open proceedings

Distributed Training of Knowledge Graph Embedding Models using Ray

Nasrullah Sheikh
IBM Research Almaden
San Jose, California, US
nasrullah.sheikh@ibm.com

Yaniv Gur
IBM Research Almaden
San Jose, California, US
guryaniv@us.ibm.com

Xiao Qin
IBM Research Almaden
San Jose, California, US
xiao.qin@ibm.com

Berthold Reinwald
IBM Research Almaden
San Jose, California, US
reinwald@us.ibm.com

ABSTRACT

Knowledge graphs are at the core of numerous consumer and enterprise applications where learned graph embeddings are used to derive insights for the users of these applications. Since knowledge graphs can be very large, the process of learning embeddings is time and resource intensive and needs to be done in a distributed manner to leverage compute resources of multiple machines. Therefore, these applications demand performance and scalability at the development and deployment stages, and require these models to be developed and deployed in frameworks that address these requirements. Ray[1] is an example of such a framework that offers both ease of development and deployment, and enables running tasks in a distributed manner using simple APIs. In this work, we use Ray to build an end-to-end system for data preprocessing and distributed training of graph neural network based knowledge graph embedding models. We apply our system to *link prediction* task, i.e. using knowledge graph embedding to discover links between nodes in graphs. We evaluate our system on a real-world industrial dataset and demonstrate significant speedups of both, distributed data preprocessing and distributed model training. Compared to non-distributed learning, we achieved a training speedup of 12× with 4 Ray workers without any deterioration in the evaluation metrics.

1 INTRODUCTION

Knowledge Graphs (KG) have become a disruptive component in enterprises by providing uniform and semantically rich access to vast and diverse data sources. The capability of "making sense" of large KGs is crucial to many tasks ranging from data integration, entity resolution, search, data extraction, data exchange, and business intelligence [6, 19]. Representation learning (RL) over KGs has emerged as one of the key enablers for many of these tasks.

The goal of RL is to embed a KG which is high-dimensional but very sparse into a low-dimensional space while preserving the characteristics of the original graph. KG embedding models such as TransE [3], DistMult [20], and ComplEx [18] process triplets. A triplet (s, r, o) represents a fact where s is the subject, o is the object and r is the relation between them. These methods treat each triplet in the knowledge graph independently, and they learn various structure-relational patterns such as *symmetric* and *inverse* relations between entities. Recent advances in knowledge graph representation learning have shown that graph neural network based knowledge graph embedding methods demonstrate superior performance over traditional embedding methods [14, 15]. State-of-the-art KG embedding methods [4, 14, 15, 17, 21] for link prediction generally follow an encoder-decoder architecture where the encoder aggregates the multi-hop context for generating node embeddings, while the decoder scores and learns the relations between the nodes.

Due to the growing size and computational complexity of KGs, distributed KG embedding training has recently attracted considerable attention in the research community. Existing frameworks [8, 16, 22] adopt distributed, data-parallel architectures to cope with high computational and large memory requirements. In these architectures, the KG is first partitioned, and then partitions are distributed across the compute nodes for training. These frameworks employ different methods for training, for example, [16] uses ring AllReduce communication architecture for gradient exchange. DGL-KE [22] and PBG [8] are designed for traditional KG embedding models such as TransE and DistMult which process the triplets independently. These models cannot be used for distributed training of GNN-based KG embedding models which require k-hop neighborhood information. One of the technical challenges in distributed training of GNN architectures is to effectively partition the graph into enough partitions such that each compute node has a relatively small amount of self-sufficient data to work with, and the workload is balanced across all the compute nodes to avoid idle time during SSGD.

Sheikh et al. [16] proposed a system for distributed training of GNN-based KG embedding models using a neighborhood expansion method to make partitions self-sufficient. The KG was first partitioned using an edge-partitioning method, and then the partitions were expanded using a neighborhood expansion process. The neighborhood expansion process included all the k-hop neighbors of all the nodes in a partition to avoid cross partition communication overhead during training at the cost of node and edge replication. The paper has also introduced constraint-based negative sampling for training. It was experimentally shown that drawing negative samples from within the partitions was effective, and it further reduced cross partition communication during training as it avoided transferring negative samples across the partitions.

