Scalable Robust Graph Embedding with Spark

Chi Thang Duong
EPFL
thang.duong@epfl.ch

Trung Dung Hoang
EPFL
trung-dung.hoang@epfl.ch

Hongzhi Yin
The University of Queensland
h.yin1@uq.edu.au

Matthias Weidlich
Humboldt-Universität zu Berlin
matthias.weidlich@hu-berlin.de

Quoc Viet Hung Nguyen
Griffith University
quocviethung.nguyen@griffith.edu.au

Karl Aberer
EPFL
karl.aberer@epfl.ch

ABSTRACT

Graph embedding aims at learning a vector-based representation of vertices that incorporates the structure of the graph. This representation then enables inference of graph properties. Existing graph embedding techniques, however, do not scale well to large graphs. While several techniques to scale graph embedding using compute clusters have been proposed, they require continuous communication between the compute nodes and cannot handle node failure. We therefore propose a framework for scalable and robust graph embedding based on the MapReduce model, which can distribute any existing embedding technique. Our method splits a graph into subgraphs to learn their embeddings in isolation and subsequently reconciles the embedding spaces derived for the subgraphs. We realize this idea through a novel distributed graph decomposition algorithm. In addition, we show how to implement our framework in Spark to enable efficient learning of effective embeddings. Experimental results illustrate that our approach scales well, while largely maintaining the embedding quality.

PVLDB Reference Format:
Chi Thang Duong, Trung Dung Hoang, Hongzhi Yin, Matthias Weidlich, Quoc Viet Hung Nguyen, and Karl Aberer. Scalable Robust Graph Embedding with Spark. PVLDB, 15(4): 914-922, 2022.
doi:10.14778/3503585.3503599

1 INTRODUCTION

Graphs represent relations between entities in complex systems, such as social networks or information networks. To enable inference on graphs, a *graph embedding* may be learned. It comprises *vertex embeddings*, each being a vector-based representation of a vertex that incorporates its relations with other vertices [10]. Inference tasks, such as vertex classification and link prediction, can then be based on the vertex embeddings rather than the original graph. Various techniques to learn a graph embedding have been proposed [10, 11, 25]. Yet, aiming at high embedding quality at the expense of computational efficiency, they often do not scale to extremely large graphs with billions of nodes and trillions of edges [16]. Embedding a graph of such size may take weeks, which renders it practically infeasible, and has a large memory footprint. Keeping a graph with two billion nodes in main memory requires 1TB RAM [16], which exceeds the capacity of commodity servers.

Existing techniques for distributed graph embedding require *continuous* communication between the nodes of a compute cluster for model or gradient synchronisation [16, 34]. For instance, in DGL [34], each compute node handles a separate subgraph. Yet, all nodes share the same embedding model, which requires synchronization: Each node needs to send gradient updates learned from its subgraph to *all* other compute nodes. As a result, these synchronous approaches suffer from large communication costs. In addition, they are highly susceptible to communication loss or node failure. Upon failure of a compute node, *all* other nodes need to restart from their latest checkpoint. This leads to longer training times and the need for manual intervention in case of failures.

Against this background, in this paper, we propose a framework for scalable and robust graph embedding that is agnostic to the underlying technique to construct the embeddings. Our idea is to ground the construction of graph embeddings in the MapReduce model. That is, a graph is split into subgraphs, so that an embedding is learned from each subgraph on a separate compute node (map phase). Without a need for synchronisation between the nodes during this computation, *any* specific technique to construct embeddings is improved in terms of scalability and robustness. As learning is done independently and each compute node considers solely a subset of vertices, the results are vectors in different embedding spaces, though. Hence, reconciliation of these spaces is needed to obtain a meaningful graph embedding (reduce phase).

Realising the above vision is challenging, since graph partitioning is an NP-hard problem, which we need to solve in a distributed manner to handle extremely large graphs. In addition, the obtained subgraphs must share vertices, referred to as *landmarks*, to enable reconciliation of embedding spaces. This constraint calls for a new distributed graph decomposition algorithm that carefully chooses the landmarks and considers them in the partitioning process.

In the remainder, we first define the problem of distributed graph embedding and outline our general approach (§2). We then present the details of our framework, making the following contributions:

MapReduce-based graph embedding (§3): Our framework includes a map phase, in which each node embeds a subgraph, and a reduce phase to reconcile the embedding spaces. As such, we learn the map and reduce functions instead of defining them upfront.

Scalable graph decomposition (§4): To facilitate MapReduce-based graph embedding, we propose a scalable graph decomposition algorithm that incorporates landmarks. The algorithm follows the vertex-centric programming model, which enables distributed computation of graph algorithms.

Proceedings of the VLDB Endowment, Vol. 15, No. 4 ISSN 2150-8097.
doi:10.14778/3503585.3503599

Implementation in Spark (§5): We show how to implement our framework in Spark [33], focusing on data locality, communication optimisation, and GPU integration. The choice of Spark is motivated by its computational model, as Spark workers can work independently on each subgraph before merging the results. Also, Spark provides communication and error handling, which helps in training large graphs as the cost of restarting in case of errors during training is extremely high. We also introduce an iterative refinement process to improve the embedding quality.

Comprehensive evaluation experiments with real-world data of billion-scale graphs illustrate the effectiveness and efficiency of our approach (§6). We show that our graph embedding framework is at least 2× faster than existing approaches, reduces communication cost by at least an order of magnitude, and still achieves better embedding quality than existing techniques. We close the paper with a review of related work (§7) and conclusions (§8).

2 PROBLEM AND APPROACH

2.1 Problem statement

Let $\mathcal{G} = (V, E)$ be an undirected graph with vertices V and edges $E \subseteq [V]^2$. A graph embedding technique aims to learn a mapping $f_\Theta : V \rightarrow \mathbb{R}^d$ from vertices V to an embedding in a low-dimensional space ($d \ll |V|$), such that 'similar' vertices are mapped to close vertex embeddings [10].

To learn embeddings for large graphs, we consider a cluster of n compute nodes. Given a graph and an embedding technique, the problem of *distributed graph embedding* is to leverage the n compute nodes for the efficient construction of the graph embedding. Here, efficiency is largely determined by the communication cost between the compute nodes, which shall be minimized.

