PYTORCH-BIGGRAPH: A LARGE-SCALE GRAPH EMBEDDING SYSTEM

ABSTRACT

Graph embedding methods produce unsupervised node features from graphs that can then be used for a variety of machine learning tasks. However, modern graph datasets contain billions of nodes and trillions of edges, which exceeds the capability of existing embedding systems. We present Pytorch-BigGraph (PBG), an embedding system that incorporates several modifications to traditional multi-relation embedding systems that allow it to scale to graphs with billions of nodes and trillions of edges. PBG uses graph partitioning to train arbitrarily large embeddings on either a single machine or in a distributed environment. We evaluate demonstrate comparable performance with existing embedding systems on common benchmarks, while allowing for scaling to arbitrarily large graphs and parallelization on multiple machines. We perform experiments on the full FreeBase dataset, which contains over 100 million nodes and 2 billion edges.

1 INTRODUCTION

Graph structured data is a common input to a variety of machine learning tasks (Wu et al., 2017; Cook & Holder, 2006; Nickel et al., 2016a; Hamilton et al., 2017b). Graph embedding methods compute a vector for each node such that the distance between vectors predicts the occurrence of an edge in the graph. These embeddings have been have been shown to serve as useful features for downstream tasks such as recommender systems in ecommerce (Wang et al., 2018), link prediction in social media (Perozzi et al., 2014), predicting drug interactions and characterizing protein-protein networks (Zitnik & Leskovec, 2017).

At modern web companies large graph data is common. For example the Facebook graph includes over two billion user nodes and over a trillion edges representing friendships, likes, posts and other connections (Ching et al., 2015). The graph of users and products at Alibaba also consists of more than one billion users and two billion items (Wang et al., 2018). At Pinterest, the user to item graph includes over 2 billion entities and over 17 billion edges (Ying et al., 2018).

There are two main challenges for embedding graphs of this size. First, an embedding system must be fast enough to embed graphs with $10^{11} - 10^{12}$ edges in a reasonable time. Second, a model with two billion nodes and 32 dimensional latent vectors (expressed as floats) would require 800GB of memory just to store its parameters, thus many standard methods are beyond the memory capacity of typical commodity servers.

We present Pytorch-BigGraph (PBG), an embedding system that incorporates several modifications to standard models. The contribution of PBG is to scale to graphs with billions of nodes and trillions of edges. The key components of PBG are:

- A block decomposition of the adjacency matrix into N buckets, training on the edges from one bucket at a time. PBG then either swaps embeddings from each partition to disk to reduce memory usage, or performs distributed execution across multiple machines.
- A distributed execution model that leverages the block decomposition for the large parameter matrices, as well as a parameter server architecture for global parameters and feature embeddings for featurized nodes.
- Efficient negative sampling for nodes that samples negative nodes both uniformly and from the data, and reuses negatives within a batch to reduce memory bandwidth.
- Support for multi-entity, multi-relation graphs with per-relation configuration options such as edge weight and choice of relation operator.

We evaluate PBG on the FreeBase, LiveJournal and YouTube graphs and show that it matches the performance of existing embedding systems, with a 75% reduction in training time.

We report results on an embedding of the full FreeBase knowledge graph (121 million entities, 2.4 billion edges), which we release publicly with this paper. Partitioning of the FreeBase graph reduces memory consumption by 88% with only a small degradation in the embedding quality, and distributed execution on 8 machines decreases training time by a factor of 4.

PBG will be released as an open source project. It is written

entirely in Pytorch (Paszke et al., 2017) with no external dependencies or custom operators.

2 Related Work

Many types of models have been developed for multi-relation graphs (Bordes et al., 2011; 2013; Nickel et al., 2011; Trouillon et al., 2016). Typically these models have been used in the context of entity representations in knowledge bases (e.g. FreeBase or WordNet). Entities are given a base vector, these vectors are transformed by a learned function for each transformation, and existence of edges is predicted by some distance measure in the new space. More recent work by Wu et al. proposes modeling some entities as bags of other entities (rather than giving them explicit embeddings). PBG borrows many insights on loss functions and transformations from this literature.

There are significant engineering challenges to scaling graph embedding models. Proposed approaches in the literature include multi-level methods (Liang et al., 2018), distributed embedding systems (Ordentlich et al., 2016; Shazeer et al., 2016), as well as specialized methods for standard algorithms such as SVD and k-means on large graphs (Ching et al., 2015). Gains from large embedding systems have been documented in e-commerce (Wang et al., 2018) and other applications.

There is an extensive literature on distributional semantics in natural language processing. A key breakthrough in this literature are algorithms such as word2vec which allowed word embedding methods to scale to larger corpora (Mikolov et al., 2013). Recent work has shown that there is economic value from ingesting even larger data sets using distributed word2vec systems (Ordentlich et al., 2016).

There is substantial prior work on scalable parallel algorithms for training machine learning models (Dean et al., 2012). Highly related to PBG is work on scaling various forms of matrix factorization (Gupta et al., 1997; Gemulla et al., 2011). Matrix factorization is closely related to embeddings, and has had widespread success in recommender systems (Koren et al., 2009).

Recent work proposes to construct embeddings by using graph convolutional neural networks (GCNs, (Kipf & Welling, 2016)). These methods have shown success when applied to problems at large-scale web companies (Hamilton et al., 2017a; Ying et al., 2018). The problem studied by the GCN is different than the one solved by PBG (mostly in that GCNs are typically applied to graphs where the nodes are already featurized). Combining ideas from graph embedding and GCN models is an interesting future direction both for theory and applications.

