PropInit: Scalable Inductive Initialization for Heterogeneous Graph Neural Networks

Soji Adeshina
AWS AI
adesojia@amazon.com

Jian Zhang
AWS AI
jamezhan@amazon.com

Muhyun Kim
AWS ML Solutions Lab
muhyun@amazon.com

Min Chen
Grab
min.chen@grab.com

Rizal Fathony
Grab
rizal.fathony@grab.com

Advitiya Vashisht
Grab
advitiya.vashisht@grab.com

Jia Chen
Grab
jia.chen@grab.com

George Karypis
AWS AI
gkarypis@amazon.com

***Abstract*—Graph Neural Networks (GNNs) require that all nodes have initial representations which are usually derived from the node features. When the node features are absent, GNNs can learn node embeddings with an embedding layer or use pretrained network embeddings for the initial node representations. However, these approaches are limited because i) they cannot be easily extended to initialize new nodes that are added to the graph for inference after training and ii) they are memory intensive and store a fixed representation for every node in the graph.**

In this work, we present PropInit a scalable node representation initialization method for training GNNs and other Graph Machine Learning (ML) models on heterogeneous graphs where some or all node types have no natural features. Unlike existing methods that learn a fixed embedding vector for each node, PropInit learns an inductive function that leverages the metagraph to initialize node representations.

As a result, PropInit is fully inductive and can be applied, without retraining, to new nodes without features that are added to the graph. PropInit also scales to large graphs as it requires only a small fraction of the memory requirements of existing methods. On public benchmark heterogeneous graph datasets, using various GNN models, PropInit achieves comparable or better performance to other competing approaches while needing only 0.01% to 2% of their memory consumption for representing node embeddings. We also demonstrate PropInit's effectiveness on an industry heterogeneous graph dataset for fraud detection and achieve better classification accuracy than learning full embeddings while reducing the embedding memory footprint during training and inference by 99.99%

***Index Terms*—graph neural network, network embedding, scalability, semi-supervised learning, heterogeneous graphs, inductive learning**

I. Introduction

Graph Neural Networks (GNNs) have seen a lot of success in recent years [1], providing sustained benchmark performance improvements, and showing many great results in solving problems in various domains. In industrial settings, GNNs have been applied to tackle problems from recommender systems [2] to fraud detection [3]. In these industrial applications, graphs are heterogeneous since they are typically created from multiple data sources and or contain many interrelated concepts. For example, a heterogeneous graph for fraud detection for an online business contains user nodes, multiple device identity nodes, session identity nodes and so on. Some of the node types like the user nodes will have natural features like demographic information or activity encoding, but it is much more likely that many of the node types will not have natural features. This presents a challenge for training GNN models on these heterogeneous graphs as GNNs require input node representations.

There are two main approaches adopted in industry to tackle this challenge i) using an embedding layer to learn the node embeddings and ii) using a pretrained network embedding method to compute the node embeddings. The first approach attempts to jointly solve the problem of node initialization and the downstream task. It relies on learning the initial embedding vectors as parameters during task training in an end-to-end fashion. The second approach tackles each problem separately. Network embedding techniques [4]–[7] have been extensively studied and are widely used to engineer features for graphs. They learn an embedding vector for each node which contains information about the graph structure by optimizing an objective function to reconstruct the node's neighborhood in the graph. These embeddings can then be used as static features for downstream tasks.

Both of these approaches have the significant drawback of resulting in an overall model that is not inductive. A GNN can generalize to subgraphs induced on nodes that were not seen during training as long as their node features are present. However, when a GNN is coupled with either of the previous node initialization methods it cannot naturally handle previously unseen nodes because their initial node embeddings will be absent. Obtaining the initial embedding for new nodes requires either retraining to recompute the node embeddings which is infeasible for real-time use cases or using an approximation technique to initialize the node representation which can introduce unbounded prediction errors. Thus, in order for GNNs to gain widespread adoption for real-time or near real-time use cases on industry heterogeneous graphs, this issue needs to be addressed.