2.2 Approach

Our approach to distributed graph embedding works in several rounds. In each round, as shown in Figure 1, the graph is decomposed into n subgraphs, so that the construction of embeddings can be distributed among n compute nodes. However, as the vertex embeddings are created independently, they belong to different spaces. To tackle this problem, embeddings from different spaces are reconciled based on vertices that are shared among the subgraphs, called *landmarks*. To obtain embeddings of high quality, the above computation is performed in several rounds, in which the models obtained in the previous round are used for further refinement in the next round.

Our approach can be formulated in the MapReduce model [4]. The construction of embeddings is akin to the map function, since a subgraph is mapped to several vertex embeddings. The reconciliation of embedding spaces denotes a reduce phase, in which a single reconciled embedding is derived. Following this model, our framework can be implemented in state-of-the-art engines for distributed data processing, as demonstrated later with Spark [33].

We note that the decomposition of the original graph into subgraphs is a crucial part of our approach. However, there is an exponential number of possible decompositions of a graph into n partially overlapping subgraphs, and each split has different consequences for the quality of the final embedding. To be able to decompose the graph in a distributed manner and chose suitable landmarks, we design a landmark-aware decomposition algorithm based on the vertex-centric programming model [19]. Our algorithm is based on the Label Propagation Algorithm (LPA) [21], in which we control the condition upon which a vertex may migrate from one partition to another.

3 MAPREDUCE-BASED EMBEDDING

To formulate distributed graph embedding in the MapReduce model, we discuss the map (§3.1) and reduce (§3.2) functions.

3.1 Learned Map Function

In our framework, the map function takes a subgraph $\mathcal{S}$ and returns its vertex embeddings $\boldsymbol{F}$, i.e., it is the function $f_\Theta : V \rightarrow \mathbb{R}^d$ that a graph embedding technique aims to learn. Hence, unlike traditional map functions in MapReduce, our map function is not defined upfront. Rather, we define the *structure* of the function to construct embeddings and rely on a learning framework such as Pytorch [24] to estimate its parameters by minimising a loss function. Then, each mapper learn its own function f_Θ based on the input subgraph and additional information, such as vertex features.

There are two approaches to construct graph embeddings and, hence, two ways to define the structure of the respective function: shallow graph embedding techniques and graph neural networks.

Shallow graph embedding. Here, the function f_Θ is just a mapping from the vertices to the embeddings, i.e., the vertex embeddings are learned directly. Hence, the parameters of function f_Θ are the vertex embeddings. That is, the function is defined as $f_\Theta : u \mapsto \boldsymbol{u}$, where $\{\boldsymbol{u} \mid \forall\, u \in V\} = \Theta$ are the parameters that we need to learn. In other words, the vertex embeddings are the model itself.

Graph neural network (GNN). GNN is a deep embedding model, where the final vertex embeddings are obtained by applying several transformation functions consecutively on the vertex features. Put it differently, function f_Θ is a composition of several transformation functions, such that each parametrised transformation function maps from one vertex embedding to another one. The initial embeddings are given directly by the vertex features.

Learning the parameters. The parameters of the functions to construct embeddings are learned by minimising a loss function. The loss function represents the objective that shall be captured by the vertex embeddings. There are two common types of loss functions: supervised and unsupervised. With a supervised function, vertices are labelled and the embeddings shall be able to predict these labels. In the unsupervised setting, vertices do not carry labels and the vertex embeddings shall capture the structure of the graph.

3.2 Landmark-based Reduce Function

Learned reduce function. The reduce function takes two embeddings $\boldsymbol{F}_1, \boldsymbol{F}_0$ as input and returns a reconciled embedding $\boldsymbol{F}$. Our idea is to reconcile embedding spaces based on landmarks. A landmark will be associated with different embeddings in different embedding spaces, even though it relates to the same entity, i.e., the same vertex in the original graph. Hence, landmarks tell us how to convert an embedding space into another one.

We realize this idea by learning a mapping function $h(\boldsymbol{F}_1)$ that takes a source embedding space as input and returns a mapped

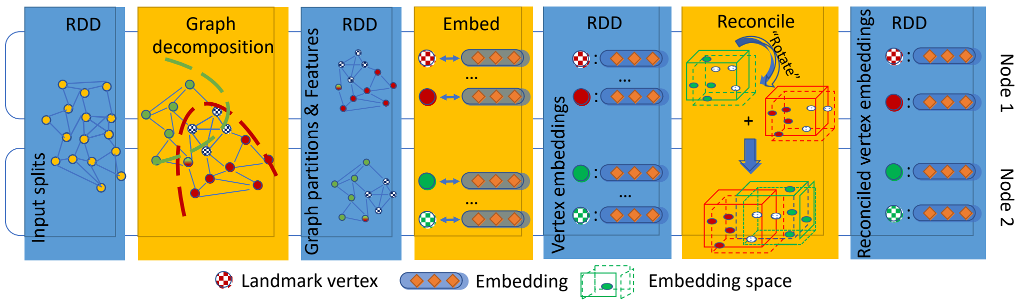

Figure 1: One round of computation in our framework on two compute nodes (yellow: operations; blue: input/output data).

space, such that the embeddings of the landmarks are close in F_0. We call the embedding space F_0 the *anchor space*. This approach is inspired by techniques for network alignment [20]. However, there is an important difference: In network alignment, the vertex correspondences are pre-specified and used as input to train the model. In our setting, the landmarks can be chosen explicitly, which we later exploit with a dedicated selection strategy. The mapping function can be linear or a multilayer perceptron [28]. In any case, our objective is captured by the following loss function:

$$L(h, F_1, F_0, L) = \sum_{v \in L} ||h(\mathbf{z}_{i,v}) - \mathbf{z}_{0,v}||_F \quad (1)$$

where $||.||_F$ is the Frobenius norm, L is the set of landmarks, and $\mathbf{z}_{i,v}$ is the embedding of landmark v in F_1. Since it was shown that a linear function is sufficient to obtain a good mapping [1, 15], we define the mapping function as $h(F_1) = F_1 \times W$ where $W \in \mathbb{R}^{d \times d}$ and d is the embedding dimensionality. The above equation is rewritten in its matrix form, if we denote the embedding matrices of the landmark nodes of F_1 and F_0 as H_1 and H_0:

$$L(H_1, H_0, W) = ||H_1 W - H_0||_F \quad (2)$$

Also, a better mapping is obtained when enforcing orthogonality on W [15]. Under this constraint, the mapping matrix W that minimises Eq. 1 is found using singular value decomposition (SVD). Let $U\Sigma V^T = H_0 H_1^T$ be the SVD of the matrix $H_0 H_1^T$. Then, W is computed as $W = UV^T$. The matrix W can be computed as discussed above since this is its closed form solution. In the general case, it can be found by minimzing the function in Eq. 2.

