the end we get for every hyperparameter combination twelve averaged measurements. For each measurement we rank all the 105 done experiments then take the average rank of each. For the defined values of each parameter of the walks (α,ϵ, Forward Weigher, Reverse Weigher, Normalizing, Only_Entities) we rank the value according to what ranks it appeared in, then we plot the values vs rank. Lower ranks are better.

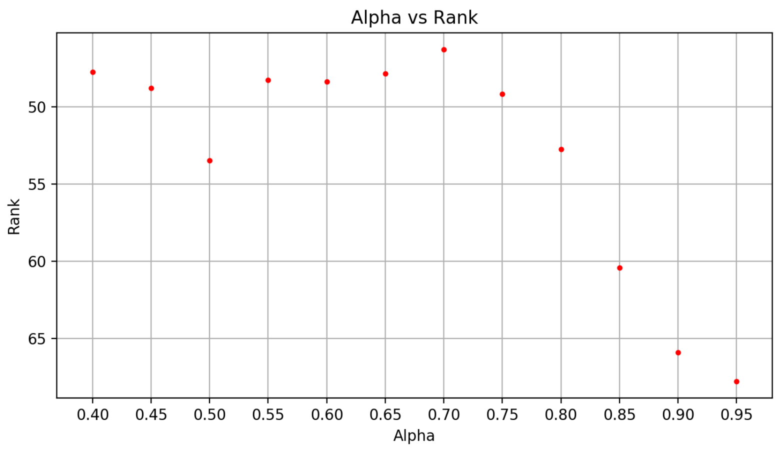


Fig. 22: α Rankings

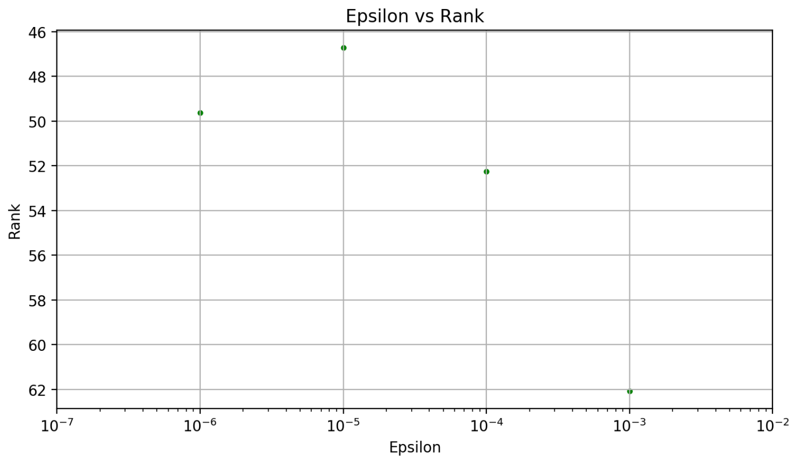


Fig. 23: ϵ Rankings

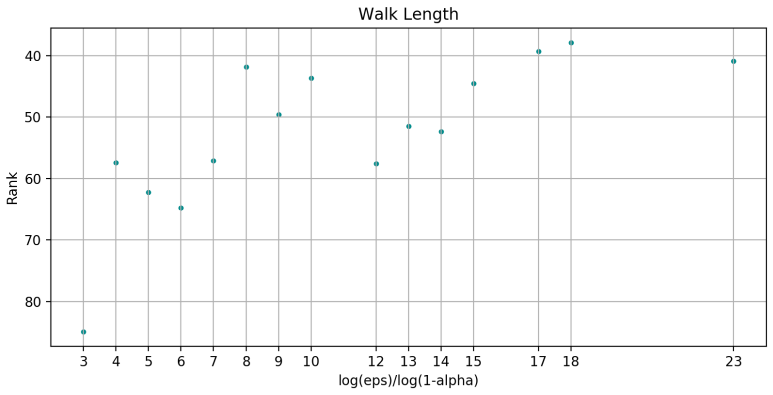


Fig. 24: Walk Length Rankings

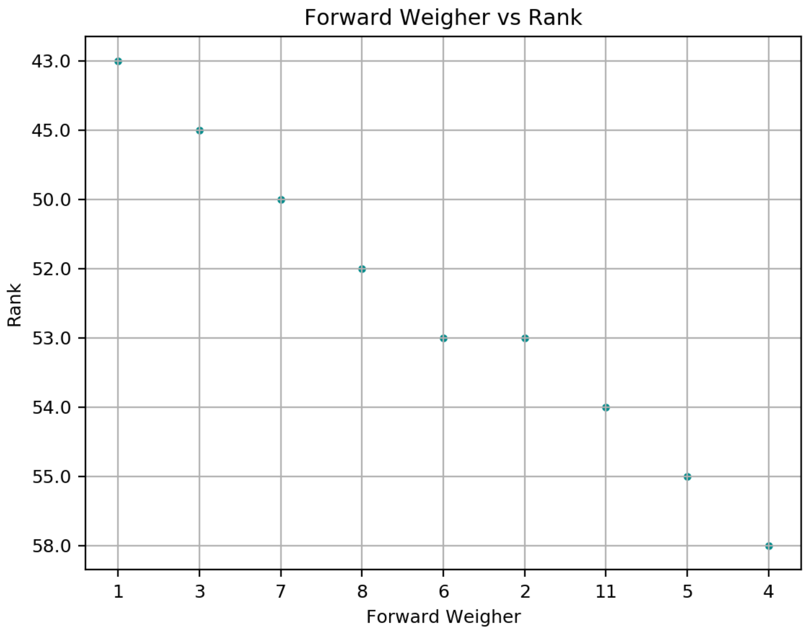


Fig. 25: Forward Weigher Rankings

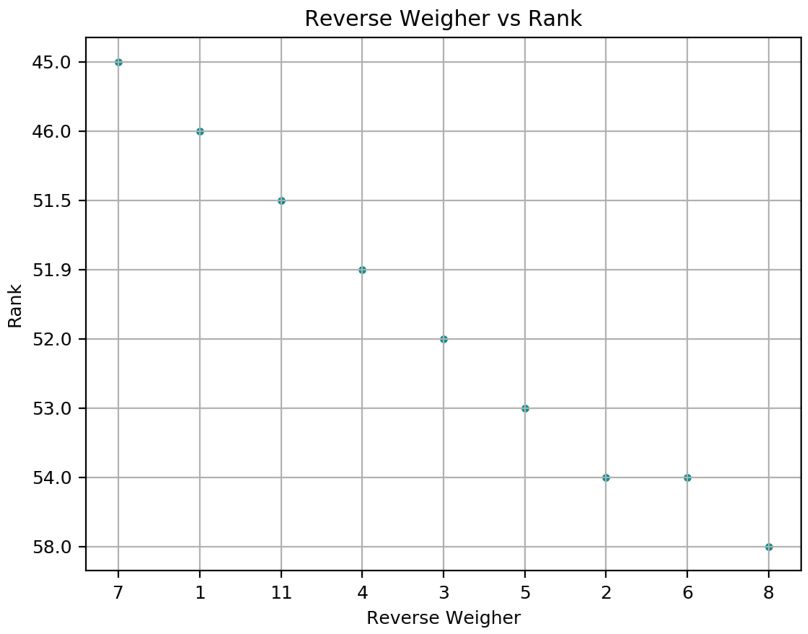


Fig. 26: Reverse Weigher Rankings

8 Conclusion

We have listed in this thesis core explanations of state of the art graph embedding approaches, that have drawn a big attention in the data analysis field of science. Some of the approaches deal directly with the graph, transferring its instances into embeddings based on the triplets they appear in. Other approaches deal with extracted statistical information about significant parts of the graph. We discussed at length how the KGlove embedding model works based on creating vectors for the instances, whose inner product gives the log probability of their relatedness. Moreover we summarized the main steps for developing the training model including the details in implementing Adagrad and Adam optimizers. Throughout the implementation we tried to propose convenient solutions for problems related to generating pseudo random numbers on the GPU using the threads ids with a light hash function and a seed random number generator, and dividing the training data into batches according to the required memory of the model. For an efficient CUDA program we listed some best practices summarized in using pinned memory (non-pageable) on the RAM for any data that has to be transferred between GPU and CPU, avoid using the default stream for memory copying and kernel launching, which is always synchronous with GPU operations and try to use local and shared memory for the threads' code, since accessing global memory