LibKGE
A knowledge graph embedding library for reproducible research

Samuel Broscheit, Daniel Ruffinelli, Adrian Kochsiek, Patrick Betz, Rainer Gemulla
Data and Web Science Group
University of Mannheim, Germany
`broscheit, daniel, adrian@informatik.uni-mannheim.de,`
`pbetz@mail.uni-mannheim.de,`
`rgemulla@uni-mannheim.de`

Abstract

LIBKGE[1] is an open-source PyTorch-based library for training, hyperparameter optimization, and evaluation of knowledge graph embedding models for link prediction. The key goals of LIBKGE are to enable *reproducible research*, to provide a framework for comprehensive experimental studies, and to facilitate analyzing the contributions of *individual components* of training methods, model architectures, and evaluation methods. LIBKGE is highly configurable and every experiment can be fully reproduced with a *single configuration file*. Individual components are decoupled to the extent possible so that they can be mixed and matched with each other. Implementations in LIBKGE aim to be as efficient as possible without leaving the scope of Python/Numpy/PyTorch. A comprehensive logging mechanism and tooling facilitates in-depth analysis. LIBKGE provides implementations of common knowledge graph embedding models and training methods, and new ones can be easily added. A comparative study (Ruffinelli et al., 2020) showed that LIBKGE reaches competitive to state-of-the-art performance for many models with a modest amount of automatic hyperparameter tuning.

1 Introduction

Knowledge graphs (KG) (Hayes-Roth, 1983) encode real-world facts as structured data. A knowledge graph can be represented as a set of (subject, relation, object)-triples, where the subject and object entities correspond to vertices, and relations to labeled edges in a graph.

KG embedding (KGE) models represent the KG's entities and relations as dense vectors, termed embeddings. KGE models compute a score based on these embeddings and are trained with the objective of predicting high scores for true triples and low scores for false triples. Link prediction is the task of predicting edges missing in the KG (Nickel et al., 2015). Some uses of KGE models are: enhancing the knowledge representation in language models (Peters et al., 2019), drug discovery in biomedical KGs (Mohamed et al., 2019), as part of recommender systems (Wang et al., 2017), or for visual relationship detection (Baier et al., 2017).

KGE models for link prediction have seen a heightened interest in recent years. Many components of the KGE pipeline—i.e., KGE models, training methods, evaluation techniques, and hyperparameter optimization—have been studied in the literature, as well as the whole pipeline itself (Nickel et al., 2016; Wang et al., 2017; Ali et al., 2020). Ruffinelli et al. (2020) argued that it is difficult to reach a conclusion about the impact of each component based on the original publications. For example, multiple components may have been changed simultaneously without performing an ablation study, baselines may not have been trained with state-of-the-art methods, or the hyperparameter space may not have been sufficiently explored.

LIBKGE is an open-source KGE library for reproducible research. It aims to facilitate meaningful experimental comparisons of all components of the KGE pipeline. To this end, LIBKGE is faithful to the following principles:

Modularization and extensibility. LIBKGE is cleanly modularized. Individual components can be mixed and matched with each other, and new components can be easily added.

Configurability and reproducibility. In LIBKGE an experiment is entirely defined by a *single* configuration file with well-documented configuration options for *every* component. When an experiment is started, its current configuration is stored alongside the model to enable reproducibility and analysis.

[1] `https://github.com/uma-pi1/kge`

Proceedings of the 2020 EMNLP (Systems Demonstrations), pages 165–174
November 16-20, 2020. ©2020 Association for Computational Linguistics

Profiling and analysis. LIBKGE performs *extensive logging* during experiments and monitors performance metrics such as runtime, memory usage, training loss, and evaluation metrics. Additionally, specific monitoring of any part of the KGE pipeline can be added via a hook system. The logging is done in both human-readable form and in a machine-readable format.

Ease of use. LIBKGE is designed to support the workflow of researchers by convenient tooling and easy usage with single line commands. Each training job or hyperparameter search job can be interrupted and resumed at any time. For *tuning of hyperparameters*, LIBKGE supports grid search, quasi-random search and Bayesian Optimization. All implementations stay in the realm of Python/PyTorch/Numpy and aim to be as efficient as possible.