While we focus on mapping the landmarks, the learned mapping function is applicable to the whole embedding space. Let W_1 be the mapping matrix from H_1 to H_0. Then, the reconciled embedding space of F_1 is $F_1 W_1$. Combined with F_0, we obtain the reconciled embedding space $[F_0, F_1 W_1]$ where $[\ .\ ,\ .\ ]$ is the concatenation operator. As such, the reduce function is given as:

$$r(F_0, F_1) = [F_0, F_1 W_1] \quad (3)$$

Reduction order. To support the parallel execution of reducers, the reduce function needs to be *commutative* and *associative*. In our case, these properties ensure that the order in which we reconcile the embedding spaces does not affect the final embeddings.

First, the reduce function learned as in Eq. 3 is commutative. While the order of reducing would return either $[F_0, F_1 W_1]$ or $[F_0 W_0, F_1]$, both results have the same meaning, as the relative positions of the embeddings are the same, i.e., one space can be obtained from the other under a rotation. The above embedding spaces are similar as $[F_0, F_1 W_1] = [F_0 W_1^{-1}, F_1] = [F_0 W_0, F_1]$ since $W_0 = W_1^{-1}$. The latter is derived from the fact that in Eq. 2, depending on the order of application, we obtain either W_1 or W_0.

However, our reduce function is not guaranteed to be associative. For $r(r(F_0, F_1), F_2)$ to be equal to $r(F_2, r(F_0, F_1))$, $[F_{01}, F_2 W_2]$ needs to be equal to $[F_{01} W_{01}, F_2]$ or $W_{01} = W_2^{-1}$. This is only true, if the space F_2 shares the same landmarks with both F_1 and F_0. Put differently, for the reduce function to be associative, all embedding spaces need to share the same landmarks.

From the above observation, we conclude that subgraphs shall share a common set of landmarks.

4 SCALABLE GRAPH DECOMPOSITION

This section introduces our approach to graph decomposition. We first propose an algorithm based on message-passing (§4.1). We then define two decomposition strategies used by our approach, landmark-aware (§4.2) and complement graph partitioning (Eq. 4.2).

4.1 General Approach

Requirements. While graph decomposition is a well-studied problem, there are several requirements that pertain to our setting:

(1) The decomposition shall be able to handle large graphs and operate in a distributed setting. Centralised algorithms, such as METIS [13], can handle large graphs, but have a large memory footprint and, hence, are not applicable for commodity clusters.
(2) The decomposition shall support constraints on the size of the subgraphs, which may be chosen based on the memory available on compute nodes to optimize resources utilisation and to prevent stragglers. If nodes have the same amount of memory, the subgraph size is $\frac{n-n_l}{k} + n_l$, where n is the graph size, n_l is the landmark subgraph size, and k is the number of nodes.
(3) Subgraphs shall share the same set of connected, important landmarks to support reconciliation of embedding spaces. Here, a high connectivity ensures meaningful landmark embeddings.
(4) Subgraphs shall have little overlap apart from the landmarks, as such boundary edges may be ignored, which could lower the embedding quality.

Vertex-centric computational model. The above requirements suggest to adopt message-passing as a computational model for the efficient and robust realization of distributed graph algorithms [19, 21]. The model supports distribution, as there is no order of local operations as part of a so-called superstep, while all communication happens between these supersteps.

In each superstep, a user-defined function for each vertex is executed [19]. The model permits three types of operations that a vertex can perform. First, a vertex may read messages it received in the previous superstep. Second, a vertex may send messages to other vertices (usually its neighbours) that they will receive in the next superstep. Third, a vertex may change its internal state and modify its outgoing edges if necessary. Based on these three operations, various graph algorithms can be implemented, e.g., Pagerank [19].

Label propagation algorithm. Most algorithms using the vertex-centric computational model are instances of the Label Propagation Algorithm (LPA) [19, 21], illustrated in Alg. 1. LPA first assigns labels to vertices randomly (line 1). Then, it iteratively improves the results by reassigning vertex labels (lines 2-10). A vertex v will take the label l, if it is the most compatible one according to a compatibility function, $comp(v, l)$. Algorithms based on LPA differ in how they measure the compatibility between a vertex and a label. Moreover, each vertex can then choose to 'migrate' from one label to another one based on information obtained from its neighbours or itself from previous iterations (lines 11-12). After the migration, depending on the function *comp*, statistics are derived to support the evaluation of function *comp* (line 14). For instance, if *comp* involves constraints on subgraph sizes, the statistics would include the measured sizes. While both the compatibility scoring and migration are implemented as vertex-centric programs, for readability, we present LPA as an iterative algorithm.

Next, we show how LPA is instantiated in our setting to design an algorithm for landmark-aware graph decomposition.

A two-step approach. A solution to landmark-aware graph decomposition would be an algorithm based on n-way graph partitioning [14]. It would decompose a graph into n subgraphs at the same time, such that all subgraphs satisfy the above constraints. Yet, finding such a decomposition is difficult, as even in the simplest case of balanced graph partitioning, the problem is NP-hard [6].

We therefore propose a heuristic algorithm that works in two steps, each tackling a subset of the constraints. In the first step, we focus on constructing the connected landmark graph of vertices of high importance. In the second step, we aim to construct the complement graphs that satisfy the constraints on subgraph sizes and crossing edges. The algorithm is illustrated in Alg. 2.