Another drawback of both of these approach is that they result in memory requirements that grow linearly with the size of the graph. Learning an embedding layer requires maintaining an embedding parameter table proportional to the

TABLE I: Comparison of some node embedding initialization methods for graph machine learning

Method	Transductive/Inductive	Node Embeddings Memory Requirement	Trainable End-to-end
TransE	Transductive	$\mathbf{O}(\lvert\text{nodes}\rvert)$	-
DeepWalk	Transductive	$\mathbf{O}(\lvert\text{nodes}\rvert)$	-
Embedding	Transductive	$\mathbf{O}(\lvert\text{nodes}\rvert)$	✓
PropInit (ours)	Inductive	$\mathbf{O}(c \cdot \lvert\text{node types}\rvert)$	✓

number of nodes. Similarly, pre-training network embeddings requires memory that's a linear function of the number of nodes. As the graph grows, this becomes a scaling bottleneck. See Table I for a summary of these issues across different methods.

To address these challenges left unsolved by existing node embedding initialization methods, we propose a framework called **PropInit**. Our approach deviates from the typical implicit assumption that when there are no node features in a heterogeneous graph, each node should be assigned a fixed node embedding and we propose an inductive function to compute the initial representation for each node.

PropInit learns embeddings vectors for each node type and inductively computes the node embedding for a node using the node types embeddings of its ego graph. Since the node type embeddings are based on the graph schema, they don't need to be recomputed when the graph is updated. Learning node type embeddings also lets nodes of the same type share a representation that encodes their semantic role in the graph. To capture community information for each node, we partition the graph, and nodes of the same type in the same partition share a representation vector. Finally, to capture the local connectivity patterns of the individual nodes, we use graph message passing functions to propagate embeddings across neighboring nodes so that each node's embeddings contains information about its neighbor's embeddings.

We demonstrate the efficacy of our method on four public benchmark heterogeneous graph datasets and one private industry dataset for fraud detection. Our experiments show that we can achieve better node classification performance than learning full embeddings while drastically reducing the memory footprint.

We summarize our main contributions as follows.

C1. We propose a formalism of learning node representations by learning representations for the graph's ontological concepts and propagating those representations in the node's ego graph.

C2. We develop **PropInit** a scalable inductive framework for learning initial node embeddings designed using the above formalism.

C3. We provide experiments on both industrial and academic benchmark datasets to demonstrate the effectiveness of **PropInit**

The rest of the paper proceeds as follows, Section II discusses related works, III introduces some important concepts and notation used in the rest of the work, IV describes **PropInit** in detail, and V presents our experiments and empirical results.

II. Related Work

A. Network Embedding Methods

There has been a long line of work on network embedding methods for feature engineering and unsupervised graph representation in the homogeneous and heterogeneous settings. In the homogeneous setting, methods like DeepWalk [8], LINE [9] and node2vec [10] are all unified by performing some kind of network matrix factorization [11] and all use random-walks on the graph to generate sequences and learn vectors using the skip-gram model [12]. Duong et al [13] finds that DeepWalk outperforms centrality based features and performs on par with real features for node classification on graphs whose structural features correlate with real features, when those features are removed. Heterogeneous network embedding methods like metapath2vec [14] or HIN2Vec [15] extend these approaches to be aware of the heterogeneous schema by generating random walk sequences according to metapaths in the heterogeneous graph. NSHE [16] uses the network schema directly in its sampling method to generate subgraphs used to train the network embeddings. Similarly, many knowledge graph embedding methods have been proposed to learn embeddings for heterogeneous knowledge graphs which take into account some information about the knowledge graph schema by explicitly learning embedding vectors for relation types like TransE [17] and ComplEX [18] or by implicitly modelling relation types in their scoring functions. NARS [19] a heterogeneous extension of SIGN [20], compares TransE with metapath2vec and finds that the downstream GNN models tested achieve their best performance with pre-trained TransE graph embedding features. However, none of these approaches learn inductive functions to generate node embeddings.

B. GNN Embedding

There are broadly two lines of works on GNN embeddings for featureless graphs related to this work. The first are methods that attempt to inductively learn node embeddings. STAR-GCN [21] uses a mask mechanism on node features to inductively learn node embedding for featureless nodes by reconstructing the node embeddings in the first GNN layer but does not handle the case where none of the nodes in the graph have features. IGEL [22] represents local graph structures by learning dense embeddings over a sparse space of structural attributes that appear in the surroundings of nodes in a graph. The second line of related work on GNN node embeddings focus on alleviating the memory requirements. HashGNN [23] jointly trains a GNN encoder for learning node representations, and a hash layer for encoding representations to hash codes