The distributed training approach in [16] is based on *PyTorch distributed*[13] APIs. Using core level APIs of an underlying

[1] https://docs.ray.io/en/master/index.html

© 2022 Copyright held by the owner/author(s). Published in Proceedings of the 25th International Conference on Extending Database Technology (EDBT), 29th March-1st April, 2022, ISBN 978-3-89318-085-7 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

framework such as PyTorch is challenging for scaling and deployment. More recent distributed frameworks such as Ray [11, 12] simplify underlying details for scaling and deployment. Ray is a general-purpose cluster computing framework that provides universal, simple APIs that enable developers to use existing libraries and systems for distributed computing. The rich Ray framework provides APIs for training (Ray Train), tuning (Ray Tune), online serving (Ray Serve), and distributed data management (Modin) [2, 9, 10]. Ray Train [1] provides a simple to use API for distributed training of neural network models using deep learning frameworks such as PyTorch. Moreover, it provides composability which allows Ray Train to inter-operate with Ray Tune for hyperparameter learning.

The contributions of this short paper are summarized as follows:

- We introduce a state-of-the-art distributed KG embedding training framework [16] using Ray, and demonstrate performance results on an industrial scale KG.
- We provide and evaluate a distributed attribute preprocessing pipeline using Ray.
- We use *Ray Train* API for distributed training of the KG embedding model, and demonstrate the simplicity of the API while performing a complex task.

The rest of the paper is organized as follows: In Section 2 we present an overview of our industrial dataset, and Section 3 describes the architecture of our system. Section 4 provides the experimental setup and results, and we conclude in Section 5.

2 DATA OVERVIEW

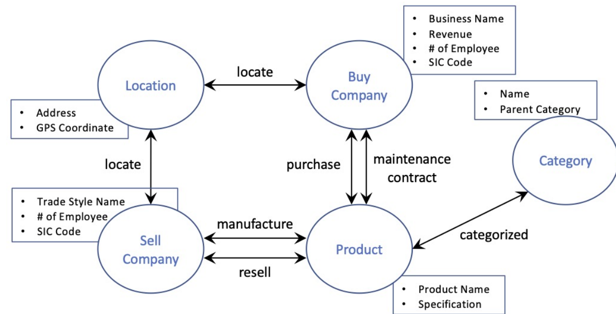


Figure 1: The schema of an enterprise KG with heterogeneous node types, edge types and node attributes.

Let $G = (V, E, A, R, \phi)$ denote a KG. $V = \{v_1, \cdots, v_n\}$ is a set of nodes in G. $A = \{a_1, \cdots, a_k\}$ defines the node attribute schema of G. Each node $v_i \in V$ is associated with a node type $\phi(v_i)$. Its corresponding attribute schema is a sub-schema of A denoted by $A_{\phi(v_i)} \subseteq A$. $E = \{e_1, \cdots, e_m\}$ is a set of directed edges and R defines the edge types in G. Each edge $e_k = (v_i, r_i, v_j)$ indicates a relationship from v_i to v_j of relation type $r_i \in R$. Each node type can include a different set of attribute types resulting in initial node features with variable lengths. Figure 1 shows the schema of an example enterprise KG.

3 SYSTEM OVERVIEW

The system architecture for distributed training of KG embedding models using Ray is shown in Figure 2. The input to the system is the original graph, node attributes and relations, and the p-partitions of the graphs. The system consists of two main components: distributed data preprocessing and distributed training component. The distributed data preprocessing component takes in raw attributes of nodes and processes them in a distributed setting using Ray to produce k-dimensional feature vectors. The objective of the distributed training component is to learn a KG embedding model. The components of the distributed setting are described below.