We first measure the importance of each vertex in the graph based on a centrality score (line 1). Next, we decompose the graph into the landmark graph and a complement graph using the LPA with the compatibility function introduced later in Eq. 4 (line 2).

After obtaining the landmark graph, we continue to split the complement graph following the same procedure (line 3). We use a different compatibility function for this step, as later introduced in Eq. 5. For both steps, we need to enforce the size constraints on the subgraphs. Hence, in the LPA, the aforementioned statistics (line 14 of Alg. 1) include the subgraph sizes. Finally, after splitting the complement graph into subgraphs, we merge each of them with the landmark graph to obtain the final decomposition (line 4).

Algorithm 1: Label propagation algorithm

```
input  : Graph $\mathcal{G} = (V, E)$; label set $\mathbb{L} = \{l_1, \ldots, l_n\}$; compatibility function $comp$,
         termination condition $\Omega$.
output : $n$ subgraphs $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n$ induced by the vertex labelling.
   // Label initialisation - Vertex program
1  for $v \in V$ do $label(v) \leftarrow init\_label(v)$;
2  while not $\Omega$ do // Label propagation
3      for $v \in V$ do                                          Vertex program
4          $best\_label \leftarrow \{\}$;
5          $best\_score \leftarrow -\infty$;
           // Compute compatibility score
6          for $l \in \mathbb{L}$ do
7              if $comp(v, l) > best\_score$ then
8                  $best\_label(v) \leftarrow l$;
9                  $best\_score \leftarrow comp(v, l)$;
10
11     for $v \in V$ do // Vertex migration                      Vertex program
12         $label(v) \leftarrow migrate(label(v), best\_label(v))$
13
       // Statistics to support comp calculation
14     $compute\_statistics(\mathcal{G}, label)$;
15 return $\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_n$ where $\mathcal{S}_i = \{v \in V \wedge label(v) = l_i\}$;
```

Algorithm 2: Landmark-aware graph decomposition

```
input  : Graph $\mathcal{G} = (V, E)$; number of subgraphs $k$; number of landmarks $m$; maximal
         subgraph sizes $\{n_1, \ldots, n_k\}$.
output : $k$ subgraphs $\mathcal{S}_1, \ldots, \mathcal{S}_k$; landmark graph $\mathcal{L}$.
   // Computing vertex centrality
1  $\delta \leftarrow centrality(G)$;
   // Landmark-Complement graph bi-partition
2  $\mathcal{L}, \overline{S} \leftarrow LPA(\mathcal{G}, \{l_{\mathcal{L}}, l_{\overline{\mathcal{L}}}\}, Eq.\ 4)$;
   // Complement graph partition
3  $\overline{S}_1, \cdots, \overline{S}_k \leftarrow LPA(\overline{S}, \{l_1, \cdots, l_k\}, Eq.\ 5)$;
4  for $i \in [1, k]$ do $\mathcal{S}_i \leftarrow \overline{S}_i + \mathcal{L}$ ;
5  return $\mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{L}$;
```

4.2 Landmark-aware Partitioning

Next, we provide details on the first step of our approach, i.e., the instantiation of LPA to construct a connected landmark graph of a specific size that contains important vertices. This requires us to define a compatibility function that takes into account the graph size, its connectedness, and vertex importance. We first discuss how to incorporate the importance of vertices.

Importance-based compatibility. While there are several ways to measure vertex importance, we focus on the computation of vertex centrality in a distributed manner. This limits our options to either the degree centrality or eigenvector centrality, of which PageRank is a particular instance. Note that some popular centrality measures are not applicable in our context due to the implied computational overhead. For instance, the betweenness measure requires computing all pairs shortest paths, which is intractable for large graphs.

Let δ be the function that measures the importance of each vertex. We define the compatibility between a vertex and a label as: $d(v, l) = 1_{l=0}\left(\frac{\delta(v)}{\delta_m} - 1\right)$ where δ_m is a parameter which signifies the smallest level importance we can tolerate. Here, the landmark graph is assumed to have label 0. The larger a vertex importance $\delta(v)$, the more likely it is compatible with the landmark graph.

Size penalty. Strictly enforcing the size constraint on the landmark graph can lead to slow convergence and instability. In practice, by allowing for a small difference [21], the algorithm can converge faster. To this end, we define a penalty score for each partition as a soft constraint to incorporate in the compatibility function.

Let n_l be the desired size of the partition having label l. We write $C(l) = cn_l$ for the maximum capacity of the partition with label l, where $c > 1$ is a slack parameter. That is, we allow the partition with label l to exceed its size n_l by a factor of c. Let $c(l)$ be the number of vertices of partition l at an iteration. We define the size-based penalty as: $s(l) = \frac{c(l)}{C(l)}$. When the partition size is close to its capacity, the penalty is high. For two-way partitioning such as considered in this step, we need to define two penalty functions based on the landmark graph size m and the complement graph size $|V| - m$. As this formulation is not limited to 2-way partitioning, we can apply the same strategy to n-way partitioning by integrating several size penalties as above to the compatibility function. This is useful for our next step of partitioning the complement graph.

Finally, we can define the compatibility score as follows:

$$comp(v, l) = \sum_{u \in N(v)} 1_{label(u)=l}(\lambda_1 d(v, l) - \lambda_2 s(l)) \tag{4}$$

where $N(v)$ is the set of neighbours of vertex v. Here, the summation condition ensures that the landmark graph is connected as the more landmarks a vertex is connected to, the more compatible it is to the landmark graph. Furthermore, λ_1, λ_2 are hyperparameters to balance the size penalty and importance-based compatibility.

Complement Graph Partitioning. The aforementioned requirements request that subgraphs from the complement graph shall have particular sizes and the splits of the complement graph shall minimise the number of edges between subgraphs. For the former conditions, we rely on the size penalty as defined above. As for the latter, we aim to maximise the edge locality. A vertex is more compatible with a label that is shared the most of its neighbours, captured as: $a(v, l) = \sum_{u \in N(v)} 1_{label(u),l}$ Combining the above functions, we obtain the following compatibility function for complement graph partitioning:

$$comp(v, l) = d(v, l) - s(l). \tag{5}$$

By applying this function as part of LPA, we split the complement graph into non-overlapping partitions. The subgraphs can then be combined with the landmarks to obtain the required subgraphs.