3 Multi-Relation Embeddings

3.1 Model

A multi-relation graph is a directed graph $G = (V, R, E)$ where V are the nodes, R is a set of relation types, and E is a set of edges where a generic element $e = (h, r, t)$ (head, relation, tail) where $h, t \in V$ and $r \in R$. We also discuss graphs that have multiple *entity types*. Such graphs have a set of entity types and a mapping from nodes to entity types, and each relation specifies a single entity type for head and tail nodes for all edges of that relation.

We will represent each entity and relation type with a vector of parameters. We will denote this vector as θ. A multi-relation graph embedding uses a score function $f(\theta_h, \theta_r, \theta_t)$ that produces a score for each edge that attempts to maximize the score of $f(\theta_h, \theta_r, \theta_t)$ for any $(h, r, t) \in E$ and minimizes it for $(h, r, t) \notin E$.

PBG considers scoring functions defined by a distance metric between a transformed version of an edge's head and tail entities' vectors (θ_h, θ_t):

$$f(\theta_h, \theta_r, \theta_t) = \text{dist}(R_1(\theta_h, \theta_r), R_2(\theta_t, \theta_r))$$

where θ_r corresponds to parameters of the relation-specific transformation operator. Using distance-based scoring functions produces a embeddings where the (transformed) distance between nodes has semantic meaning.

PBG uses a cosine distance metric by default, and a relation operator g_r which is either a linear transformation $g_r(t) = A_r t$ or translation $g_r(t) = t + \Delta_r$ for g_r, which correspond to the RESCAL and transE models, respectively (Nickel et al., 2011; Bordes et al., 2013), as well as an affine transform $g_r(t) = A_r t + \Delta_r$ that nests them. [1] A subset of relation types may use the identity relation, so that the untransformed entity embeddings predict edges of this relation.

We consider sparse graphs, so the input to PBG is a list of positive (existing) edges. We work with sparse graphs so we construct negative examples by sampling. In PBG negative samples are generated by corrupting positive edges by sampling either a new head or a tail for each existing edge (Bordes et al., 2013).

Because edge distributions in real world graphs are heavy tailed, the choice of how to sample nodes to construct negative examples can affect model quality (Mikolov et al.,

[1] On small knowledge graphs, a general linear transform (RESCAL) does not perform as well as transformations with fewer parameters such as translation (as well as transformations that can be represented in the RESCAL model) because the relation operators overfit (Nickel et al., 2016b). However, we are interested in web interaction graphs which have a very small number of relations relative to entities, so the relation parameters do not contribute substantially to model size, nor are they prone to overfitting.

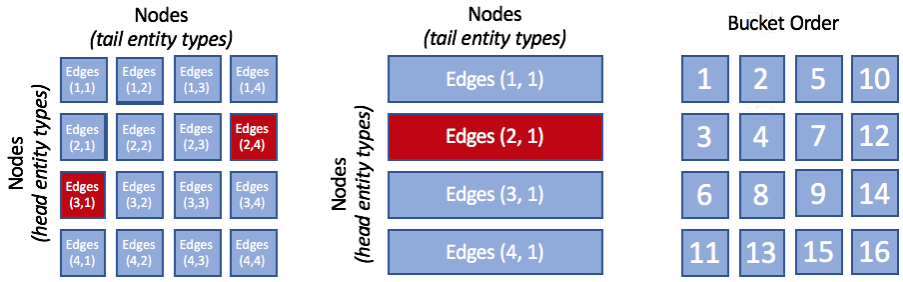

Figure 1. The PBG partitioning scheme for large graphs. **Left:** nodes are divided into P partitions that are sized to fit in memory. Edges are divided into buckets based on the partition of their head and tail nodes. In distributed mode, multiple buckets with non-overlapping partitions can be executed in parallel (red squares). **Center:** Entity types with small cardinality do not have to be partitioned; if all entity types used for tail nodes are unpartitioned, then edges can be divided into P buckets based only on head node partitions. **Right:** the 'inside-out' bucket order guarantees that buckets have at least one previously-trained embedding partition. Empirically, this ordering produces better embeddings than other alternatives (or random)

2013). On one hand, if we sample negatives strictly according to the data distribution, there is no penalty for the model predicting high scores for edges with rare nodes. On the other hand, if we sample negatives uniformly, the model can perform very well (especially in large graphs) by simply scoring edges proportional to their head and tail node frequency in the dataset. Both of these results are undesirable, so in PBG we sample a fraction α of negatives according to their prevalence in the training data and $(1-\alpha)$ of them uniformly at random. By default PBG uses $\alpha = .5$.

In multi-entity graphs, negatives are only sampled from the correct entity type for an edge's relation. Thus, in our model, the probability of an 'invalid' edge (wrong entity types) is undefined. The approach of using entity types has been studied before in the context of knowledge graphs (Krompaß et al., 2015), but we found it to be particularly important in graphs that have entity types with highly unbalanced numbers of nodes, e.g. 1 billion users vs. 1 million products. With uniform negative sampling over all nodes, the loss would be dominated by user negative nodes and would not optimize for ranking between user-product edges.

PBG optimizes a margin-based ranking objective between each edge e in the training data and a set of edges e' constructed by corrupting e with either a sampled head or tail node (but not both).

$$\mathcal{L} = \sum_{e \in G} \sum_{e' \in S'_e} \max(f(e) - f(e') + \lambda, 0))$$

where λ is a margin hyperparameter and

$$S'_e = \{(h', r, t) | h' \in V\} \cup \{(h, r, t' | t' \in V\}\,.$$

Model updates to the embeddings and relation parameters are performed via minibatch stochastic gradient descent (SGD). We use the Adagrad optimizer, and sum the accumulated gradient $\mathcal{G}$ over each embedding vector to reduce memory usage on large graphs (Duchi et al., 2011).