3.1 Distributed Data Preprocessing

Our input data was comprised of node attributes that had to be encoded to feature vectors for the purpose of training the GNN. These node attributes can be of different formats such as text, numerical, date, and categorical types. Specifically, dates are converted into numerical values, we used scikit-learn *KBinsDiscretizer* and *OneHotEncoder* (or *MultiLabelBinarizer*) to encode the numerical and categorical values, and we used a pre-trained *BERT* language model for encoding the text attributes.

The process of encoding the node attributes was divided into two stages: in the first stage, we loaded a pre-trained *BERT* model and fit the KBinsDiscretizer, OneHotEncoder, and MultiLabelBinarizer model to all the nodes in the input data. The process of learning the models was not executed in distributed manner. In the second stage, we applied the learned models to the node attributes to transform them into feature vectors. This transformation step was executed in a distributed manner using Ray.

Using *@ray.remote*, we implemented a Ray remote worker that receives as an input a subset of nodes with their attributes and a reference to the pre-trained models. The attributes transformation was parallelized by splitting the nodes data and distributing the data among multiple Ray workers. Since all the Ray workers used the same pre-trained scikit-learn and *BERT* models, the models were placed in the distributed Plasma object store using *ray.put()*, and the returned object ID was passed to the workers.

The transformation results were gathered from the Ray workers using *ray.get()* and were concatenated into one array that contained the transformed node attributes for all the data. The running time measurements of the distributed attributes processing are reported in Section 4.2.

To cope with the heterogeneity of the entity attributes e.g. in Figure 1, we modeled the attributes as the immediate neighbors to the entity and used a 1-hop GCN [7] like network without self-loop to obtain the entity attribute embedding. That is, we used a virtual node to represent an entity and its attribute embedding is the result of an aggregation of its attribute neighbors' embeddings. Different from GCN, each attribute type has a dedicated embedding matrix for the transformation. Note that this created entity-attribute graph is only used for attribute embedding but not for the later graph convolutional layers. The entity embeddings are used as initial node features during knowledge embedding model training.

3.2 Distributed Training

We use the *Ray Train* API for distributed training of our GNN model. The API exposes a *Trainer* class for users. Users have to pass backend and a training logic function as input. The *Trainer* creates an Executor process and handles callbacks from the training logic function. The Executor is responsible for distributed training. It creates an actor group, and it passes worker resources, number of resources, and placement strategies to worker group