5 IMPLEMENTATION & OPTIMISATION

Having introduced our graph embedding framework, we discuss how it is implemented in Spark [33].

Data storage. Our implementation handles graphs, vertex features, and vertex embeddings, using a Distributed File System (DFS), the main memory, and the local file system (LFS) of a compute node. Spark manages data using Resilient Distributed Datasets (RDDs), multisets of data elements kept in main memory. Data on stored on a DFS and in an RDD can be accessed by all machines, in contrast to data stored on the LFS. Accessing data on the LFS does not incur communication cost. Also, LFS has generally bigger capacity than RDDs, which are limited by the available main memory. Note that DataFrames and RDDs can be used interchangeably in Spark, with largely the same performance.

Storing a graph: Initially, the input graph is stored on the DFS to provide access for all compute nodes. The graph is then split randomly where each random subgraph is stored as an RDD, as hinted at already in Figure 1. The actual graph decomposition can then operate on these RDDs, while the subgraphs obtained by the decomposition algorithm are stored on the LFS of compute nodes. This leverages data locality as a mapper can directly access the partition without communicating with other nodes. As a storage format for the DFS and the LFS, we rely on edge lists, i.e., files in which each line represents a node and its neighbours.

Storing vertex features and embeddings: Vertex features and embeddings are stored similarly, i.e., each line in a file represents a vertex and its features or its embeddings. As using a GNN as an embedding technique requires both vertex features and the subgraph for training, we co-locate them together at a compute node to reduce communication cost. The vertex embeddings obtained after the map phase and each reduce phase are stored as RDDs.

Training with GPU. Our implementation supports the use of GPUs to speed up the training process. To this end, elements of an RDD are piped from Spark to an Automatic Differentiation Framework (ADF) that supports graph learning using Pytorch-Geometric [7]. As the training is based on the graph and the vertex features, we would need to pass these information, stored in RDDs, to an ADF. However, sending large subgraphs would be inefficient, so that we store only the path to a subgraph and its vertex features in an RDD. Then, the ADF reads them directly from the DFS. After training, the results are piped back to Spark and stored as RDDs.

Note that the overhead of combining Spark with an ADF is small. First, all communication happens locally on the same worker instance. Second, there is no inter-process communication between Pytorch and Spark during training. Hence, the overhead is only the initial I/O, where Pytorch reads subgraphs and vertex features, and the final I/O, where Spark reads the output from Pytorch. Since Pytorch reads the subgraphs and features directly from the LFS, the initial I/O overhead is small. Third, there is no preprocessing involved as the subgraphs, vertex features, and embeddings are stored in data formats that are natively supported by Pytorch-Geometric [7]. In our experiments, we have observed the total overhead to be around 30s for a graph of 1M nodes.

Fault tolerance. Spark can recover from node failure by reconstructing the input data from the stored data linage. However, if node failure occurs while the data is being processed, all the processed information is lost. While this problem is inherent to the design of Spark, we are able to alleviate this problem by checkpointing the processed data. At specific epochs of the training process, we save the embedding model to the DFS. When a node restarts computation after a failure, training continues from the last checkpoint by loading the model from the DFS. We adopt checkpointing for the map phase, as it incurs high runtimes.

Lazy Reconciliation. After learning the mapping matrix $\boldsymbol{W}$ in the reduction step (see §3.2), we can reconcile the embedding spaces $\boldsymbol{F}_0$ and $\boldsymbol{F}_1$ immediately based on Eq. 3. The reconciled embedding space can then be stored for further reduction step, if needed. Yet, each reduction step requires inter-node communication to transfer *all* the vertex embeddings from the nodes at which they are stored to the node evaluating the reduce function. However, we note that to learn the mapping matrix $\boldsymbol{W}$, in Eq. 2, only the embeddings of the landmarks are needed. Hence, to reduce communication cost, at each reduction step, we only need to fetch the embeddings of the landmarks for reconciliation. Only afterwards, when the reduction step finishes, we apply the stored mapping matrices to reconcile also the non-landmark vertices. This optimisation reduces the communication cost significantly, with only minor overhead caused by the need to store the mapping matrices.

Iterative Refinement. To further improve the embedding quality, our implementation iteratively refines the constructed embeddings. Our idea is to incorporate the embeddings learned in one round also in the subsequent round. Once a round has completed, we store the learned models in a *model bank* (on the DFS). In the next round, the models from this model bank are used as additional input in the learning process. The intuition is that the vertex embeddings constructed in a round should inherit the results from the previous rounds. Also, to maintain continuity, the models obtained in one round should not be vastly different from those of previous rounds. Depending on the graph embedding model, we propose different ways to leverage model bank to improve the embedding quality.

For *shallow graph embedding models*, the vertex embeddings correspond to the learned model. In the first round, the model bank is initialised with random vertex embeddings. Then, in each round, we initialise the vertex embeddings using those in the model bank. After each round, the model bank is updated with the newly constructed embeddings. Hence, unlike a traditional setup that would adopt random initialisation, our learning process takes advantage of the results from previous rounds.

For *graph neural networks*, the models of previous rounds in the model bank are further refined with different subgraphs. Specifically, the model $f^{(k)}(\mathcal{S}_i, \boldsymbol{F}_i)$ obtained at the k-th round by training on subgraph $\mathcal{S}_i$ and vertex features $\boldsymbol{F}_i$ will be trained on another subgraph $\mathcal{S}_j$ with vertex features $\boldsymbol{F}_j$: $f^{(k+1)} = f^{(k)}(\mathcal{S}_j, \boldsymbol{F}_j)$.

6 EXPERIMENTS

This section reports on an experimental evaluation of our approach. We first outline the setup (§6.1). Then, we evaluate different components of our framework (§6.2) before turning to the end-to-end performance of our approach (§6.3).

