Learning Knowledge Graph Embeddings at scale

Udit Gupta (14454) [1] **Nashez Zubair (14973)** [1]

Knowledge Graph Embeddings (KGE) are used to represent the entities and relations of a KG in a low dimensionalvector space. KGEs can then be used in a downstream task such as entity classification, link prediction and knowledge base completion. Over the years several algorithms have been proposed to learn KGEs. The focus of this research area has been to obtain higher accuracy (w.r.t standard metrics) while providing fast computations. Due to the large size of KGs, simple models are preferred as they take less time but may suffer in terms ofaccuracy. This motivates us to distribute the learning of KGEs so that we can train computationally expensive models faster(without compromising on accuracy). In this work we parallelize 3 different models, using Horovod

1. Introduction

A Knowledge Graph (KG) is a multi-relational graph composed of entities as nodes and relations as edges. Each instance in the KG is represented as a triple also called a fact (head entity, relation, tail entity; denoted as (h, r, t) or (subject, predicate, object)) where relation (or predicate) models the relationship between the two entities. The models proposed for learning KGEs can broadly be classified as additive(Sharma et al., 2018) (entity and relation vectors interact via addition and subtraction) and multiplicative(Sharma et al., 2018) (head-relation-tail triple likelihood is measured by a multiplicative score function). For a large knowledge base consisting of billions of training instances the training may take several days. This motivates us to explore learning KGEs in a distributed setting such that we reduce training time without significant drop in accuracy. A recent work (Goyal et al., 2017), by Facebook researchers, parallelized a deep neural network efficiently using the concepts of large scale machine learning and parallel programming. However, the problem of learning KGEs in a parallel setting has been unexplored and our work makes progress in this direction. In this work we make the following contributions:

1. A sequential single node model selects a minibatch by random sampling from the dataset. In the distributed setting we experiment with 3 different variations of sampling across various nodes and present results.

2. We compare 3 different additive models and parallelize the learning of KGE for these 3 algorithms.

2. Related Work

As the datasets become larger and models become more complex single node computations take more amount of time. Ideally, we would want to work with very large datasets (such that it nearly depicts the whole population) as it will allow us to work with more complex models. Deeplearning has shown promising results for image and NLP related tasks, however the amount of time required to train a deep network can span over several days (e.g. Resnet-50 on Imagenet dataset) if we use a single node for computations. This proves to be a bottleneck for hyperparameter optimization and inference since any changes made to the model would require many days for generating results. This has prompted researchers to look into distributed training of deep learning models. A recent work by Facebook researchers (Goyal et al., 2017) trained a deep learning model (Resnet 50) on Imagenet dataset using 256 GPUs which obtained a scaling efficiency of 90% while maintaining the accuracy (as obtained on single node). In their work they proposed techniques like *Linear scaling*, *warmup* and *batch normalisation* which helped them in training the model in less than an hour without compromising on accuracy.
A similar approach can also be applied to learn KGEs. In spite of growing sizes of knowledge bases there hasn't been much effort to parallelize the learning of KGEs.
In our work we use Horovod (Sergeev & Del Balso, 2018) with Tensorflow to parallelize the KGE learning process. Horovod uses the ring-allreduce algorihtm (Patarasuk & Yuan, 2009) for aggregation of gradients from all workers. Unlike the parameter server approach (which results in network bottleneck), this algorithm is bandwidth optimal as the workers are arranged in logical ring and each worker communicates only with its neighboring two workers. Horovod is preferred over vanilla distributed Tensorflow because training with Horovod was shown to be twice as fast as compared to standard Tensorflow (Sergeev & Del Balso,

[1]Department of Computational and Data Sciences, Indian Institute of Science.

Project Report for the course Machine Learning with large datasets (DS-222)

2018) (compared using official Tensorflow benchmarks on Inception V3 and Resnet-101 models).

2.1. KGE models used for experimentation

ADDITIVE MODELS

Models in which entity and relation vectors interact via additive operations.

1. **TransE**: This model is based on the assumption that the embedding of head entity (h) plus the embedding of relation (connecting h with t) should be close to the embedding of the tail entity (Bordes et al., 2013). The loss function (for one instance in a minibatch) is formulated as:

$$max(0, \gamma + d(h + r,\ t) - d(h' + r,\ t'))$$

where γ is a margin based hyperparameter.
$d(x, y)$ is a distance metric (such as L2 norm) between x and y
h', r, t' corresponds to the corrupted triplet that is generated for an instance h, r, t using negative sampling.