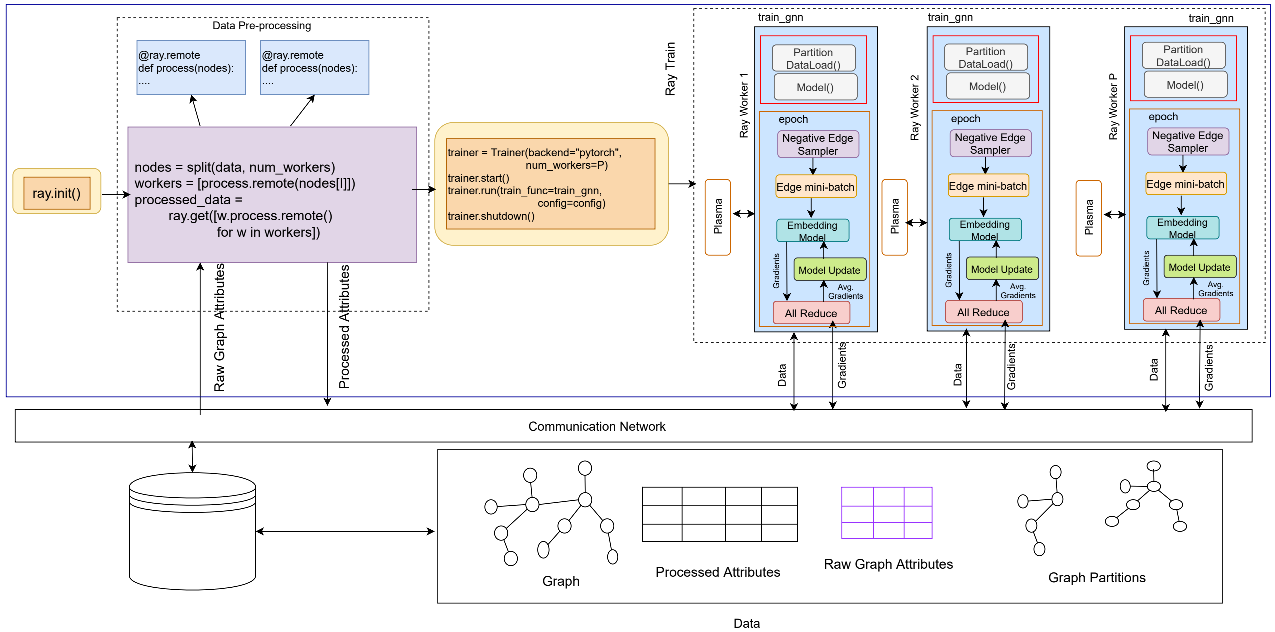


Figure 2: Architecture for Training KG Embedding Models in Ray

processes. *Ray Train* also provides flexibility in choosing the backend. It supports libraries *Torch*, *Tensorflow* and *Horovod*.

We use the training architecture described in [16] and encapsulate it in *Ray Train* for distributed training as shown in Figure 2. When the workers are launched using *trainer.run()*, each worker[2] loads its graph partition and initializes the model for training. The number of workers is determined by the number of graph partitions. The training function (train_gnn) logic is followed using [16]. After each epoch, *Ray Train* allows to collect intermediate results from the distributed training workers. We use PyTorch as our backend. Hence, *Ray Train* will use torch.distributed for gradient sharing and update.

4 EXPERIMENTAL EVALUATION

In this section, we describe the evaluation settings of the KG embedding model training using the data described in Section 2. The data is comprised of 2.8M nodes, 24M edges and 5 relation types. The feature vector per node obtained from processing the node attributes had a dimension of 64. For training the KG embedding model, we divided the data into train, test and validation sets. We randomly chose 80% of the edges as training edges, and test and validation sets use the 10% remaining edges each.

4.1 System Setup

We use *Ray 1.8.0* [11] as the distributed training framework, *PyTorch Geometric 1.7.2* [5] as the graph embedding framework, and *PyTorch 1.9.0* [13] as the deep learning backend.

We ran our experiments on a cluster of 4 machines. Each node had two Intel Xeon 6138 CPUs @ 2.00 GHz (80 virtual cores), 726 GB DDR4 DRAM @ 2666 MT/s, 40 Gb Ethernet, 2 P100 GPUs with 16GB RAM, and was running CentOS Linux 7.9.

We used RGCN [15] and RelGNN [14] to learn the embeddings of nodes and relationships. For both models, we used three neural network layers: one for attribute embedding, and two graph convolutional layers. The hidden dimension size of GNN-layers is 64, and we found that an embedding size of 32 dimensions produced good quality results. For RGCN, we set the hyperparameters as follows: learning rate 0.01, dropout 0.2. For RelGNN, we used the following hyperparameters: regularization 0.001, learning rate 0.001, and Adam decay 0.0001.

4.2 Results of Node Attributes Processing

Our input data contained 2.8M nodes that were evenly distributed among the Ray workers for preprocessing, where each worker was assigned to one GPU or CPU depending on the task. Since *scikit-learn* does not support GPU computations, the attribute transformations were computed on a CPU, whereas the *BERT* transformations were computed on a GPU. We measured the attributes processing time for 1, 2, 4, 6 and 8 workers.

As shown in Figure 3, the processing time for all the node attributes using one worker was 10K seconds and by using 8 workers, the processing time was reduced to 2500 seconds. While increasing the number of workers from 1 to 2 reduced the processing time by 2×, going up to 8 workers reduced the processing time by only 4×. Although we achieved a significant speedup using the distributed processing setup, we did not achieve linear scalability by using the full Ray cluster resources. To explain this behavior, we compared the net workers' processing time (averaged across all workers) with the overall processing time that included retrieving the results. As shown in the figure, while the net processing time scaled linearly and reduced from 6912 seconds with 1 worker to 846 seconds with 8 workers, there was a significant overhead introduced by the *ray.get*() operation. This operation is necessary to collect the results from all the workers which are distributed across the cluster.