6.1 Experimental Setup

Datasets. We rely on five real-world standard benchmark datasets, see Table 1. Flickr and Youtube are social networks with a medium number of nodes but a large number of edges [30]. Arxiv [17] and Papers [12] are two bibliographic networks connecting papers, while Products [12] is a network of Amazon products linked by customer purchases. Vertices of Products and Papers are attributed.

Baselines. We compare against DGL [34] and PBG [16], which are two frameworks for distributed graph embedding. While DGL is a general-purpose approach that can be used for any embedding technique, PBG is a scalable shallow embedding model. Both approaches are similar to our approach as they involve partitioning the original graph to scale out the learning process.

Also, we compare our landmark-aware decomposition against graph partitioning with Spinner [21] and DGL [34]. Spinner is a distributed graph partitioning algorithm based on Pregel [19], which can be seen as an extension of the LPA. DGL uses a centralised approach that first abstracts the graph for partitioning and then refines the coarse-grained partitions to obtain the result.

Measures. To measure the embedding quality, we train a linear classifier using the embeddings as features. We then evaluate the classifier on a test set and measure its *accuracy* to obtain a metric for the embedding quality. Efficiency is measured by the *communication cost*, the data volume transferred per epoch of the training process, the *training time* per epoch which is the total training time divided by the number of epochs, and the *speedup*, the training time per epoch normalised by the number of compute nodes used.

Configuration. Unless stated otherwise, we use the following hyperparameters. We split the graph into 5 equal partitions with a landmark subgraph of size 0.01%. Due to the high cost of these experiments, we do not perform hyperparameter tuning and use the suggested default values. For node2vec, we use 10 walks per node with a walk length of 10, batch size 2000, embedding size 128, and learning rate 0.01. For GraphSAGE, we use 2 layers of GNN with 10 and 5 neighbours, respectively, a hidden size of 128, a dropout after the first layer with probability 0.5, batch size 2000, embedding size 128, and learning rate 0.03. We train these algorithms for 5 epochs and report the test accuracy of the last model. We use an AWS cluster of p2.xlarge instances (4 VCPU, 61GB RAM, 1 VGPU) except for experiments involving PBG. In these cases, we use m5a.4xlarge instances, as PBG cannot leverage a GPU. For experiments with the Papers dataset, we use an m5a.12xlarge cluster (48 VCPU, 192 GB memory) due to DGL's memory requirements.

6.2 Component-wise Evaluation

Effectiveness of Graph Decomposition. We first explore the effectiveness of our landmark-aware graph decomposition. The landmark graph size is set as 0.1% of the original graph and we measure the vertex importance by their degree. The quality of the decomposition is assessed by the average degree of the landmark nodes and the normalised number of edge cuts. A good partition shall have a large average degree and a few edge cuts.

Table 2 shows that our approach outperforms the baselines significantly on both metrics, over all datasets. For instance, on the Arxiv dataset, our approach returns a landmark graph with an average degree twice that of Spinner and 6 times that of DGL. In addition, our approach has a significantly lower number of edge cuts in comparison with the baselines. We also observe that only our technique, which follows a distributed approach, is able to handle billion-scale graphs, such as the Papers dataset.

Effects of Reconciliation. We compare the quality of vertex embeddings obtained with and without reconciliation based on accuracy. The results shown in Figure 4-A confirm that the reconciled embedding space has higher accuracy than the non-reconciled one across all datasets. For instance, on the Arxiv dataset, the accuracy of the non-reconciled embedding space is only 0.35 while after reconciliation, the accuracy is 0.47.

Table 1: Statistics of datasets

	|V|	|E|	#features
Flickr	80,513	5,899,882	n/a
Arxiv	169,343	1,166,243	128
Youtube	495,957	1,936,748	n/a
Products	2,449,029	61,859,140	100
Papers	111,059,956	3,231,371,744	128

Table 2: Effectiveness of graph decomposition

	Average degree			Normalised #edge cuts		
	Spinner	DGL	Ours	Spinner	DGL	Ours
Arxiv	674	211	**1214**	3.89	1.08	**0.52**
Products	2323	213	**3331**	35.23	3.77	**1.92**
Youtube	464	11	**7822**	0.46	0.203	**0.09**
Flickr	2383	292	**2487**	1.595	0.95	**0.73**
Papers	906	N/A	**1784**	9.52	N/A	**11.6**

Effects of landmark selection strategy. Next, we analyse the role of landmark by comparing of two landmark selection strategies: random and our proposed degree-based selection. We also measure the quality by comparing the accuracy of the resulting vertex embeddings. Figure 4-B shows that having important landmarks is key in improving the quality of vertex embeddings. The improvement is consistent across all datasets with the highest is 0.2 on Products.

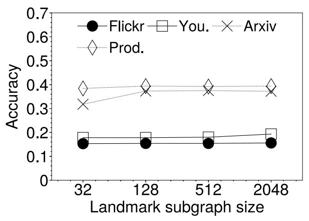

Figure 2: Subgraph size

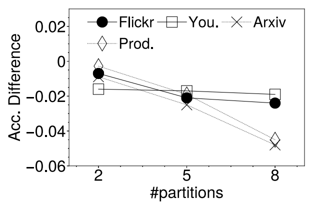

Figure 3: Dist. vs. Single.

Effects of landmark subgraph size. In this experiment, we vary the size of the landmark subgraph from 32 to 2048 to analyse the effect on the embedding quality. Figure 2 shows that the accuracy tends to increase as we increase the subgraph size. However, the rate of increase is small for sizes larger than 128. In general, using more landmarks enables better reconciliation of embedding spaces, which leads to an improvement of the embedding quality. However, increasing the number of landmarks has a diminishing return.

6.3 End-to-end Evaluation

Having evaluated the individual parts of our solution, we turn to its end-to-end performance in comparison to other techniques.

Comparative analysis. Table 3 compares the performance of our approach with state-of-the-art techniques. For the Papers dataset, we rely on m5a.12xlarge instances and evaluate it in a supervised setting. For fair comparison, no GPU is used as PBG cannot use it.