LIBKGE supports the needs of researchers who want to investigate new components or improvements of the KGE pipeline. The strengths of LIBKGE enabled a comprehensive study that provided new insights about training KGE models (Ruffinelli et al., 2020). For an overview about usage, pretrained models, and detailed documentation, please refer to LIBKGE's project page. In this paper, we discuss the key principles of LIBKGE.

2 Modularization and extensibility

LIBKGE is highly modularized, which allows to mix and match training methods, models, and evaluation methods (see Figure 1). The modularization allows for simple and clean ways to extend the framework with new features that will be available for every model.

For example, LIBKGE decouples the *RelationalScorer* (the KGE scoring function) and *KgeEmbedder* (the way embeddings are obtained) as depicted in Figure 1. In other frameworks, the embedder function is hardcoded to the equivalent of LIBKGE's *LookupEmbedder*, in which embeddings are explicitly stored for each entity. Due to LIBKGE's decoupling, the embedder type can be freely specified independently of the scoring function, which enables users to train a KGE model with other types of embedders. For example, the embedding function could be an encoder that computes an entity or relation embedding from textual descriptions or pixels of an image (Pezeshkpour et al., 2018; Broscheit et al., 2020, inter alia).

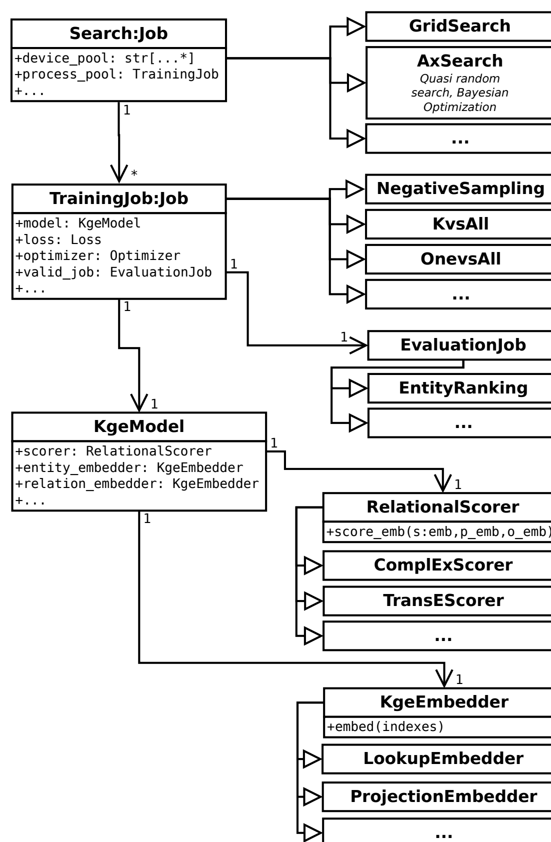

Figure 1: A brief overview of LIBKGE's architecture.

3 Configurability and reproducibility

Reproducibility is important, which means that configuration is important. To enable reproducibility, it is key that the entire configuration of each experiment be persistently stored and accessible. While this sounds almost obvious, the crux is how this can be achieved. Typically, source code can and will change. Therefore, to make an experiment in a certain setting reproducible, the configuration for an experiment has to be decoupled from the code as much as possible.

In LIBKGE *all* settings are always retrieved from a configuration object that is initialized from configuration files and is used by all components of the pipeline. This leads to comprehensive configuration files that fully *document* an experiment and make it reproducible as well.