4 Training at Scale

PBG is designed to operate on arbitrarily large graphs running on either a single machine or can be distributed across multiple machines. In either case, training occurs on a number of CPU threads equal to the number of machine cores, with no explicit synchronization between cores as described in (Recht et al., 2011).

4.1 Partitioning of Entities and Edges

PBG uses a partitioning scheme to support models that are too large to fit in memory on a single machine. This partitioning also allows for distributed training of the model.

Each entity type in G can be either partitioned or remain unpartitioned. Partitioned entities are split into P parts. P is chosen such that each part fits into memory or to support the desired level of parallelism for execution.

After entities are partitioned, edges are divided into *buckets* based on their source and tail entities' partitions. For example, if an edge has a head in partition p_1 and tail in partition p_2 then it is placed into bucket (p_1, p_2). This creates P^2 buckets when both head and tail entity types are partitioned and P buckets if only head (or tail) entities are partitioned.

Each epoch of training iterates through each of the edge buckets. For edge bucket (p_i, p_j), head and tail partitions i and j respectively are swapped from disk, and then the edges (or a subset of edges) are loaded and subdivided among the threads for training.

This partitioning creates 2 changes to the base algorithm described in the last section. First, each candidate edge

(h, r, t) is only compared to negatives (h, r, t') in the ranking loss where t' is drawn from the same partition (same for head nodes)[2]. We find that this can degrade performance of smaller models but is not an issue for large models where partitioning is actually needed in practice.

Second, edges are no longer sampled i.i.d. but are grouped by partition. Convergence to a stationary or chain-recurrent point, equivalent to SGD, is still guaranteed under this modification (see (Gemulla et al., 2011), Sec. 4.2), but may suffer from slow convergence[3].

We observe that the order of iterating through edge bucket may affect the final model. Specifically, for each edge bucket (p_1, p_2) except the first, it is important that an edge bucket $(p_1, *)$ or $(*, p_2)$ was trained in a previous iteration. This constraint ensures that embeddings in all partitions are aligned in the same space. For single-machine embeddings, we found that an 'inside-out' ordering, illustrated in Figure 2(c), achieved the best performance while minimizing the number of swaps to disk.

4.2 Distributed Execution

Existing distributed embedding systems typically use a parameter server architecture. In this architecture, a (possibly sharded) parameter server contains a key-value store of embeddings. At each SGD iteration, the embedding parameters required by a minibatch of data are requested from the parameter server, and gradients are (asynchronously) sent to the server to update the parameters.

The parameter server paradigm has been effective for training large sparse models (Li et al., 2014), but it has a number of drawbacks. One issue is that parameter-server based embedding frameworks require too much network bandwidth to run efficiently, since all embeddings for each minibatch of edges and their associated negative samples must be transferred at each SGD step (Ordentlich et al., 2016) [4]. Furthermore, we found it necessary for effective research use that the same models could be run in a single-machine or distributed context, but the parameter server architecture limits the size of models that can be run on a single machine. Finally, we would like to avoid the potential convergence problems from asynchronous model updates since our embeddings are already partitioned into independent sets.

Given partitioned entities and edges PBG employs a parallelization scheme that combines a locking scheme over the model partitions described in Section 4.1, with an asynchronous parameter server architecture for shared parameters for the relation operators as well as unpartitioned or featurized entity types.

In this parallelization scheme, illustrated in Figure 4.2, partitioned embeddings are locked by machines for training. Multiple edge buckets can be trained in parallel as long as they operate on disjoint sets of partitions, as shown in Figure 2. Training can proceed in parallel on up to $P/2$ machines. The locking of partitions is handled by a centralized lock server on one machine, which parcels out buckets to the workers in order to minimize communication (i.e. favors re-using a partition) The lock server also maintains the invariant described in Section **??**, that only the first bucket should operate on two uninitialized partitions.

The partitioned embeddings themselves are stored in a partition server sharded across the N training machines. A machine fetches the head and tail partitions, which are often multiple GB in size, from the partition server, and trains on a bucket of edges loaded from shared disk. Checkpoints of the partitioned entities are intermittently saved to shared disk.

Some model parameters are global and thus cannot be partitioned. This most importantly includes relation parameters, as well as entity types that have very small cardinality or use featurized embeddings. There are a relatively small number of such parameters ($< 10^6$), and they are handled via asynchronous updates with a sharded parameter server. Specifically, each trainer maintains a background thread that has access to all unpartitioned model parameters. This thread asynchronously fetches the parameters from the server and updates the local model, and pushes accumulated gradients from the local model to the parameter server. This thread performs continuous synchronization with some throttling to avoid saturating network bandwidth.

4.3 Batched Negative Sampling

The negative sampling approach used by most graph embedding methods is highly memory (or network) bound because it requires $B \cdot B_n \cdot d$ floats of memory access to perform $O(B \cdot B_n \cdot d)$ floating-point operations ($B \cdot B_n$ dot products). Indeed, Wu et al.. report that "number of epochs ... is close to an inverse linear function of [number of negatives]".