[2] train_gnn is the worker process

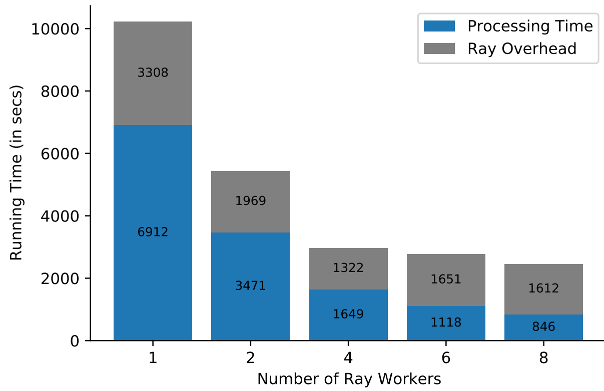
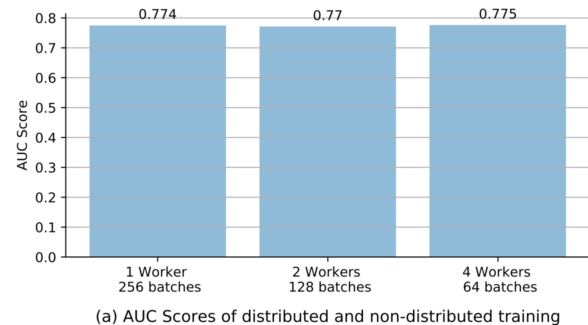


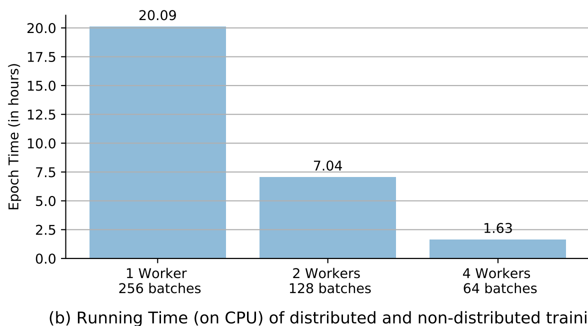
Figure 3: Distributed node attributes processing time with Ray.

4.3 Results of Distributed Training

We evaluated the accuracy and scalability performance of distributed training using Ray against non-distributed training for the link prediction task. We used the same hyperparameters in both settings. RGCN and RelGNN treats link prediction as a binary classification task, i.e. predict if a given triplet is valid or not.



(a) AUC Scores of distributed and non-distributed training

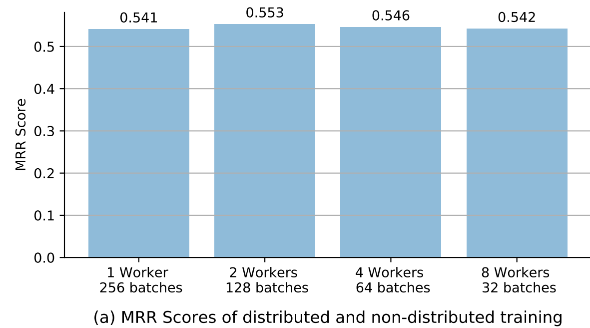


(b) Running Time (on CPU) of distributed and non-distributed training

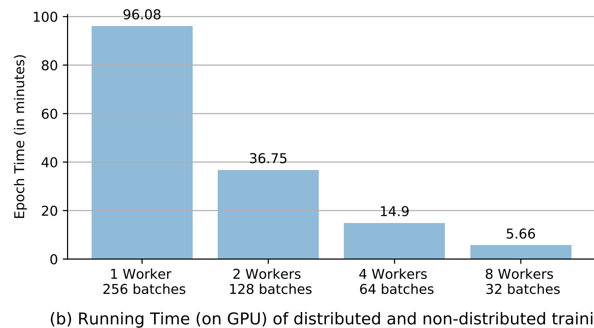
Figure 4: AUC and epoch time of RelGNN Training in Ray (1 worker represents training in non-distributed settings)