TABLE II: Notation

Symbol	Description
a	A scalar.
$\mathbf{a}$	A vector.
$\mathbf{A}$	A matrix.
$\mathcal{G}$	A graph.
$\mathcal{V}$	The set of nodes in the graph $\mathcal{G}$.
$\mathcal{E}$	The set of edges in the graph $\mathcal{G}$.
$\mathcal{V}_t$	The set of nodes in the graph $\mathcal{G}$ of type t.
$\mathcal{E}_r$	The set of edges in the graph $\mathcal{G}$ of type r.
$\mathbf{X}$	The node feature matrix.
$N^k(v)$	The k-hop neighborhood of a node $v \in \mathcal{V}$.
$\widehat{N^k(v)}$	A randomly subsampled k-hop neighborhood of node v.

to reduce embeddings sizes. PosHashEmb [24] uses hashing and graph partitioning to reduce the memory footprint of embeddings. None of these methods focus on using the graph schema and the partitions to learn inductive node embeddings.

III. Notation and Background

Heterogeneous Graph A heterogeneous graph is one that contains multiple types of nodes and/or multiple types of edges. It can be represented as a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is a set of nodes of T types i.e $\mathcal{V} = \mathcal{V}_1 \cup \ldots \cup \mathcal{V}_T$. For each node type $t \in \{1, \ldots, T\}$, the set of nodes $\mathcal{V}_t = \{v_1^1, ..., v_1^{n_t}\}$ where n_t is the number of nodes of type t. The edge set is defined as $\mathcal{E} = \mathcal{E}_1 \cup \ldots \cup \mathcal{E}_R$ with R the number of edge types. The *metagraph* of $\mathcal{G}$ is a graph $\mathcal{M} = (\mathcal{V}_\mathcal{M}, \mathcal{E}_\mathcal{M})$ that represents the node types and edge types present in $\mathcal{G}$ such that $|\mathcal{V}_\mathcal{M}| = T$ and $|\mathcal{E}_\mathcal{M}| = R$

A *partitioning* of $\mathcal{G}$, denoted by $\mathcal{P} = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_P\}$, divides $\mathcal{V}$ into P disjoint subsets. Each partition $\mathcal{V}_p$ is itself heterogeneous i.e $\mathcal{V}_p = \{\mathcal{V}_{p1} \cup \ldots \cup \mathcal{V}_{pT}\}$.

Node Classification The goal of the task is to predict the node label $\mathbf{y}_{t^*}$ for each node of the target node type t^*. Each node v in $\mathcal{V}$ of type t may be associated with a feature vector $\mathbf{x}_t^{(v)} \in \mathbb{R}^{d_t}$. A node v of target node type t^* it may be associated with a label $\mathbf{y}^{(i)} \in \mathbb{R}^c$ that is the one-hot vector representation of the c classes. We have access to the labels at only a subset of nodes $\{\mathbf{y}_{t^*}^{(v)}\}_{v \in \mathcal{L}}$, with $\mathcal{L} \subset \mathcal{V}$. In this work, we address two settings, where the task in both is to predict the labels of the unlabeled nodes $\{\mathbf{y}_{t^*}^{v}\}_{v \in \mathcal{U}}$, with $\mathcal{U} = \mathcal{V} \setminus \mathcal{L}$.

Transductive learning. In the transductive setting, given a set of node labels $\{\mathbf{y}_{t^*}^{v}\}_{v \in \mathcal{L}}$, the features for nodes $\{\mathbf{X}_{t_{t \in 1,\ldots,T}}\}$, and the full connectivity of the graph $\mathcal{E}$, predict $\{\mathbf{y}_{t^*}^{v}\}_{v \in \mathcal{U}}$. In other words, in the transductive setting, the set of unlabelled nodes $\mathcal{U}$ to predict on are already present in the graph during training.

Inductive learning. Given a set of node labels $\{\mathbf{y}_{t^*}^{v}\}_{v \in \mathcal{L}}$, the features for nodes $\{\mathbf{X}_{t_{t \in 1,\ldots,T}}\}$, learn a function to predict $\{\mathbf{y}_{t^*}^{v}\}_{v \in \mathcal{U}}$ using the graph and features induced from connectivity of the target nodes $v \in \mathcal{U}$. In the inductive setting, the model is required to generalize to new nodes and subgraphs that were not present in the graph during training.