Our approach leads to better or comparable accuracy of the constructed embeddings, while outperforming the baseline techniques in communication cost and training time. This is expected as the only communication in our framework is from the compute nodes to the DFS during the reduction phase. In contrast, both DGL and PBG require continuous communication between compute nodes during training. This communication overhead also implies higher training times per epoch. On datasets with vertex features such as Papers, our approach is better than DGL in terms of accuracy even when the same graph embedding algorithm is used. As our approach splits the graph and performs graph embedding independently, it can be considered as an ensemble of independent models, which is usually better than a single model. As each model may access only the subgraph, but not the full graph, training relies more on vertex features. Hence, our method performs better on graphs with vertex features. For featureless graphs, such as Flickr or Youtube, our approach achieves comparable accuracy with DGL.

Table 3: Comparative analysis

	Time (s)			Accuracy			Communication (GB)		
	PBG	DGL	Ours	PBG	DGL	Ours	PBG	DGL	Ours
Arxiv	76	29	**22**	0.31	0.36	**0.49**	0.04	0.05	**0.006**
Products	649	2081	**361**	0.39	0.55	**0.64**	0.64	4.44	**0.08**
Youtube	312	136	**107**	0.13	**0.21**	0.201	0.6	0.14	**0.04**
Flickr	56	30	**19**	0.15	**0.17**	**0.17**	0.03	0.27	**0.003**
Papers	N/A	3764	**717**	N/A	0.435	**0.478**	N/A	5.324	**0.022**

Distributed vs. Single machine. To understand the trade-off between performance and scalability, we measure the difference in accuracy between our distributed version and a single machine setup. We use the node2vec embedding model (10 walks of length 5, batch size 2000, learning rate 0.01). Using this model, only the Flickr, Arxiv, Youtube and Products datasets fit in main memory, with sizes 4.2GB, 3.9GB, 7.1GB, and 24GB, respectively. Figure 3 shows that the difference in accuracy is very small, less than 0.05 in absolute terms, across all datasets. While the difference increases with the level of parallelism, even with 8 partitions, the drop in accuracy is only 0.048 in comparison to a centralized version.

Scalability. Next, we analyse the scalability of our approach in terms of the speedup, as we increase the number of partitions (and thus compute nodes). Figure 5 shows the speedup as the relative improvement of training time using the setup with two partitions as the reference. The speedup of our approach increases with more partitions and is consistently higher than the one observed for the baseline techniques. For instance, with eight partitions and the Products dataset, our speedup is 3.8, whereas DGL and PBG achieve 0.9 and 1.1, respectively. For small datasets, using more compute nodes actually increases training times for PBG due to the increase in I/O and communication overhead. Figure 5 also shows that our approach maintains stable communication costs as the number of partitions increases. For instance, on the Products dataset, there is an 26% increase in communication, as the number of node increases from two to eight, compared to 56% for DGL and 143% for PBG. For the Papers dataset, we achieve a speedup of 3.9 with 8 partitions and a low communication cost of around 34MB.

Robustness. We measure the recovery cost, i.e., the time required to get back to the same state before the node failure. This time includes the time to load the data to the compute nodes. PBG is excluded in this experiment due to the difficulty in measuring the data loading time, as PBG uses partial data loading. We simulate node failure by terminating the training process of a node at different training rounds. Figure 6 confirms the robustness of our approach,

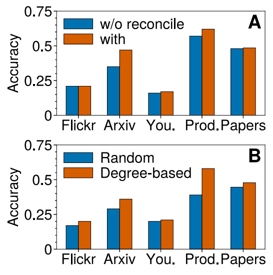

Figure 4: Reconciliation.

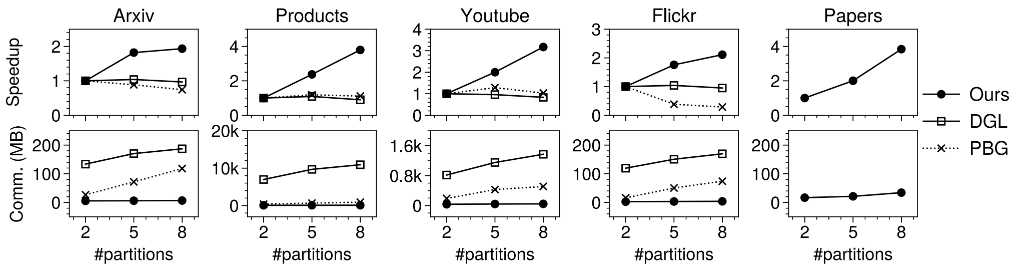

Figure 5: Scalability

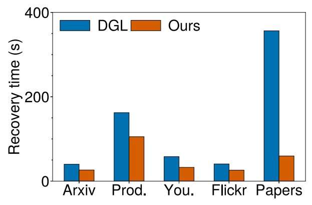

Figure 6: Robustness.

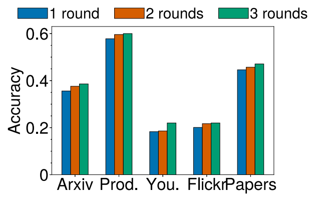

Figure 7: Refinement.

illustrating that recovery costs are lower than those observed for DGL. The reason is that DGL requires restarting the whole training process, since, even though it supports model checkpointing, all compute nodes need to be restarted before training can

Effects of Iterative Refinement. Finally, we analyse the effects of our optimisation based on iterative refinement on the quality of the embeddings. We increase the number of rounds from one to three for each dataset, expecting an increased embedding quality. This is confirmed in Figure 7. However, the improvement is largest initially, reaching a plateau after two rounds. For instance, for the Products dataset, accuracy increases from 0.57 to 0.59, when going from one to two rounds, while another round leads to a minor improvement of 0.01. This phenomenon is consistent on all datasets including the Papers dataset. While we omit the results for training time and communication cost due to space constraint, these measures turned out to increase linearly with the number of rounds.

7 RELATED WORK

Existing technique to learn graph embeddings differ in how they map a vertex into an embedding space, and in the structural properties that shall be retained. To embed a vertex, one may employ shallow or deep encoders [10], or matrix factorisation. Techniques that use shallow encoders, such as [9, 23, 25, 26, 29], consider vertices as words and random walks as sentences, which allows them to use neural word embedding techniques, such as word2vec [22] to construct word/vertex embeddings. Deep encoder approaches [2, 5, 8, 31], such as GraphSAGE [11] or SGN [32], incorporate the neighbourhood of a vertex to generate its embedding. As a consequence, both vertex features and the graph structure may be captured.