2. **TransH**: This model overcomes the issues when handling reflexive relations. In this model we have two vectors for each relation; a translation vector (d_r) and a normal vector to the hyperplane (w_r) (Wang et al., 2014). The loss function is formulated as:

$$max(0, \gamma + d(h_\perp + d_r,\ t_\perp) - d(h'_\perp + d_r,\ t'_\perp))$$

where $h_\perp = h - w_r^T h w_r$ and $t_\perp = t - w_r^T t w_r$

3. **TransD**: This model uses two vectors to represent each entity and relation. One vector is used for translation and the other is used for projection to construct a mapping matrix (Ji et al., 2015). The loss function is formulated as:

$$max(0, \gamma + d(h_\perp + d_r,\ t_\perp) - d(h'_\perp + d_r,\ t'_\perp))$$

where $h_\perp = M_{rh} h$ and $t_\perp = M_{rt} t$
$M_{rh} = r_p h_p^T + I$; $M_{rt} = r_p t_p^T + I$
h, r, t are translation vectors.
h_p, r_p, t_p are projection vectors used to construct mapping matrices.

3. Methodology

The standard practice followed by researchers for training is that they randomly sample a batch of size *batch_size* from the dataset, compute a loss function (particular to the model being implemented) and then use an optimization algorithm such as SGD to update the vector embeddings. In our work we parallelize the learning (or updation) of vector embeddings. Each worker builds its own computation graph and has access to all the parameters (the entity embedding matrix and the relation embedding matrix). Here we use the *Bulk Synchronous parallelism* paradigm for parameter updates. After computation on a batch of input, gradient aggregation (averaging in this case) is performed using the ring all-reduce algorithm (Patarasuk & Yuan, 2009) and the parameters are updated. So, after each batch computation, every worker has the same set of parameters.

3.1. Distributing the training process

We adopt a data parallel approach for distributing the training process across multiple workers. Let's say we operate on a single worker with minibatch size of *batch_size*. When we distribute training across N workers we ensure that each worker operates with the same minibatch size of *batch_size* thus the overall minibatch size for the whole system becomes $N * batch\_size$. Therefore, the number of batches (per epoch) required in distributed setting becomes $nbatches/N$; where $nbatches$ $(= \#training\_instances/batch\_size)$ is the number of batches required per epoch for a single worker.

3.2. Distributed KGE models

The distributed training process is descibed as follows:

1. For distributing work among N workers, N processes are spawned such that each process is pinned to a separate GPU.
2. Here we do synchronous distributed training in which the minibatch size is scaled by the number workers (GPUs in our case); to compensate for this, the learning rate is also scaled by the number of workers (as suggested in (Goyal et al., 2017)).
3. The computational graph for a KGE algorithm (eg. TransE) is built on each worker using Tensorflow.
4. After each minibatch computation, gradient averaging is performed using allreduce and allgather operations using the ring-allreduce algorithm and every worker obtains the updated parameters. (Patarasuk & Yuan, 2009).

4. Experiment setup

4.1. Dataset

FB15K and WN18 are widely used datasets for the task of link prediction in Knowledge graphs. In (Sharma et al., 2018) it was shown that the KGEs learnt by different KGE algorithms show the same characteristic for FB15K and WN18 datasets. For this work we perform experiments with FB15K dataset.

4.2. Hyperparameters

1. The dimensionality of the vector embedding for both, the entities and relations, is set as 100.

2. We generate 1 negative sample per training instance in which either the head or the tail entity is corrupted randomly.

3. The margin used in the loss function i.e. $\gamma = 1$.

4. We use SGD optimzation algorithm for all KGE models unless otherwise specified.

5. The *batch_size* (and hence *nbatches*) and the *training_epochs* are determined by doing prior experiments. For the results presented in section *V* the *training_epochs* and the *nbatches* are specified under each section. These values have been determined (by experiments not shown in this report) such that we can distinguish between the strategies and algorithms in the multi-worker setting.