To increase memory efficiency on large graphs, we observe that a single batch of B_n sampled head or tail nodes can be reused to construct multiple negative examples. In a typical setup, we take a batch of $B = 1000$ positive edges from the training set, and break it into chunks of 50 edges. The tail

[2] This would not matter if we were using an independent loss for positives and negatives, e.g. a binary cross-entropy loss

[3] The slower convergence may be ameliorated by switching between the buckets ('stratum losses' (Gemulla et al., 2011)) more frequently, i.e. in each epoch divide the edges from each bucket into N parts and iterate over the buckets N times, operating on one edge part each time. We did not find this to be important in practice.

[4] In fact, our approach to batched negative sampling, described in Section 4.3 reduces the number of negatives that must be retrieved so would require less bandwidth than (Ordentlich et al., 2016) if a parameter server was used.

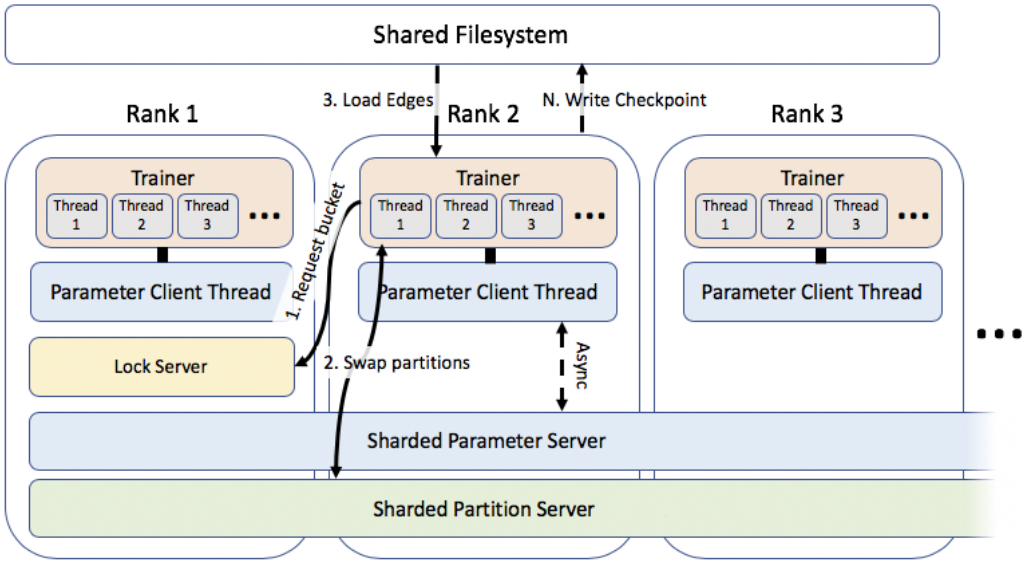

Figure 2. A block diagram of the modules used for PBG's distributed mode. Arrows illustrate the communications that the Rank 2 Trainer performs for the training of one bucket. First, the trainer requests a bucket from the lock server on Rank 1, which locks that bucket's partitions. The trainer then saves any partitions that it is no longer using and loads new partitions that it needs to and from the sharded partition servers, at which point it can release its old partitions on the lock server. Edges are then loaded from a shared filesystem, and training occurs on multiple threads without inter-thread synchronization(Recht et al., 2011). In a separate thread, a small number of shared parameters are continuously synchronized with a sharded parameter server. Model checkpoints are occasionally written to the shared filesystem from the trainers.

(equivalently, head) embeddings from each chunk is concatenated with 50 embeddings sampled uniformly from the tail entity type. The outer product of the 50 positives with the 200 sampled nodes equates to 9900 negative examples (excluding the induced positives). The training computation is summarized in Figure 4.

This approach is much cheaper than sampling negatives for each batch. For each batch of B positive edges, only $3B$ embeddings are fetched from memory and $3BB_n$ edge scores (dot products) are computed. The edge scores for a batch can be computed as a batched $B_n \times B_n$ matrix multiply, which can be executed with high efficiency. Figure 3 shows the performance of PBG with different numbers of negative samples, with and without batched negatives.

In multi-relation graphs with a small number of relations, we construct batches of edges that all share the same relation type r. This improves training speed specifically for the linear relation operator $f_r(t) = A_r t$, because it can be formulated as a matrix-multiply $f_r(T) = A_r T$.

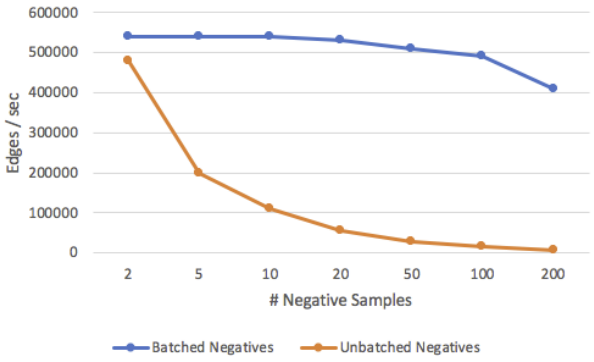

Figure 3. Effect of number of negative samples per edge on training speed ($d = 100$). With unbatched negatives, training speed is inversely proportional to number of negatives, but with batched negatives, speed is nearly constant for $B_n \leq 100$.

5 Experiments

We evaluate PBG on 3 data sets from the literature. The FreeBase knowledge graph (both the commonly used FB15k subset and the full FreeBase graph), and user to user interaction graphs from LiveJournal (Backstrom et al., 2006) and YouTube (Tang & Liu, 2009).