To make this comprehensive configurability feasible—while also remaining modular—LIBKGE includes a lightweight `import` functionality for configuration files. In Figure 2, we show an (almost) minimal configuration for an experiment for training a *ComplEx* KGE model (Trouillon et al., 2016). The main configuration file

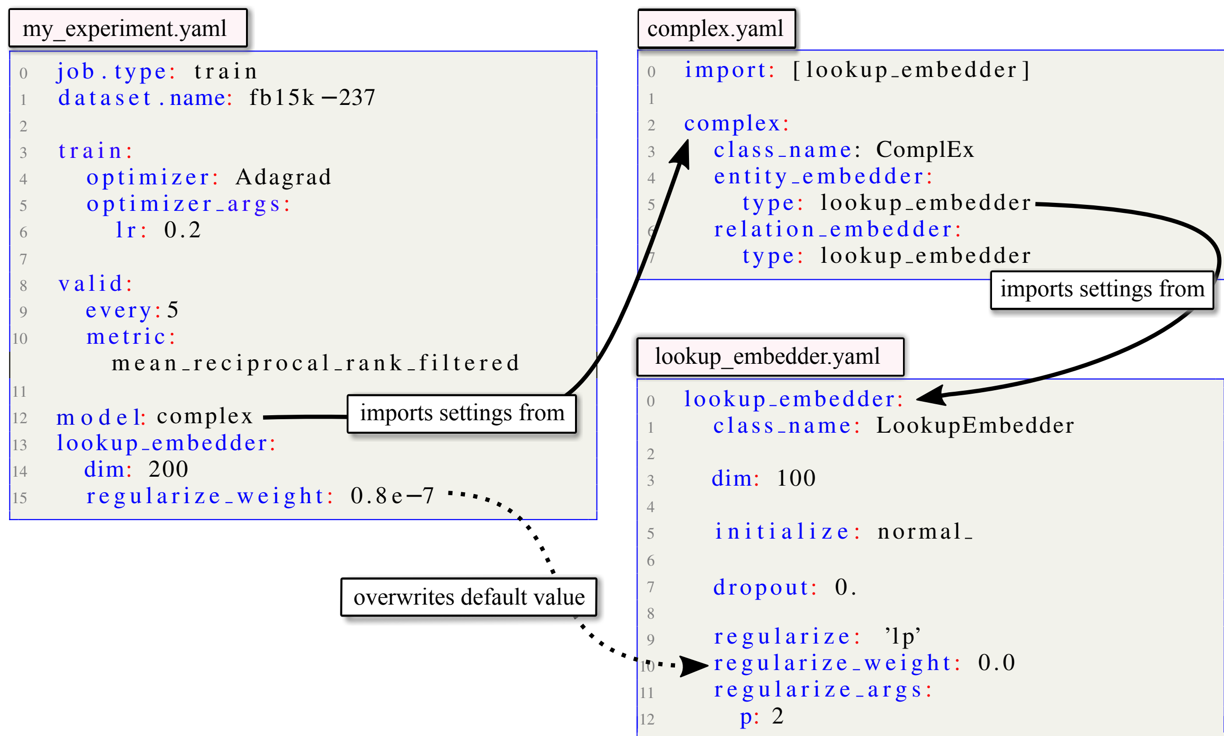

Figure 2: A minimal configuration `my_experiment.yaml` that defines 10 out of ≈ 100 configurable settings. *All* settings from the main configuration file `my_experiment.yaml` and from the imported configurations are merged and stored in one combined file. No default settings are defined in the code.

`my_experiment.yaml` in Figure 2 will automatically import the model-specific configuration `complex.yaml`, which in turn imports the configuration `lookup_embedder.yaml`. The latter defines the default configurations of the *LookupEmbedder* for entities and relations, which associates every entity and relation identifier with its respective embedding. All configurations are merged into a single configuration object. During merging, the settings in the main configuration file always have precedence over the settings from imported files. The resulting single configuration will be automatically saved in the experiment directory along with the checkpoints and the log files.

As an example of how configurability also helps modularization, we come back to the example of switching the *LookupEmbedder* with an encoder that computes entity embeddings from string tokens. For this purpose, one may implement a *TokenPoolEmbedder*. The simple changes to the configuration that uses the new embedder type are demonstrated in Figure 3 (see line 12).