4.3. Evaluation protocol

We follow standard evalutaion protocol as was followed by TransE (Bordes et al., 2013). We present results for 2 metrics i.e. *Mean Rank* (lower the better) and *Hits@10* (higher the better). To calculate the mean rank we replace the tail by every other entity in the KG and calculate the score function. The scores are then ranked in ascending order and the rank of the correct triple is noted. This procedure is repeated for every instance in the test dataset and the ranks are averaged to get the Mean Rank. Finally this procedure is repeated by replacing the head instead of the tail and a Mean Rank metric corresponding to head is obtained. The hits@10 metric is computed by considering the correct entities ranked in the top 10 after sorting them according to the score function's output.
The setting described above is termed as *raw*. But this evaluation may be flawed when the corrupted triple is already present in the dataset; In this case the corrupted triple may be ranked above the original triple as both of them are correct facts occurring in the KG. So, we also consider a different evaluation setting termed as *filtered* in which the corrupted triples occurring in the dataset are not considered.

4.4. Resource used for computation

The experiments are performed on Cray XC40 CPU cluster which has Intel Xeon Haswell processors. On this machine experiments were performed with 1, 2, 4, 8 and 16 nodes with 24 cores in each node.

5. Results

5.1. Distributed KGE models using sparse updates

For each model we did a convergence analysis (using triple classification accuracy as the metric) on the validation set to determine the number of epochs to run for each algorithm.

BATCH SIZE = 30 ; #BATCHES = 16384

1. TransE:
   Epochs to convergence: 144

# Workers	Mean Rank		Hits@10		Triple	Training time
	raw	filter	raw	filter	classification acc%	per epoch (sec)
1	203	73	0.44	0.62	81.04	18.40
2	203	73	0.44	0.62	81.27	43.75
4	203	72	0.44	0.62	81.46	37.30
8	204	74	0.44	0.62	81.72	11.14
16	205	74	0.44	0.62	81.25	6.875

2. TransH:
   Epochs to convergence: 167

# Workers	Mean Rank		Hits@10		Triple	Training time
	raw	filter	raw	filter	classification acc%	per epoch (sec)
1	231	88	0.43	0.62	86.28	23.19
2	230	87	0.42	0.62	86.39	78.92
4	231	88	0.43	0.62	86.23	37.17
8	230	87	0.43	0.62	86.36	20.20
16	230	88	0.43	0.62	86.28	10.91

3. TransD:
   Epochs to convergence: 81

# Workers	Mean Rank		Hits@10		Triple	Training time
	raw	filter	raw	filter	classification acc%	per epoch (sec)
1	231	96	0.41	0.57	85.74	21.48
2	231	97	0.41	0.57	86.15	73.93
4	233	99	0.41	0.57	85.90	35.90
8	232	98	0.41	0.57	85.81	11.70
16	234	100	0.41	0.57	85.70	21.56

BATCH SIZE = 500 ; #BATCHES = 1024

1. TransE:
   Epochs to convergence: 240

# Workers	Mean Rank		Hits@10		Triple	Training time
	raw	filter	raw	filter	classification acc%	per epoch (sec)
1	204	67	0.47	0.66	83.19	1.89
2	206	68	0.46	0.66	83.84	9.22
4	204	66	0.46	0.66	83.69	5.50
8	203	67	0.47	0.66	83.73	6.03
16	205	68	0.46	0.66	83.63	4.36

2. TransH:
   Epochs to convergence: 323

# Workers	Mean Rank		Hits@10		Triple classification acc%	Training time per epoch (sec)
	raw	filter	raw	filter		
1	235	87	0.44	0.66	87.37	2.81
2	233	90	0.43	0.62	86.15	27.14
4	230	87	0.43	0.62	86.22	11.92
8	232	84	0.44	0.66	87.47	10.08
16	231	83	0.44	0.66	87.70	7.09

3. TransD:
   Epochs to convergence: 456

# Workers	Mean Rank		Hits@10		Triple classification acc%	Training time per epoch (sec)
	raw	filter	raw	filter		
1	235	86	0.47	0.70	88.6	2.82
2	235	87	0.47	0.70	88.74	18.23
4	234	86	0.47	0.70	88.81	16.46
8	235	86	0.47	0.70	88.68	15.52
16	232	84	0.47	0.70	88.66	11.81

5.2. Observations

1. The models scale better when we reduce the batch size.
2. TransD model does not scale well with increased number of workers.
3. If we increase the batch size, the scalability in terms of speedup goes down.

5.3. Analysis

1. For small batch sizes we have few entities and relations to be updated. We found that using allgather (rather than allreduce) works better in this case as it collects the sparse gradients from all workers and updates the parameters accordingly.
2. For large batch sizes there are more number of parameters to be updated and hence updates are no longer sparse. We found that using allreduce works better for large batch sizes.
3. As the number of workers increase, the overall mini-batch size increases, hence we do not benefit from sparse updates and hence allreduce is preferred over allgather.