Graph Neural Networks. Graph Neural Networks (GNNs) [25], [26] are neural networks designed to learn representations on graph structured data. Different forms of GNNs like Graph Convolution Networks (GCNs) [27], Graph Attention Networks (GAT) [28], Relational Graph Convolution Networks (RGCN) [29] are unified under the formulation of message passing [30], [31] networks. In this view, the representation for each node in a graph is derived from collecting messages that encode the representation of other nodes in the neighbourhood of that node. Specifically, to compute the representation $h_v^{(l)}$ for a node v at layer l, the message is computed by aggregating the neighbouring nodes representations

$$m_v^{(l)} = \sum(f_1(h_v^{(l-1)}, h_u^{(l-1)}, e_{uv}) : u \in \mathcal{N}(v)) \quad (1)$$

where f_1 is a learnable function and $\sum$ is a reduction. Then the aggregate message $m_v^{(l)}$ is combined with the nodes previous representation to derive the new representation.

$$h_v^{(l)} = f_2^{(l)}(h_v^{(l-1)}, m_v^{(l)}) \quad (2)$$

where f_2 function is also a learnable function.

The details of f_1, f_2 and $\sum$ functions differ based on which GNN implementation is being used. Using GNNs allows for graph tasks like node classification be done in a semi-supervised manner [27]. In this work, we use Graph Neural Networks both in the initialization framework **PropInit** for message propagation and as one of the architectures for the downstream model that consumes the initialized node embeddings for the node classification task.

IV. Method

In this section, we present our **PropInit** framework in detail. Let $\mathcal{G}$ be a heterogeneous graph with nodes $\mathcal{V} = \mathcal{V}_1, \ldots, \mathcal{V}_T$ and edges $\mathcal{E} = \mathcal{E}_1, \ldots, \mathcal{E}_R$ where T is the number of node types in the graph and R is the number of relation types. Without loss of generality, assume there are no natural features for any node type $t \in \{1, \ldots, T\}$. The goal is to create a matrix $\mathbf{H}^0$ of dimension $|\mathcal{V}| \times d$, that contains the input node representations for a downstream GNN or Graph ML model f_ϕ, where $|\mathcal{V}|$ is the number of nodes in the graph.

PropInit generates the node representations $\mathbf{H}^0$ in two stages. The first is the *metagraph embedding* stage which uses learnable embedding layers to obtain embeddings for each node type t in the graph. In the second stage, termed the *embedding propagation* stage, each node in the graph in $\mathcal{G}$ is initialized with its node type embedding and updated by propagating information within its local neighborhood.

The intuition behind our method is that each node's representation consists of two components - one to capture the semantic role of the node and the other to account for the local graph topology around the node. In **PropInit**, the learnable embeddings will learn vector representations that encode the semantic role of each node type in the graph. The embedding propagation component then propagates this information locally across the graph so that each node's representation for the downstream graph task contains both the semantic and local topological information.

In the following subsections, we delve into each component in more detail

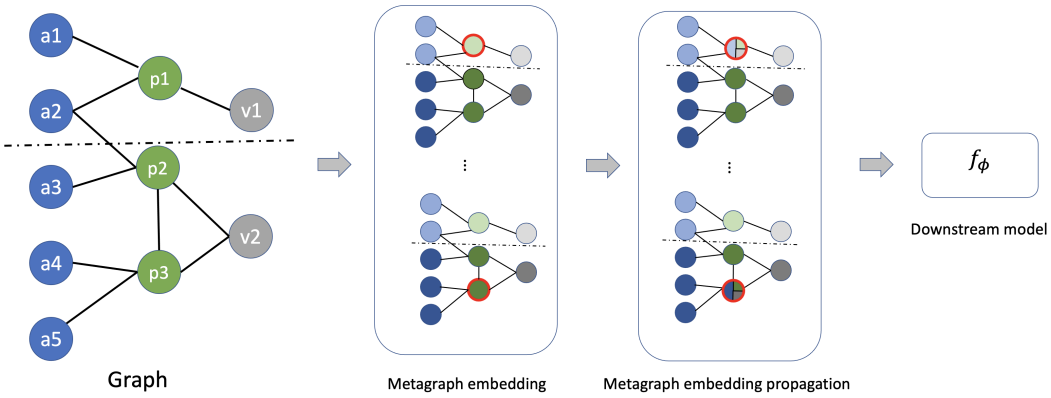


Fig. 1: Depiction of how **PropInit** generates initial node embeddings in two steps. Here, the graph is partitioned into 2 and each node's embeddings, depicted by the color of the node, is initially assigned based on its node type and the partition id. For simplicity, this shows how embeddings are updated for nodes: p1, p3 after one step of embedding propagation. This process actually applies to all nodes in the graph for K steps