It is worth noting that while the default settings in LIBKGE's main configuration file reflect the currently known best practices, LIBKGE also includes—and makes configurable—some settings that might not be considered *best* practice, e.g., different tie breaking schemes for ranking evaluations (Sun et al., 2020). Therefore, with regards to configurability, the goal is not only that the frame-

```
0  token_pool_embedder:
1    class_name: TokenPoolEmbedder
2    dim: 100
```

```
0  import: [token_pool_embedder]
1  job.type: train
2  dataset.name: fb15k-237
3
4  train:
5    optimizer: Adagrad
6    optimizer_args.lr: 0.2
7
8  model: complex
9  complex:
10   class_name: ComplEx
11   entity_embedder:
12     type: token_pool_embedder
13   relation_embedder:
14     type: lookup_embedder
```

Figure 3: Example of using a token-based embedder.

```
0  job.type: search
1  search:
2    type: ax
3    device_pool: [cuda:0, cuda:1]
4    num_workers: 4
5
6  dataset.name: wnrr
7
8  model: complex
9
10 train:
11   optimizer: Adagrad
12   type: negative_sampling
13
14 valid.metric:
       mean_reciprocal_rank_filtered
15
16 ax_search:
17   num_trials: 30
18   # remaining trials after random
19   # search are Bayesian Optimization
20   num_sobol_trials: 10
21   parameters:
22     - name: train.batch_size
23       type: choice
24       values: [256, 512, 1024]
25       is_ordered: True
26     - name: train.optimizer_args.lr
27       type: range
28       bounds: [0.001, 1.0]
29       log_scale: True
30     - name: negative_sampling.num.s
31       type: range
32       bounds: [16, 8192]
33     - name: negative_sampling.num.o
34       type: range
35       bounds: [16, 8192]
```

Figure 4: An example for a hyperparameter optimization job. This configurations first runs 10 trials of a quasi-random search followed by 10 trials of Bayesian Optimization (see `ax_search.num_trials` and `ax_search.num_sobol_trials`). By setting the keys `search.device_pool` and `search.num_workers` in lines 3 and 4 the execution of the trials is parallelized to run 4 parallel trials distributed over two GPU devices.

work reflects best practices, but also reflects popular practices that might influence ongoing research.

4 Hyperparameter optimization

Hyperparameter optimization is crucial in empirically investigating the impact of individual components of the KGE pipeline. LibKGE offers manual search, grid search, random search, and Bayesian Optimization; the latter two provided by the hyperparameter optimization framework Ax.[2] In this context, LibKGE further benefits from its configurability because everything can be treated as a hyperparameter, even the choice of model, score function, or embedder. The example in Figure 4 shows a simple hyperparameter search with an initial quasi-random search, and a subsequent Bayesian Optimization phase over the learning rate, batch size and negative samples for the *ComplEx* model. The trials during the quasi-random search are independent, which can be exploited by parallelizing their runs over multiple devices. In this way, a comprehensive search over a large space of hyperparameters can be sped up significantly (also shown in the example; for more details, please refer to the documentation).

[2] https://ax.dev/

5 Profiling and metadata analysis

LibKGE provides extensive options for profiling, debugging, and analyzing the KGE pipeline. While most frameworks print the current training loss and some frameworks also record the validation metrics, LibKGE aims to make every moving part of the pipeline observable. Per default, LibKGE records during training things such as runtimes, training loss and penalties (e.g., from the entity and relation embedders), relevant meta data such as the PyTorch version and the current commit hash, and dependencies between various jobs. We show an example logging output during training one epoch in Appendix B. For more fine-grained logging, LibKGE also can log at the batch level.

During evaluation, the framework records many variations of the evaluation metrics, such as grouping relations by relation type, relation frequency, head or tail. Additionally, users can extract and add information by adding their custom function to one of multiple hooks that are executed before and after all relevant calls in the framework. In this way, users can interact with all components of the pipeline, without risking divergence from LibKGE's master branch.

Finally, LibKGE provides convenience methods to export (subsets of) the logged meta data into plain CSV files.

6 Comparison to other KGE Projects

In this section, we compare LibKGE to other open source software (OSS) that provides functionality around training and evaluating KGE models for link prediction. The assessments are a snaphot taken at the end of May 2020. All model-specific comparisons have been evaluated w.r.t. the *ComplEx* model, which is supported by all projects.