5.4. Distributed KGE models using dense updates

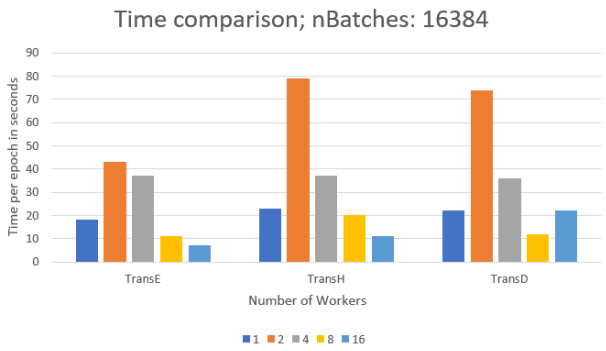
BATCH SIZE = 500 ; #BATCHES = 1024

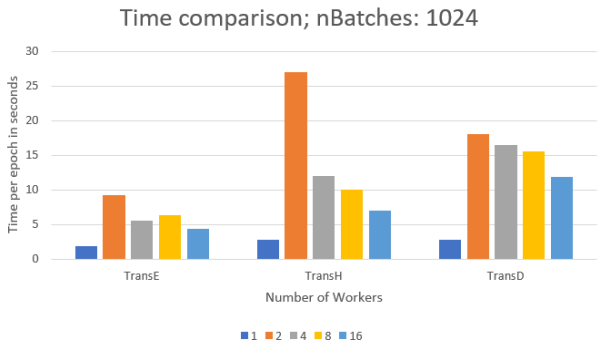
TransE:
Epochs to convergence: 240

# Workers	Mean Rank		Hits@10		Triple classification acc%	Training time per epoch (sec)
	raw	filter	raw	filter		
1	204	67	0.47	0.66	83.19	1.89
2	205	67	0.46	0.66	83.24	9.22
4	205	68	0.46	0.66	83.58	8.77
8	204	67	0.47	0.66	83.69	4.68
16	205	68	0.46	0.65	83.59	2.43
32	213	74	0.46	0.65	83.02	1.21

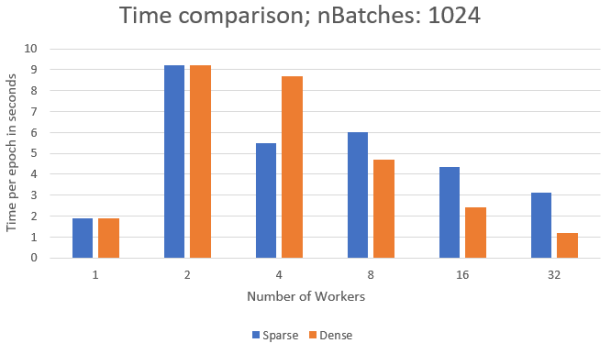
5.5. Comparisons

5.5.1. W.R.T BATCH SIZES





5.5.2. SPARSE VS DENSE UPDATES FOR TRANSE



6. Conclusion and Future Work

In this work we initiate a study to look into the scope for parallelization of different KGE models. We present the time and accuracy comparisons for various models when multiple workers are utilized. Through a comparitive study we analysed the role of batch size in the scalabilty of models across multiple workers. We observed that using small batch sizes we can achieve higher speedup as compared to large batch sizes. Also we found that as the number of workers

increase it is better to do dense updates for achieving good speedup.

This work explores the data parallel approach to achieve scalability in which all the parameters are available with all workers. In some cases it might be required to split the parameters across multiple workers due to memory constraints. The future work will involve exploring this model parallel approach.

References

Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., and Yakhnenko, O. Translating embeddings for modeling multi-relational data. In *Advances in neural information processing systems*, pp. 2787–2795, 2013.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Ji, G., He, S., Xu, L., Liu, K., and Zhao, J. Knowledge graph embedding via dynamic mapping matrix. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, volume 1, pp. 687–696, 2015.

Patarasuk, P. and Yuan, X. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

Sergeev, A. and Del Balso, M. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

Sharma, A., Talukdar, P., et al. Towards understanding the geometry of knowledge graph embeddings. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pp. 122–131, 2018.

Wang, Z., Zhang, J., Feng, J., and Chen, Z. Knowledge graph embedding by translating on hyperplanes. In *AAAI*, volume 14, pp. 1112–1119, 2014.