We evaluate PBG in two ways. First, we will compare the loss on the graph embedding task itself (link prediction). Second, a common use-case of embeddings in real world applications is as features in a downstream supervised task. The YouTube data set contains some attributes for each user node. We will evaluate the use of the embeddings

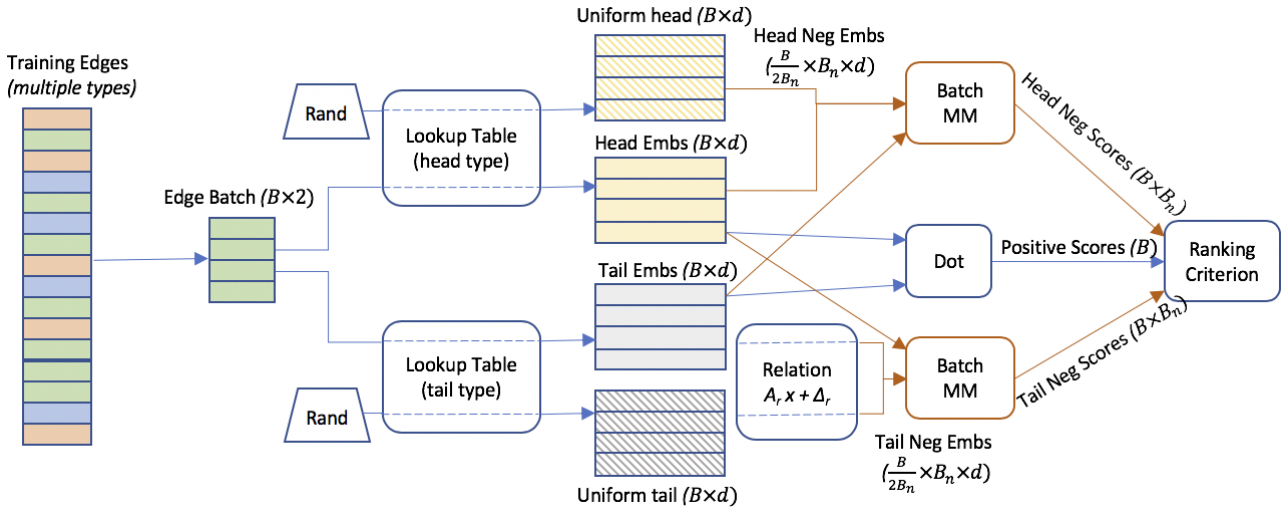

Figure 4. Memory-efficient batched negative sampling. Embeddings are fetched for the B head and tail nodes in a batch of edges, as well as B uniformly-sampled head and tail entities. Each chunk of $B_n/2$ edges is corrupted with all head or tail nodes in its chunk, as well as the corresponding chunk of the uniform embeddings, resulting in B_n negative examples per positive edge. The negative scores are computed via a batch matrix multiply.

learned from the interaction graph as features for training a supervised classifier to predict the the node features.

We show 3 main results. First, PBG is much faster and more scalable than existing methods while achieving comparable performance. Second, the distributed partitioning does not impact the quality of the learned embeddings on large graphs. Third, PBG allows for parallel execution and thus can decrease wallclock training time proportional the number of partitions.

5.1 Experimental Setup

For each dataset, we report the best results from a grid search of learning rates from $0.001 - 0.1$, margins from $0.05 - 0.2$ and negative batch sizes of $100 - 500$, and choose the parameter settings based on the validation split. Results for FB15k are reported on the separate test split.

All experiments are performed on machines with 24 Intel®Xeon®cores (two sockets) and two hyperthreads per core, for a total of 48 virtual cores, and 256 GB of RAM. We use 40 HOGWILD threads for training. For distributed execution, we use a cluster of machines connected via 50Gb/s ethernet. We use the TCP backend for pytorch.distributed which in practice achieves approximately 1 GB/s send/receive bandwidth. For memory usage measurements we report peak resident set size sampled at 0.1 second intervals.

5.2 Freebase Knowledge Graph

Freebase (FB) is a large knowledge graph that contains general facts extracted from Wikipedia, etc. The FB15k dataset consists of a subset of Freebase consisting of 14,951 entities, 1345 relations and 592,213 edges.

5.2.1 FB15K

We compare the performance of PBG embeddings on a link prediction task with existing embedding methods for knowledge graphs. We compare mean reciprocal rank and Hits@k with existing methods for knowledge graph embeddings reported in (Trouillon et al., 2016).[5] Results are shown in in Table 1.

We embed FB15k with a translational relation operator as in (Wu et al., 2017)[6]. Like them, we also find it beneficial to use separate relation embeddings for head negatives and tail negatives (described as 'reciprocal predicates' in (Lacroix et al., 2018)).

[5]We report both raw and *filtered* ranking metrics for FB15k as described in (Bordes et al., 2013). For the filtered metrics, all edges that exist in the training, validation or test sets are removed from the set of candidate corrupted edges for ranking. This avoids artificially poor results due to true edges from the data being ranked above a test edge.

[6]For large interaction graphs with far more relations per parameter, we find that a more expressive relation operator - affine transforms - improves performance. However, small graphs like FB15k suffer from overfitting, so a more domain-specific operator such as translation, circular correlation, or complex multiplication is known to perform better.

PBG performs comparably to results reported for modern knowledge graph embeddings such as transE and HolE(Bordes et al., 2013; Nickel et al., 2016b). Several recent papers have reported very strong results for FB15k (and other small knowledge graphs like WordNet) as a result of extensive hyperparameter tuning, very large embedding sizes, and modifications to the scoring function (Trouillon et al., 2016; Lacroix et al., 2018). Since PBG is not optimized for small knowledge graphs and the purpose of this section is simply to demonstrate that PBG performs comparably to modern method we do not perform extensive tuning to attempt to match these results.