A. Metagraph Embeddings

First we initialize a learnable embedding layer which learns an embedding vector of dimensionality d for each node type t. Let $\mathbf{E}$ be the embedding table and $\mathbf{e_t}$ be embedding vector for node type t then we have:

$$\mathbf{e_t} = \mathbf{E}[t, :] \tag{3}$$

We also perform a P-way partition on the graph to group together nodes into distinct communities to allows more flexibility in expressing the semantic role of node types. The number of partitions P is a hyperparameter that can be tuned. For each partition, we initialize a learnable embedding layer which learns an embedding vector of dimensionality d for each node type t. Thus we have P embedding tables of size $T \times d$. Let $\mathbf{E}^P$ the partitions embedding tables represented as a 3D tensor and $e_t^{(p)}$ be the partition specific embedding for node type t then we have:

$$\mathbf{e_t^{(p)}} = \mathbf{E}^P[p, t, :] \tag{4}$$

Each node $v \in \mathcal{V}$ belongs to a node type t and is assigned to a particular partition p. The initial node representation for v is obtained by summing the node type embedding vector with the partition specific node type embedding. In other words, let $\chi_\mathbf{v} \in \mathbf{R}^d$ be the node initialization, then we have

$$\chi_\mathbf{v} = \mathbf{e_t} + \mathbf{e_t^{(p)}} \tag{5}$$

In the special case, where $P = 1$ then all nodes belong to the same partition i.e the whole graph, then all nodes in the graph of type t are initialized to the same vector $\mathbf{e_t}$. We also note that the scheme here can be further generalized to hierarchical partitions by creating additional embedding tables per hierarchical level. i.e

$$\chi_\mathbf{v} = \mathbf{e_t} + \mathbf{e_t^{(p^0)}} + \mathbf{e_t^{(p^1)}} + \dots \tag{6}$$

However, we adopt a flat partitioning approach and the overall size of learnable embeddings is $T \cdot (P+1) \times d << |\mathcal{V}| \times d$

Once, $\chi_\mathbf{v}$ is computed, we proceed to embedding propagation stage which uses message passing to transform $\chi_\mathbf{v}$ into $\mathbf{h_v^0}$, the node features for node v for the downstream model.

B. Metagraph Embedding Propagation

The objective of the embedding propagation is to capture the local connectivity information of a particular node v into its representation $\chi_\mathbf{v}$. We define a message passing function g_ψ that updates $\chi_\mathbf{v}$ by aggregating its neighbors' representations in the graph for k propagation steps. k is a hyperparameter that can be tuned. If we let $\chi \in \mathbf{R}^{n \times d}$ be the matrix output of all node initializations from the previous stage and $\mathbf{H^0} \in \mathbf{R}^{n \times d}$ be the computed node representations for the downstream model then we have:

$$\mathbf{H^0} = g_\psi\left(\mathcal{G}, \chi\right) \tag{7}$$

In our method, g_ψ can be any parametric or non-parametric message passing function. For example, g_ψ can be a k-layer GNN model where the parameters ψ is learnable and updated during training or a function that has no learnable parameters, where messages are simply and aggregated and combined without any transformation steps. In our experiments, we use a 3 layer GAT model for g_ψ, although we also show the method's effectiveness using a non-parametrized g_ψ.

The above computation can also be presented in the ego-centric view for each node. For a node $v \in \mathcal{V}_t$, the initialization from the previous stage $\chi_\mathbf{v}$ is combined with the representations of its neighbors in node v's k-hop ego graph $\{\chi_\mathbf{w} : w \in N^k(v)\}$.

$$\mathbf{h_v^0} = g_\psi\left(N^k(v), \chi_\mathbf{v}\right) \tag{8}$$

When g_ψ is parametrized as a GNN, this is a different GNN than the one that may be used as an encoder in the downstream model.