	KGE project	Implementation language(s)	Config-urable keys	Log-ging train/eval	Hyperparam. Optimization			Res-ume	Act-ive
					Grid	Rnd	BO		
Framework	LIBKGE (Ours)	PyTorch	91	17/414	x	x	x	x	668
	PyKeen	PyTorch	61	2/27		x	x		575
	Ampligraph	TF 1.x	20	0/8	x	x			286
	OpenKE	PyTorch/C++	19	1/30					22
	SK-libkge	TF 1.x	14	5/7	x				24
Large Scale	GraphVite	C++/PyTorch	34	2/5					14
	DGL-KE	PyTorch/MxNET	15	10/6				x	134
	PyTorch-BG	PyTorch	52	12/8				x	102
Paper Code	KBC	PyTorch	10	4/12					2
	Hyperb. KGE	TF 2.x/PyTorch	20	6/5					12/19
	ConvE	PyTorch	15	5/36				x	2
	RotatE	PyTorch	28	3/5					3

Table 1: Comparing LIBKGE and other OSS that provide functionality around training KGE models for link prediction. All assessments have been made at the end of May 2020. *Frameworks* denotes focus on fostering KGE research with modularization, extensibility and coverage of relevant models and training methods. *Large Scale* denotes focus on extremely large-scale graphs, with support of training in multi-node or multi-gpu mode, or both. *Paper Code* denotes projects published alongside a KGE model's publications. *Configurable keys* are the number of possible options for training a single *ComplEx* model, i.e. not counting options for hyperparameter search. *Logging* denotes the number of metadata keys that are logged per epoch for training and evaluation in a machine readable format for later analysis. *Hyperparameter optimization* shows if the project supports grid search, random search and Bayesian Optimization. *Resume* denotes the feature to resume hyperparameter search or training from checkpoints at any time. *Active* is the amount of commits to the master branch in the last 12 months.

In Table 1, we provide an overview of other KGE projects (full references in Appendix C) and compare them w.r.t. configurability and ease of use. We mainly included projects that could be considered as a basis for a researcher's experiments because they are active, functional, and cover at least a few of the most common models. All projects can be extended with models, losses, or training methods. Large-scale projects and paper code projects—in comparison to more holistic frameworks—typically have a more narrow scope, e.g., they often do not feature hyperparameter optimization. Large-scale projects are typically tailored towards parallelizing training methods and models.

The focus on configurability and reproducibility in LIBKGE is reflected by the large amount of configurable keys. For example, in contrast to other projects, LIBKGE does not tie the regularization weights of the entity and relation embedder to be the same. For entity ranking evaluation, only LIBKGE and PyKeen transparently implement different tie breaking schemes for equally ranked entities. This is important, because evaluation under different tie breaking schemes can result in differences of $\approx .40$ MRR in some models and can lead to misleading conclusions, as shown by Sun et al. (2020). OpenKE, for example, only supports the problematic tie breaking scheme named *TOP* by Sun et al. (2020). LIBKGE and PyKeen are the only frameworks that provide machine-readable logging. Only LIBKGE offers resuming from a checkpoint for training and hyperparameter search. LIBKGE, Ampligraph, and PyKeen are the most active projects in terms of amount of commits during the past 12 months.

Efficiency In Table 2, we show a comparison of KGE frameworks in terms of time for one full training epoch. The configuration setting was chosen such that it was supported by all frameworks, and also facilitates to demonstrate behaviour under varying load. We translate the configurations faithfully to each framework, ensuring that total number of embedding *parameters* per batch are the same for each framework. Most projects, including LIBKGE, can handle small numbers of negative samples efficiently, but LIBKGE seems to scale

Project	Parallel batch construction	64 random	64 pseudo negative	1024 random	1024 pseudo negative	16384 random	16384 pseudo negative
LibKGE (Ours)	x	9 s	13 s	14 s	19 s	74 s	113 s
PyKeen	x	10 s	-	64 s	-	930 s	-
Ampligraph		21 s	-	190 s	-	OOM	-
OpenKE	x	7 s	7 s	59 s	63 s	OOM	OOM
SK-libkge		99 s	-	1210 s	-	OOT	-
GraphVite (*)		54 s	-	58 s	-	82 s	-