5.2.2 *Full Freebase*

Next, we compare different numbers of partitions and distributed training using the full Freebase dataset[7]. We use all entities and relations that appeared at least 5 times in the full dataset, resulting in a total of 121,216,723 nodes, 25,291 relations and 2,725,070,599 edges. We construct train, validation and test splits of the dataset, which contain 90%, 5%, 5% of the total edges,respectively. The data format we use for the full freebase dataset is the same as in the freebase 15k dataset described in Section 5.2.1.

To investigate the effect of number of partitions, we partition Freebase nodes uniformly into different numbers of partitions and measure model performance, training time, and peak memory usage. We then consider parallel training on different numbers of machines. For each number of machines M, we use $2M$ partitions (which is the minimum number of partitions that allows this level of parallelism. Note that the full model size ($d = 100$) is 48.5 GB.

We train each model for 1 epoch, using the same grid search over hyperparameters for each number of partitions chosen from the same set grid search as FB15k. For the multi-machine evaluation, we use a consistent hyperparameters that had the best performance on single-machine training.

We perform a model evaluation similar to that described in Section 5.2.1. However due to the large number of candidate nodes, for each edge in the eval set we select $10,000$ candidate neative nodes sampled from the set of entities according to their prevalence in the training data to produce negative edges which we use to compute mean reciprocal rank and hits@k[8]. We report these results raw (unfiltered), following prior work on large graphs(Bordes et al., 2013).

Results are reported in Table 2, along with training time and memory usage.

We observe that on a single machine, peak memory usage decrease almost linearly with number of partitions, but training time increases somewhat due to extra time spent on I/O[9]. On multiple machines, the full model is sharded across the machines rather than on disk, so memory usage is higher with 2 machines, but decreases linearly as the number of machines increases. Training time also decreases almost linearly with increasing number of machines, although there is again some overhead for training on multiple machines. This consists of a combination of I/O overhead and incomplete occupancy. The occupancy issue arises because there may not always be an available bucket with non-locked partitions for a machine to work on. Increasing the number of partitions relative to the number of machines will thus increase occupancy, but we don't examine this tradeoff in detail.

Our results show a modest decrease in model quality when using our node partitioning scheme. The main cause of this degradation is the non-i.i.d. sampling of the edges (they're grouped by bucket). That degradation can be ameliorated by running for more than a single epoch, or switching between the buckets more than once per epoch (Gemulla et al., 2011).

Table 2 also shows some decrease in model quality when running on multiple machines due to asynchronously sharing the relation parameters, especially with 8 machines. The asynchronous parameter sharing is more difficult for Freebase than web interaction graphs, because Freebase has over 20MB of relation parameters while web interaction graphs with $10 - 100$ relations have less than 100KB of parameters, so the synchronization is much less frequent for a fixed bandwidth budget.

5.3 LiveJournal

We evaluate PBG performance on the LiveJournal dataset (Backstrom et al., 2006) collected from the blogging site LiveJournal [10], where users can follow others to form a social network. The dataset contains 4,847,571 nodes and 68,993,773 edges. We construct train and test splits of the dataset that contains 75% and 25% of the total edges.

We compare the PBG embedding performance with MILE, which can also scale to large graphs. MILE repeatedly coarsens the graphs into smaller ones and applies traditional embedding methods on coarsened graph at each level as well as a final refinement step to get the embeddings of the original graph. We also show the performance of DeepWalk, which is used as the base embedding method for MILE.

[7] https://developers.google.com/freebase

[8] We sample candidate negative nodes according to their prevalence in the data because the full Freebase dataset has such a long-tailed degree distribution that we find that models can achieve $> 50\%$ hit@1 against $10,000$ uniformly-sampled negatives, which suggests that it is just performing ranking based on the degree distribution.

[9] This I/O overhead is higher on sparser graphs and lower on denser graphs.

[10] https://www.livejournal.com

Method	MRR Raw	MRR Filtered	Hit@10
RESCAL (Nickel et al., 2011)	0.189	0.354	0.587
TransE (Bordes et al., 2013)	0.222	0.463	0.749
HolE (Nickel et al., 2016b)	0.232	0.524	0.739
ComplEx (Trouillon et al., 2016)	0.242	0.692	0.840
R-GCN+ (Schlichtkrull et al., 2018)	0.262	0.696	0.842
StarSpace (Wu et al., 2017)	-	-	0.838
IRN (Shen et al., 2016)	-	-	0.927
Reciprocal ComplEx-N3 (Lacroix et al., 2018)	-	0.86	0.91
PBG	0.265	0.594	0.785

Table 1. Test metrics on FB15k dataset. PBG performs comparably to the tuned performance reported for TransE embeddings in (Nickel et al., 2016b), which is consistent with our use of the TransE model for embedding FB15k. New recent models optimized for knowledge graphs achieve better performance on FB15k.

# Parts	MRR	Hits@10	Time (h)	Mem (GB)
1	0.140	0.257	2.96	59.6
4	0.137	0.248	3.1	30.4
16	0.122	0.228	4.0	6.8

# Machines	# Parts	MRR	Hits@10	Time (h)	Mem (GB)
1	1	0.140	0.259	2.96	59.6
2	4	0.128	0.239	2.30	64.4
8	16	0.087	0.177	0.77	15.0

Table 2. Model evaluation, training time and peak memory usage for embeddings of the full FreeBase knowledge graph. MRR and Hits@10 are evaluated in the *raw* setting. **Left:** Training with different numbers of partitions on a single machine. **Right:** Distributed training on different numbers of machines.