costs relative high latency. We discussed the parallelization of the GloVe training model and its issues regarding collisions and memory efficiency, for the collision problem we introduced two theoretical approaches, rotating approach and alternating approach, which organize the updating process of the threads in a pattern that reduces the probability of a collision. In order to avoid the latencies caused by memory transfers we implemented two models of concurrent memory transfer and kernel launches. We compared atomic and normal addition operations on our experimental device concluding that atomics overperform normal additions when using any datatype except for 8 bytes floating point numbers(double). Our final implementation gave an average speed up of $\times 1.71$ compared with the original implementation for cpu. We used the accelerated model to investigate the effect of the hyperparameters used to extract a co-occurrence matrix from a given graph based on a scalable Bookmark-Coloring algorithm. Based on our investigations after a random search among evaluations of 105 different experiments, the correlation of α and ϵ influences not only the number of hops reached in every walk, but also the decaying function that plays a role by determining the importance of the nodes. The experiments show good results for $\alpha = 0.7$ and $\epsilon = 10^{-5}$ and for a Forward weigher: "Uniform Weigher" and Reverse weigher: "Inverse Object Frequency".

9 Future Work

Deeper investigation in tuning the hyperparameter should be done, since our resources during the thesis work phase allowed us to perform a relatively small amount of random experiments, therefore a larger scale experiments using our parallel training model can be done to cover a larger space of the parameters. Further experiments regarding other levels of parameters like vector_length, Adam optimizer, different damping functions and X_{max} values are still needed to complete the whole picture. Implementing the rotating and alternating approaches is a future piece of work that requires a deeper thoughts on mapping the co-occurrences to their responsible threads in a more efficient way than our existing implementation.