For RelGNN, we evaluated the training accuracy on the test set using Area Under the Curve (AUC) of Receiver Operating Characteristic curve (ROC). Since both the model and input data were large, we could not take advantage of GPU acceleration due to limited GPU memory. Thus, we performed the training on CPUs and used neighborhood sampling. In neighborhood sampling, a fixed number of node neighbors are randomly selected. We selected 25 and 10 neighbors as first-hop and second-hop neighbors, respectively. The knowledge graph was partitioned into 2 and 4 partitions for training using 2 and 4 Ray workers. We trained using the edge mini-batch approach and used 256, 128, and 64 batches for 1, 2, 4 workers, respectively. This approach yielded the same number of training edges in each training scenario.

In the case of RGCN, we evaluated the performance on Mean Reciprocal Rank (MRR) metric of the test set. MRR was calculated by averaging the reciprocal ranks of the test set. We partitioned the graph into 2,4, and 8 partitions to train using 2,4, and 8 workers on GPUs. For non-distributed training, i.e single trainer process, we used a batch size of 256 which led to 128, 64 and 32 batches for 2, 4 and 8 workers, respectively.



(a) MRR Scores of distributed and non-distributed training



(b) Running Time (on GPU) of distributed and non-distributed training

Figure 5: MRR and epoch time of RGCN Training in Ray (1 worker represents training in non-distributed settings)

In the non-distributed setting, training on the whole graph was performed using only one trainer process, and it did not involve any gradient sharing. From Figure 4(a), it is evident that the AUC scores of the distributed training were comparable with the non-distributed settings. With respect to scaling, we were able to achieve approximately 12× speedup for 4 workers as shown in Figure 4(b). The results produced using Ray are in agreement with the non-Ray training approach of [16]. We achieved similar results for distributed training of RGCN as shown in Figure 5(a,b). The MRR scores obtained for distributed and non-distributed training were similar. Moreover, we were able to achieve approximately 16× speedup with 8 trainers. In both KG embedding models, super-linear speedup is primarily achieved by the large time decrease in the compute intensive computational graph formulation which is proportional to the size of the partition. Sheikh et al. [16] presents the detailed discussion of this super-linear speedup.

The overhead introduced by using the *Ray Trainer* API was negligible, approximately 5 seconds at the start of training. We also performed evaluation using *pytorch.distributed* API, and found that there were no significant differences in running times

between these two approaches. Thus, developing applications using Ray APIs simplifies the distributed training process with no or negligible additional timing cost, where Ray takes care of managing tasks, such as launching the training session.

5 CONCLUSION

We implemented an end-to-end system for distributed training of a knowledge embedding model using Ray APIs. Ray provides easy-to-use, powerful, and flexible APIs that hide the low-level underlying details for scaling and deployment. We used these APIs effectively for distributed data preprocessing and training.

Our experimental evaluation on a Ray cluster showed that we were able to achieve a 4× speedup on the data preprocessing task. On the distributed training task we achieved a speedup of 12× on RelGNN and 16× on RGCN, and obtained the same AUC/MRR scores as in the non-distributed training setting. The overhead introduced by Ray APIs was negligible.

As future work, we plan to utilize Ray Serve to support scalable model inference over input graph and integrate business services for task specific requests. We will build the training and inference as containerized applications[3] and deploy them in a Kubernetes[4] cluster.

REFERENCES

[1] 2021. RayTrain: Distributed Training Wrappers. https://docs.ray.io/en/latest/train/train.html.

[2] 2021. Serve: Scalable and Programmable Serving. https://docs.ray.io/en/latest/serve/index.html.