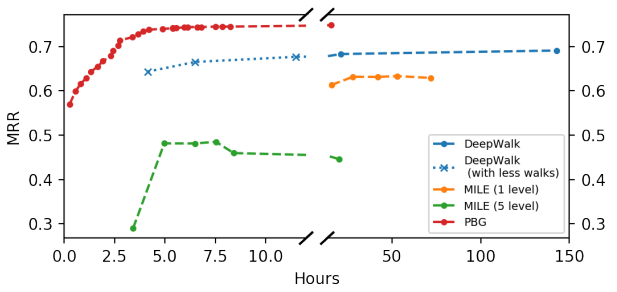

Figure 5. Learning curve for different approaches when training embeddings of the LiveJournal dataset. For each approach, we evaluate the MRR of the learned embeddings at different epochs. DeepWalk takes more than 20 hours to train for one epoch. Therefore, we limit the number of walks during training and show the embedding performance (marked as 'x') to further reduce the computation time.

We perform a model evaluation of link prediction task that is similar to that described in Section 5.2.2. Similarly, we sample 10,000 nodes uniformly from the set of entities to create sampled negative nodes which we use to report the mean of these ranks, the mean reciprocal rank and the hits@10. Results are reported in Table 3 along with memory usage.

To compare the computation time across different approaches, we report the learning curve of evaluation MRR obtained by different approaches during training with respect to time (see Figure 5). We note that PGB is able to achieve good performance much more quickly than existing methods.

5.4 YouTube

To show that PBG embeddings can be used for downstream supervised task, we apply PBG to the Youtube dataset (Tang & Liu, 2009). The dataset contains a social network between users on YouTube[11], as well as the labels of these users that represent categories of groups they subscribed. This social network dataset contains 1,138,499 nodes and 2,990,443 edges.

We compare the performance of PBG embeddings with MILE embeddings and DeepWalk embeddings by applying those embeddings as features to perform a multi-label classification of users. We follow the typical methods (Perozzi et al., 2014; Liang et al., 2018) to evaluate the embedding performance, where we run a 10-fold cross validation by randomly selecting 90% of the labeled data as training data and the rest as testing data. We use the learned embedding as features and train a one-vs-rest logistic regression model to solve the multi-label node classfication problem.

We find that PBG embeddings perform comparably (slightly better) to other methods (see Table 3) with the advantage that the PBG architecture is much easier to scale and distribute.

[11] www.youtube.com

LiveJournal				
Metric	MRR	MR	Hits@10	Memory
DeepWalk*	0.691	234.6	0.842	61.23 GB
MILE (1 level)*	0.629	**174.4**	0.785	60.88 GB
MILE (5 levels)*	0.505	462.8	0.632	22.78 GB
PBG (1 partition)	**0.749**	245.9	**0.857**	20.88 GB

YouTube		
Metric	Micro-F1	Macro-F1
DeepWalk†	45.2%	34.7%
MILE (6 level)†	46.1%	38.5%
MILE (8 levels)†	44.3%	35.3%
PBG (1 partition)	**48.0**%	**40.9**%

Table 3. Test metric on the LiveJournal dataset and the YouTube dataset. **Left:** Link prediction evaluation and peak memory usage for the trained embeddings of the LiveJournal dataset and YouTube dataset. The ranking metrics on the test set are obtained by ranking positive edges among randomly sampled corrupted edges. **Right:** Micro-f1 and Macro-f1 on the user categories prediction task of the YouTube dataset when using learned embeddings as features. * Results obtained running software provided by the original authors. † Results reported in (Liang et al., 2018).

6 Web Application Interaction Graphs

PBG has been used to construct embeddings of large graphs at a large web company. This includes

- A graph of social interactions between users containing over two billion nodes and over one trillion edges (Ching et al., 2015).
- A graph of users interacting with content, containing over two billions nodes and tens of billions of edges per day of interactions.

These graphs are trained in a distributed context, distributed over 10 machines using 20 partitions of the nodes[12].

The content interaction graph is embedded with $d = 100$, resulting in a model size of 950 GB. In this configuration PBG processes 200-400 billion edges per day, allowing multiple epochs to be run.

Good quality embeddings can be helpful for a number of downstream tasks, including identifying fake user accounts, spam, and as features for ranking and retrieval for user and content recommendations.

7 Discussion and Conclusion

In this paper, we proposed Pytorch-BigGraph, an embedding system that scales to graphs with billions of nodes and trillions of edges. PBG supports multi-entity, multi-relation graphs with per-relation configuration such as edge weight and choice of relation operator. To save on memory usage and to allow parallelization PBG performs a block decomposition of the adjacency matrix into N buckets, training on the edges from one bucket at a time.

We evaluated PBG on the FB15k and LiveJournal graphs and show that it matches the performance of existing embedding systems, with a big reduction in training time. We also show results for an embedding of the full FreeBase knowledge graph. We show that partitioning of the Freebase graph reduces memory consumption by 88% without degrading embedding quality, and distributed execution on 8 machines speeds up training by a factor of 4.

We discuss two causes of model degradation in Freebase in Section 5.2.2. We plan to explore solutions to these issues, such as evaluating the accuracy/performance tradoff for different frequencies of switching buckets, as well as periodically shuffling entities between partitions, in future work.

We have presented PBG's performance on the largest publicly available graph datasets that we are aware of. However, the largest benefits of the PBG architecture come from graphs that are $1-2$ orders of magnitude larger than these, where more fine-grained partitioning is necessary and exposes more parallelism with a smaller effect on model quality. We hope that this work helps to motivate the release of larger graph datasets, and an increase in research and reported results on larger graphs.