(Column groups 64, 1024, 16384: Number of negative samples per triple)

Table 2: Runtime comparison between frameworks. The runtime is the time per epoch in seconds (averaged over 5 epochs executed on the same machine). The configuration is fixed to be similar for all frameworks (details in Appendix A). For negative samples, we show runtimes for *random*, i.e., sampling triples without checking if they are contained in the KG, and for *pseudo-negative*, which avoids sampling triples contained in the KG. The column *parallel batch construction* indicates whether the code in the training loop for generating the batches is parallelized; if yes, then we set the number of workers to 4. OOM stands for out-of-memory. OOT is short for out-of-time; we stopped the run when the first epoch did not finish within 30 minutes. (*) Graphvite is optimized for multi-gpu training with large batch sizes, therefore the chosen settings might not be optimal.

	MRR	HITS@10
LibKGE (Ours)	0.35	0.54
PyKeen	-	0.44
Ampligraph	0.32	0.50
OpenKE	-	0.43
GraphVite	0.27	0.45

Table 3: The *reported* best performances (on the project's homepage or the related publication as of May 2020) for *ComplEx* on FB15K-237 for each project. The performances have been obtained with different amount of effort for hyperparameter optimization and should not be compared directly. Reported ranking metrics: Mean Reciprocal Rank (MRR) and HITS@10.

better to higher numbers of negative samples. A large number of negative samples becomes important when large KGs with millions of entities are embedded. Although the runtimes are purely anecdotal and should be taken with a grain of salt, they do show that LibKGE can provide competitive runtime performance. Currently, LibKGE only supports single-node single-gpu training. It nevertheless fares well when compared to GraphVite, one of the large-scale frameworks that dispatches some routines into C/C++. LibKGE also has optimized versions of negative sampling for large graphs, which enables it to train *ComplEx* on Wikidata-5m (Wang et al., 2019), a large KG with 5M entities.

Predictive performance. In Table 3, we collected the reported performances for *ComplEx* on the dataset FB15K-237 (Toutanova and Chen, 2015). The numbers are not comparable due to different amount of effort to find a good configuration[3], but they reflect the performance that the framework authors achieved in their experiments. The results show that with LibKGE's architecture and hyperparameter optimization a state-of-the-art result can be achieved. For more results obtained with LibKGE and an in-depth analysis of the impact of hyperparameters on model performance we refer to the study by Ruffinelli et al. (2020).

7 Conclusions

In this work, we presented LibKGE, a configurable, modular, and efficient framework for reproducible research on knowledge graph embedding models. We briefly described the internal structure of the framework and how it facilitates LibKGE's goals. The framework is efficient and yields state-of-the-art performance. We hope that LibKGE is a helpful ingredient to gain new insights into knowledge graph embeddings, and that a lively community gathers around this project to improve and extend it further.

[3]We did not attempt to use our best configuration with other frameworks because they only partly support the settings, e.g., they do not offer dropout or independent regularization for entity and relation embeddings.

References

Mehdi Ali, Max Berrendorf, Charles Tapley Hoyt, Laurent Vermue, Mikhail Galkin, Sahand Sharifzadeh, Asja Fischer, Volker Tresp, and Jens Lehmann. 2020. Bringing light into the dark: A large-scale evaluation of knowledge graph embedding models under a unified framework. *arXiv preprint arXiv:2006.13365*.

Stephan Baier, Yunpu Ma, and Volker Tresp. 2017. Improving visual relationship detection using semantic modeling of scene descriptions. In *Proceedings of the 16th International Semantic Web Conference (ISWC)*.

Antoine Bordes, Nicolas Usunier, Alberto García-Durán, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pages 2787–2795.

Samuel Broscheit, Kiril Gashteovski, Yanjie Wang, and Rainer Gemulla. 2020. Can we predict new facts with open knowledge graph embeddings? A benchmark for open link prediction. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2296–2308, Online. Association for Computational Linguistics.