[3] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, and Oksana Yakhnenko. 2013. Translating Embeddings for Modeling Multi-Relational Data. In *Proceedings of the 26th International Conference on Neural Information Processing Systems (NIPS'13)*. Red Hook, NY, USA, 2787–2795.

[4] Liwei Cai and William Yang Wang. 2017. KBGAN: Adversarial Learning for Knowledge Graph Embeddings. *CoRR* abs/1711.04071 (2017). arXiv:1711.04071 http://arxiv.org/abs/1711.04071

[5] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[6] Yanchao Hao, Yuanzhe Zhang, Kang Liu, Shizhu He, Zhanyi Liu, Hua Wu, and Jun Zhao. 2017. An End-to-End Model for Question Answering over Knowledge Base with Cross-Attention Combining Global Knowledge. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*. 221–231. https://doi.org/10.18653/v1/P17-1021

[7] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

[8] Adam Lerer, Ledell Wu, Jiajun Shen, Timothée Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. 2019. Pytorch-BigGraph: A Large Scale Graph Embedding System. In *Proc. of Machine Learning and Systems 2019, MLSys 2019*.

[9] Eric Liang, Zhanghao Wu, Michael Luo, Sven Mika, Joseph E. Gonzalez, and Ion Stoica. 2021. RLlib Flow: Distributed Reinforcement Learning is a Dataflow Problem. arXiv:cs.LG/2011.12719

[10] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. 2018. Tune: A Research Platform for Distributed Model Selection and Training. arXiv:cs.LG/1807.05118

[11] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. arXiv:cs.DC/1712.05889

[12] Robert Nishihara, Philipp Moritz, Stephanie Wang, Alexey Tumanov, William Paul, Johann Schleier-Smith, Richard Liaw, Mehrdad Niknami, Michael I. Jordan, and Ion Stoica. 2017. Real-Time Machine Learning: The Missing Pieces.

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*.

[3] https://www.docker.com/

[4] https://kubernetes.io/

[14] Xiao Qin, Nasrullah Sheikh, Berthold Reinwald, and Lingfei Wu. 2021. Relation-aware Graph Attention Model with Adaptive Self-adversarial Training. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 11 (May 2021), 9368–9376.

[15] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings*. Springer.

[16] Nasrullah Sheikh, Xiao Qin, Berthold Reinwald, and Chuan Lei. 2022. Scaling Knowledge Graph Embedding Models. *CoRR* abs/2201.02791 (2022). arXiv:2201.02791 https://arxiv.org/abs/2201.02791

[17] Nasrullah Sheikh, Xiao Qin, Berthold Reinwald, Christoph Miksovic, Thomas Gschwind, and Paolo Scotton. 2021. Knowledge Graph Embedding using Graph Convolutional Networks with Relation-Aware Attention. *CoRR* abs/2102.07200 (2021). arXiv:2102.07200 https://arxiv.org/abs/2102.07200

[18] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. 2016. Complex Embeddings for Simple Link Prediction. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16)*.

[19] Zhichun Wang, Qingsong Lv, Xiaohan Lan, and Yu Zhang. 2018. Cross-lingual Knowledge Graph Alignment via Graph Convolutional Networks. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Brussels, Belgium.

[20] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. 2015. Embedding Entities and Relations for Learning and Inference in Knowledge Bases. In *3rd Int. Conference on Learning Representations, ICLR San Diego, CA, USA*.

[21] Ningyu Zhang, Shumin Deng, Zhanlin Sun, Guanying Wang, Xi Chen, Wei Zhang, and Huajun Chen. 2019. Long-tail Relation Extraction via Knowledge Graph Embeddings and Graph Convolution Networks. In *Proc of the 2019 Confof the North American Chapter of the ACL: Human Language Technologies*.

[22] Da Zheng, Xiang Song, Chao Ma, Zeyuan Tan, Zihao Ye, Jin Dong, Hao Xiong, Zheng Zhang, and George Karypis. 2020. DGL-KE: Training Knowledge Graph Embeddings at Scale. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '20)*. Association for Computing Machinery, New York, NY, USA, 739–748.