The model presented in this paper only uses identity features for nodes in the graph. PBG supports graphs in which a subset of entities are sparsely featurized, i.e. bags of embeddings (Wu et al., 2017), or begin with a vector featurization. In particular this allows PBG to embed graphs in which some nodes are text (i.e. bags of word embeddings).

In the results reported here we have explored different parallelization schemes and some types of relation operators. Important future extensions are to consider other components of the PBG system: the loss function (Trouillon et al., 2016), the negative sampling, alternative distance functions (Nickel & Kiela, 2017), or the use of graph convolutional networks for aggregating information about a node's neighborhood (Kipf & Welling, 2016).

[12] Smaller graphs are trained on single machines

8 REFERENCE

REFERENCES

Backstrom, L., Huttenlocher, D., Kleinberg, J., and Lan, X. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 44–54. ACM, 2006.

Bordes, A., Weston, J., Collobert, R., Bengio, Y., et al. Learning structured embeddings of knowledge bases. In *AAAI*, volume 6, pp. 6, 2011.

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.

Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., and Muthukrishnan, S. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

Cook, D. J. and Holder, L. B. *Mining graph data*. John Wiley & Sons, 2006.

Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pp. 1223–1231, USA, 2012. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=2999134.2999271.

Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

Gemulla, R., Nijkamp, E., Haas, P. J., and Sismanis, Y. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pp. 69–77, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0813-7. doi: 10.1145/2020408.2020426. URL http://doi.acm.org/10.1145/2020408.2020426.

Gupta, A., Karypis, G., and Kumar, V. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5): 502–520, 1997.

Hamilton, W., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pp. 1024–1034, 2017a.

Hamilton, W. L., Ying, R., and Leskovec, J. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017b.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Koren, Y., Bell, R., and Volinsky, C. Matrix factorization techniques for recommender systems. *Computer*, (8): 30–37, 2009.

Krompaß, D., Baier, S., and Tresp, V. Type-constrained representation learning in knowledge graphs. In *International Semantic Web Conference*, pp. 640–655. Springer, 2015.

Lacroix, T., Usunier, N., and Obozinski, G. Canonical tensor decomposition for knowledge base completion. *arXiv preprint arXiv:1806.07297*, 2018.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pp. 583–598, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL http://dl.acm.org/citation.cfm?id=2685048.2685095.

Liang, J., Gurukar, S., and Parthasarathy, S. Mile: A multi-level framework for scalable graph embedding. *arXiv preprint arXiv:1802.09612*, 2018.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

Nickel, M. and Kiela, D. Poincaré embeddings for learning hierarchical representations. In *Advances in neural information processing systems*, pp. 6338–6347, 2017.

Nickel, M., Tresp, V., and Kriegel, H.-P. A three-way model for collective learning on multi-relational data. In *ICML*, volume 11, pp. 809–816, 2011.

Nickel, M., Murphy, K., Tresp, V., and Gabrilovich, E. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, Jan 2016a. ISSN 0018-9219. doi: 10.1109/JPROC.2015.2483592.

Nickel, M., Rosasco, L., Poggio, T. A., et al. Holographic embeddings of knowledge graphs. 2016b.

Ordentlich, E., Yang, L., Feng, A., Cnudde, P., Grbovic, M., Djuric, N., Radosavljevic, V., and Owens, G. Network-efficient distributed word2vec training system for large

vocabularies. In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management (CIKM)*, pp. 1139–1148, 2016.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

Perozzi, B., Al-Rfou, R., and Skiena, S. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pp. 701–710, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2956-9. doi: 10.1145/2623330.2623732. URL http://doi.acm.org/10.1145/2623330.2623732.

Recht, B., Re, C., Wright, S., and Niu, F. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pp. 693–701, 2011.

Schlichtkrull, M., Kipf, T. N., Bloem, P., van den Berg, R., Titov, I., and Welling, M. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pp. 593–607. Springer, 2018.

Shazeer, N., Doherty, R., Evans, C., and Waterson, C. Swivel: Improving embeddings by noticing what's missing. *CoRR*, abs/1602.02215, 2016. URL http://arxiv.org/abs/1602.02215.

Shen, Y., Huang, P.-S., Chang, M.-W., and Gao, J. Implicit reasonet: Modeling large-scale structured relationships with shared memory. 2016.

Tang, L. and Liu, H. Scalable learning of collective behavior based on sparse social dimensions. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pp. 1107–1116. ACM, 2009.

Trouillon, T., Welbl, J., Riedel, S., Gaussier, É., and Bouchard, G. Complex embeddings for simple link prediction. In *International Conference on Machine Learning*, pp. 2071–2080, 2016.

Wang, J., Huang, P., Zhao, H., Zhang, Z., Zhao, B., and Lee, D. L. Billion-scale commodity embedding for e-commerce recommendation in alibaba. *arXiv preprint arXiv:1803.02349*, 2018.

Wu, L., Fisch, A., Chopra, S., Adams, K., Bordes, A., and Weston, J. Starspace: Embed all the things! *arXiv preprint arXiv:1709.03856*, 2017.

Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., and Leskovec, J. Graph convolutional neural networks for web-scale recommender systems. *arXiv preprint arXiv:1806.01973*, 2018.

Zitnik, M. and Leskovec, J. Predicting multicellular function through multi-layer tissue networks. *CoRR*, abs/1707.04638, 2017. URL http://arxiv.org/abs/1707.04638.