Ines Chami, Adva Wolf, Da-Cheng Juan, Frederic Sala, Sujith Ravi, and Christopher Ré. 2020. Low-dimensional hyperbolic knowledge graph embeddings. *Annual Meeting of the Association for Computational Linguistics*.

Luca Costabello, Sumit Pai, Chan Le Van, Rory McGrath, Nicholas McCarthy, and Pedro Tabacof. 2019. AmpliGraph: a library for representation learning on knowledge graphs.

Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. 2018. Convolutional 2d knowledge graph embeddings. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 1811–1818.

Xu Han, Shulin Cao, Xin Lv, Yankai Lin, Zhiyuan Liu, Maosong Sun, and Juanzi Li. 2018. OpenKE: An open toolkit for knowledge embedding. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Frederick Hayes-Roth. 1983. *Building expert systems*, volume 1 of *Advanced book program*. Addison-Wesley.

Timothée Lacroix, Nicolas Usunier, and Guillaume Obozinski. 2018. Canonical tensor decomposition for knowledge base completion. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 2869–2878.

Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. PyTorch-BigGraph: A Large-scale Graph Embedding System. In *Proceedings of the 2nd SysML Conference*, Palo Alto, CA, USA.

Sameh K. Mohamed, Aayah Nounu, and Vít Nováček. 2019. Drug target discovery using knowledge graph embeddings. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC 19, page 1118, New York, NY, USA. Association for Computing Machinery.

Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. 2015. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*.

Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. 2016. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33.

Matthew E. Peters, Mark Neumann, Robert Logan, Roy Schwartz, Vidur Joshi, Sameer Singh, and Noah A. Smith. 2019. Knowledge enhanced contextual word representations. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 43–54, Hong Kong, China. Association for Computational Linguistics.

Pouya Pezeshkpour, Liyan Chen, and Sameer Singh. 2018. Embedding multimodal relational data for knowledge base completion. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3208–3218, Brussels, Belgium. Association for Computational Linguistics.

Daniel Ruffinelli, Samuel Broscheit, and Rainer Gemulla. 2020. You CAN teach an old dog new tricks! on training knowledge graph embeddings. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*.

Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. 2019. RotatE: Knowledge graph embedding by relational rotation in complex space. In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*.

Zhiqing Sun, Shikhar Vashishth, Soumya Sanyal, Partha Talukdar, and Yiming Yang. 2020. A re-evaluation of knowledge graph completion methods. In *Proceedings of the 58th Annual Meeting of the*

Association for Computational Linguistics, pages 5516–5522, Online. Association for Computational Linguistics.

Kristina Toutanova and Danqi Chen. 2015. Observed versus latent features for knowledge base and text inference. In *Proceedings of the 3rd Workshop on Continuous Vector Space Models and their Compositionality*, pages 57–66, Beijing, China. Association for Computational Linguistics.

Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. 2016. Complex embeddings for simple link prediction. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 2071–2080.

Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. 2017. Knowledge graph embedding: A survey of approaches and applications. *IEEE Transactions on Knowledge and Data Engineering*.

Xiaozhi Wang, Tianyu Gao, Zhaocheng Zhu, Zhiyuan Liu, Juanzi Li, and Jian Tang. 2019. Kepler: A unified model for knowledge embedding and pre-trained language representation.

Da Zheng, Xiang Song, Chao Ma, Zeyuan Tan, Zi-Hao Ye, J. Dong, Hao Xiong, Zheng Zhang, and G. Karypis. 2020. Dgl-ke: Training knowledge graph embeddings at scale. *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*.

Zhaocheng Zhu, Shizhen Xu, Meng Qu, and Jian Tang. 2019. Graphvite: A high-performance cpu-gpu hybrid system for node embedding. In *The World Wide Web Conference*, pages 2494–2504. ACM.