Approaches to scale graph embedding techniques to large graphs are classified as *centralised* or *distributed.* Centralised approaches, e.g., SIGN [27], Cluster-GCN [3], SGN [32], MILE [18], rely on 'simpler' models to achieve scalability. For instance, the embedding models of SGN [32] and SIGN [27] consist of several layers of matrix multiplication on the vertex features and graph adjacency matrix. MILE [18] first abstracts the graph into a smaller one to perform the embedding. As centralised approaches use only a single machine, they are inherently limited by the machine's capacity.

Distributed approaches leverage a cluster of compute nodes. They are complementary in the sense that they may rely on centralised techniques for graph embedding on each individual compute node. To date, there are two distributed graph embedding frameworks, PBG [16] and DGL [34]. PBG is tailored to scale shallow embedding techniques that use negative sampling. It has been proposed for compute nodes with shared storage that also communicate during training. DGL aims to scale any GNN embedding technique by synchronising the model across compute nodes during training. Each node is responsible for one partition of the graph. Gradient updates obtained from one compute node are then transferred to all other nodes for global model synchronisation.

As our work, both PBG and DGL strive for scaling embedding techniques by distributing computation across compute nodes. Yet, there are several differences. First, our focus is on a shared-nothing infrastructure, as commonly encountered in compute clusters. Both DGL and PBG require continuous communication between nodes, which slows down the training process. Second, PBG performs random partitioning of the graph. DGL proposes a centralised approach to graph partitioning, which cannot handle extremely large graphs that exceed a single node's capacity. Our approach includes a distributed algorithm for graph decomposition. Third, DGL and PBG are susceptible to node failure, even though the probability of node failure increases with the cluster size. Our approach enables a fault tolerant implementation, due to the reduced inter-node communication and the presented checkpointing approach.

8 CONCLUSION

To achieve scalable and robust graph embedding, we proposed a distributed learning process based on the MapReduce model, which can distribute any existing embedding technique. In essence, in the map phase, we learn vertex embeddings for subgraphs, while the reduce phase reconciles the obtained embedding spaces. For the reconciliation to work, we introduced a distributed graph decomposition algorithm based on a vertex-centric computational model. We also presented an implementation of the approach in Spark. Experiments with several real-world datasets confirm the efficiency, scalability, and robustness of our approach.

ACKNOWLEDGMENTS

This work was supported by ARC Discovery Early Career Researcher Award (Grant No. DE200101465).

REFERENCES

[1] Mikel Artetxe, Gorka Labaka, and Eneko Agirre. 2016. Learning principled bilingual mappings of word embeddings while preserving monolingual invariance. In *ACL*. 2289–2294.

[2] Hongyun Cai, Vincent W Zheng, and Kevin Chang. 2018. A comprehensive survey of graph embedding: problems, techniques and applications. *TKDE* (2018).

[3] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *KDD*. 257–266.

[4] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[5] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *NIPS*. 3837–3845.

[6] Andreas Emil Feldmann and Luca Foschini. 2015. Balanced partitions of trees and applications. *Algorithmica* 71, 2 (2015), 354–376.

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

[8] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *ICML*. 1263–1272.

[9] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *KDD*. 855–864.

[10] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation learning on graphs: Methods and applications. *IEEE Data Engineering Bulletin* (2017).

[11] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NIPS*. 1024–1034.

[12] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *NIPS*.

[13] George Karypis and Vipin Kumar. 1995. METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0. (1995).

[14] George Karypis and Vipin Kumar. 1998. A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN* (1998).

[15] Guillaume Lample, Alexis Conneau, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. 2018. Word translation without parallel data. In *ICLR*.

[16] 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 *MLSys*.

[17] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *TKDD* 1, 1 (2007), 2–es.

[18] Jiongqian Liang, Saket Gurukar, and Srinivasan Parthasarathy. 2018. Mile: A multi-level framework for scalable graph embedding. *arXiv preprint arXiv:1802.09612* (2018).

[19] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD*. 135–146.

[20] Tong Man, Huawei Shen, Shenghua Liu, Xiaolong Jin, and Xueqi Cheng. 2016. Predict Anchor Links across Social Networks via an Embedding Approach. In *IJCAI*. 1823–1829.

[21] Claudio Martella, Dionysios Logothetis, Andreas Loukas, and Georgos Siganos. 2017. Spinner: Scalable graph partitioning in the cloud. In *ICDE*. 1083–1094.

[22] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[23] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. 2016. Asymmetric Transitivity Preserving Graph Embedding. In *KDD*. ACM, 1105–1114. https://doi.org/10.1145/2939672.2939751

[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, 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 *NIPS*. 8024–8035.

[25] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: online learning of social representations. In *KDD*. 701–710.

[26] Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. 2018. Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec. In *WSDM*. 459–467.

[27] Emanuele Rossi, Fabrizio Frasca, Ben Chamberlain, Davide Eynard, Michael Bronstein, and Federico Monti. 2020. Sign: Scalable inception graph neural networks. *arXiv preprint arXiv:2004.11198* (2020).

[28] Dennis W Ruck, Steven K Rogers, Matthew Kabrisky, Mark E Oxley, and Bruce W Suter. 1990. The multilayer perceptron as an approximation to a Bayes optimal discriminant function. *IEEE Transactions on Neural Networks* 1, 4 (1990), 296–298.

[29] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. LINE: Large-scale Information Network Embedding. In *WWW*. 1067–1077.

[30] Lei Tang and Huan Liu. 2009. Relational learning via latent social dimensions. In *KDD*. 817–826.

[31] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[32] Felix Wu, Amauri H. Souza Jr., Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. 2019. Simplifying Graph Convolutional Networks. In *ICML*, Vol. 97. 6861–6871.

[33] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[34] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. *arXiv preprint arXiv:2010.05337* (2020).