References

1. Berkhin, P.: Bookmark-coloring algorithm for personalized pagerank computing. Internet Mathematics 3(1), 41–62 (2006)
2. Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., Yakhnenko, O.: Translating embeddings for modeling multi-relational data. In: Burges, C.J.C., Bottou, L., Welling, M., Ghahramani, Z., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems 26, pp. 2787–2795. Curran Associates, Inc. (2013), http://papers.nips.cc/paper/5071-translating-embeddings-for-modeling-multi-relational-data.pdf
3. Brandes, U., Delling, D., Gaertler, M., Görke, R., Hoefer, M., Nikoloski, Z., Wagner, D.: On modularity clustering (2008)
4. Cochez, M., Ristoski, P., Ponzetto, S.P., Paulheim, H.: Biased graph walks for rdf graph embeddings. In: Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics. pp. 21:1–21:12. WIMS '17, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3102254.3102279
5. Cochez, M., Ristoski, P., Ponzetto, S.P., Paulheim, H.: Global RDF vector space embeddings. In: d'Amato, C., Fernandez, M., et al. (eds.) The Semantic Web – ISWC 2017: 16th International Semantic Web Conference, Vienna, Austria, October 21–25, 2017, Proceedings, Part I, pp. 190–207. Springer International Publishing, Cham (2017), http://users.jyu.fi/~miselico/papers/GlobalRDFEmbedding.pdf
6. Cohen, E.: node2vec: Embeddings for graph data (2018), https://towardsdatascience.com/node2vec-embeddings-for-graph-data-32a866340fef
7. DIMITROV, K.: Cyber platforms for adaptive cyber defence in industry 4.0 and logistics systems based on intelligent fault diagnosis. Cyber Defence in Industry 4.0 Systems and Related Logistics and IT Infrastructures 51, 38 (2018)
8. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. Tech. Rep. UCB/EECS-2010-24, EECS Department, University of California, Berkeley (Mar 2010), http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-24.html
9. Garofalo, M., Pellegrino, M.A.: Graph embedding evaluation framework (2018), https://datalab.rwth-aachen.de/embedding/evaluation/
10. Garofalo, M., Pellegrino, M.A., Altabba, A., Cochez, M.: Leveraging knowledge graph embedding techniques for industry 4.0 use cases (2018)
11. Grover, A., Leskovec, J.: node2vec: Scalable feature learning for networks. In: Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 855–864. ACM (2016)
12. Harris, M.: How to optimize data transfers in cuda c/c++, https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc
13. Harris, M.: Cuda pro tip: Write flexible kernels with grid-stride loops (2013), https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/
14. Jia, H., Saule, E.: Graph embedding for citation recommendation. arXiv preprint arXiv:1812.03835 (2018)
15. Johnson, A.: Common problems in hyperparameter optimization (2017), https://sigopt.com/blog/common-problems-in-hyperparameter-optimization
16. Kaspar, R., Horst, B.: Graph classification and clustering based on vector space embedding, vol. 77. World Scientific (2010)

17. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. CoRR abs/1412.6980 (2014), http://arxiv.org/abs/1412.6980
18. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
19. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. CoRR abs/1609.02907 (2016), http://arxiv.org/abs/1609.02907
20. Luitjens, J.: Cuda streams best practices and common pitfalls, http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf
21. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Advances in neural information processing systems. pp. 3111–3119 (2013)
22. Niu, F. and Recht, B. and Re, C. and Wright, S. J.: HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. ArXiv e-prints (jun 2011)
23. NVIDIA: Doxygen for nvidia cuda library, https://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/html/annotated.html
24. Pennington, J., Socher, R., Manning, C.D.: Glove: Global vectors for word representation. In: Empirical Methods in Natural Language Processing (EMNLP). pp. 1532–1543 (2014), http://www.aclweb.org/anthology/D14-1162
25. Reed, N.: Quick and easy gpu random numbers in d3d11 (2013), http://www.reedbeta.com/blog/quick-and-easy-gpu-random-numbers-in-d3d11/
26. Ristoski, P., de Vries, G.K.D., Paulheim, H.: A collection of benchmark datasets for systematic evaluations of machine learning on the semantic web. In: International Semantic Web Conference. pp. 186–194. Springer (2016)
27. Schlichtkrull, M., Kipf, T.N., Bloem, P., van den Berg, R., Titov, I., Welling, M.: Modeling relational data with graph convolutional networks. In: European Semantic Web Conference. pp. 593–607. Springer (2018)
28. Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., Mei, Q.: Line: Large-scale information network embedding. In: Proceedings of the 24th International Conference on World Wide Web. pp. 1067–1077. International World Wide Web Conferences Steering Committee (2015)
29. Xiao, H., Huang, M., Zhu, X.: Transg: A generative model for knowledge graph embedding. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). vol. 1, pp. 2316–2325 (2016)
30. Zinkevich, M., Langford, J., Smola, A.J.: Slow learners are fast. In: Bengio, Y., Schuurmans, D., Lafferty, J.D., Williams, C.K.I., Culotta, A. (eds.) Advances in Neural Information Processing Systems 22, pp. 2331–2339. Curran Associates, Inc. (2009), http://papers.nips.cc/paper/3888-slow-learners-are-fast.pdf