A Runtime comparison experiment

The configuration was as follows: Dataset FB15K (Bordes et al., 2013), batch size 512, model *ComplEx*, effective parameter embedding size per entity/relation 128, optimizer Adagrad, negative sampling with negative log likelihood loss or sigmoid loss, no regularization. The hardware was a 8-core Intel Xeon E5-1630m v4.0, TitanXP GPU, dataset on SSD.

B Logging


```
0  {
1     "entry_id":84d75bf2-c3fe-4c6f-ac5e-001e1edb85de,
2     "event":"job_created",
3     "folder":/home/USER/kge/local/experiments/20200705-215353-toy-complex-train,
4     "git_head":7fad132,
5     "hostname":USER-Workstation,
6     "job":"eval",
7     "job_id":683d00bf-520d-4919-937e-d9b634c11d2e,
8     "parent_job_id":dc960211-9cbe-4ba1-ad62-7ffd41d2017e,
9     "timestamp":1593978837.304522,
10    "torch_version":1.5.0,
11    "username":"USER"
12 }{
13    "entry_id":418889f0-728b-486f-9977-48795f6ed5fa,
14    "event":"job_created",
15    "folder":/home/USER/kge/local/experiments/20200705-215353-toy-complex-train,
16    "git_head":7fad132,
17    "hostname":USER-Workstation,
18    "job":"train",
19    "job_id":dc960211-9cbe-4ba1-ad62-7ffd41d2017e,
20    "timestamp":1593978837.4033182,
21    "torch_version":1.5.0,
22    "username":"USER"
23 }{
24    "avg_cost":1.06542689547832,
25    "avg_loss":1.0650610147438764,
26    "avg_penalties":{
27       complex.entity_embedder.L2_penalty:0.00031927969330354246,
28       complex.relation_embedder.L2_penalty:4.6601041140093004e-05
29    },
30    "avg_penalty":0.0003658807344436354,
31    "backward_time":0.0765678882598877,
32    "batches":20,
33    "entry_id":4b07adfa-3e2b-42f4-a994-2b4f02e1b3f4,
34    "epoch":1,
35    "epoch_time":1.161754846572876,
36    "event":"epoch_completed",
37    "forward_time":0.7509596347808838,
38    "job":"train",
39    "job_id":dc960211-9cbe-4ba1-ad62-7ffd41d2017e,
40    "lr":[ 0.2 ],
41    "optimizer_time":0.015013933181762695,
42    "other_time":0.2690012454986572,
43    "prepare_time":0.05021214485168457,
44    "scope":"epoch",
45    "size":1949,
46    "split":"train",
47    "timestamp":1593978838.5940151,
48    "type":"KvsAll"
49 }
```


Figure 5: Example for training logging output for one epoch. Evaluation logging output is too verbose to add an example here. Please see https://github.com/uma-pi1/kge/blob/master/docs/examples/train_and_valid_trace_after_one_epoch.yaml for an example for the output after one epoch of training and evaluation.

C Related projects

KGE project	URL	Reference
LIBKGE (Ours)	https://github.com/uma-pi1/kge	Ruffinelli et al. (2020)
PyKeen	https://github.com/pykeen/pykeen	Ali et al. (2020)
Ampligraph	https://github.com/Accenture/AmpliGraph	Costabello et al. (2019)
OpenKE	https://github.com/thunlp/OpenKE	Han et al. (2018)
GraphVite	https://github.com/DeepGraphLearning/graphvite	Zhu et al. (2019)
DGL-KE	https://github.com/awslabs/dgl-ke	Zheng et al. (2020)
Pytorch-Biggraph	https://github.com/facebookresearch/PyTorch-BigGraph	Lerer et al. (2019)
SK-libkge	https://github.com/samehkamaleldin/libkge	
KBC	https://github.com/facebookresearch/kbc	Lacroix et al. (2018)
Hyperbolic KGE	https://github.com/tensorflow/neural-structured-learning/tree/master/research/kg_hyp_emb	Chami et al. (2020)
ConvE	https://github.com/TimDettmers/ConvE	Dettmers et al. (2018)
RotatE	https://github.com/DeepGraphLearning/KnowledgeGraphEmbedding	Sun et al. (2019)

Table 4: Overview of related work