Algorithm 1 PropInit

Require: graph $\mathcal{G}$, embedding dimension d, number of node types T, number of nodes n, number of partitions P, number embedding propagation steps K, available features $X = \{X_t : t \in \{1 \dots T\}\}$

$Z = metis(\mathcal{G}, P)$ ▷ Partition the Graph into P partitions, Z contains the partition id for each node
for *epoch* **do**
  $\chi \leftarrow zeros(n, d)$
  for $t \leftarrow 0$ to $T-1$ **do**
    for v in $\mathcal{V}_t$ **do**
      $p = Z[v]$ ▷ Get partition id for v
      $\chi[v] \leftarrow Embedding(t) + Embedding(t, p)$
    end for
  end for
  for $k \leftarrow 0$ to $K-1$ **do**
    $\chi \leftarrow g_\psi^{(k)}(\chi, \mathcal{G})$
  end for
  $H^0 \leftarrow \chi$
  $y_{t^*} \leftarrow f_\phi(H^0, X, \mathcal{G})$
  Update learnable parameters of f_ϕ, g_ψ and $Embedding$
end for

C. Training

Now, we present how **PropInit** is jointly trained with the objective from the downstream task in an end-to-end fashion. We focus on node classification tasks where we want to predict $\mathbf{y_{t^*}}$. Let f_ϕ be the node classification model, then we have:

$$\hat{\mathbf{y}}_{\mathbf{t}*} = f_\phi\left(\mathbf{H}^0, \mathbf{X}, \mathcal{G}\right) \tag{9}$$

where $\mathbf{H}^0$ is the node initialization from equation 7, and $\mathbf{X}$ are the available node features. f_ϕ has 3 main components - i) a *feature module* to combine the node initializations with any natural features, ii) an *encoder module* to perform (graph) representation learning and iii) a *decoder module* to predict the label with the representation.

For each node type, t, if natural features are present, we concatenate the features $\mathbf{X}$ with the **PropInit** embeddings $\mathbf{H}^0$ and then pass this through a dedicated one layer MLP for that node type. We primarily use a RGCN for the encoder module in our experiments, although we show that using an MLP encoder is still effective, similar to how MLP encoders can be used on pretrained network embeddings. The decoder module is a one layer MLP whose output dimension is the number of class labels.

Combining equations 7 and 9 we have:

$$\hat{\mathbf{y}}_{\mathbf{t}*} = f_\phi\left(g_\psi\left(\mathcal{G}, \chi\right), \mathbf{X}, \mathcal{G}\right) \tag{10}$$

We can compute the classification loss by comparing $\hat{\mathbf{y}}_{\mathbf{t}*}$ with the node labels $\mathbf{y_{t^*}}$ and use that to directly optimize the parameters of f_ϕ, g_ψ, $\mathbf{E}$ and $\mathbf{E^P}$.

Algorithm 1 provides an overview of the method as described in this section.

D. Stochastic Implementation

In order to train GNNs on large scale graphs, minibatch training with graph sampling is required. Usually, at each training step a subgraph is generated for sampled training node targets.

When using Stochastic **PropInit** in the minibatch setting, at each iteration, only the embeddings of the nodes present in the minibatch subgraph need to be initialized. In the embedding propagation stage, the sampled subgraph for the downstream model can be re-used. In our implementations, we use simple neighbor sampling, starting from the target nodes, to construct a sequence of subgraphs for each layer of the downstream GNN encoder. Then, we use the subgraph for the first layer of the GNN encoder, which contains all the nodes needed to compute the downstream GNN representation, as the subgraph for k rounds of embedding propagation.

E. Memory Usage Analysis

A major advantage of **PropInit** over traditional network embedding techniques its reduced memory footprint during training and inference. In this section, we analyze the memory requirements of **PropInit** and compare to other competing methods. To simplify, we assume that there are no natural features on any of the nodes.

At training time, using Stochastic **PropInit**, and at inference we don't need to materialize the embeddings for all the nodes in the graph therefore the memory requirement is

$$O\left(dT \cdot (P+1) + B(K \cdot Mem(g_\psi) + L \cdot Mem(f_\phi)\right) \tag{11}$$

where T is the number of node types P is the number of partitions, d is the embedding dimension size, B is the total number of nodes in the minibatch subgraph, K is the number of **PropInit** propagation steps, $Mem(g_\psi)$ is the memory requirement per layer of g_ψ the choice of propagation method, L is the number of layers of the downstream model f_ψ and $Mem(f_\phi)$ is the memory requirement per layer of f_ϕ. It is important to note that there is no term $|\mathcal{V}|$ unlike using pretrained node embeddings or learnable node embeddings which incurs

$$O\left(d|\mathcal{V}| + B(L \cdot Mem(f_\phi)\right) \tag{12}$$

Therefore, even as the graph grows and $|\mathcal{V}|$ increases, the memory requirement for **PropInit** only grows $\tilde{P} \times T$ where T is fixed by the graph schema, and P grows very slowly compared to $|\mathcal{V}|$.

V. Experiments

A. Data

We perform experiments on four open-source heterogeneous datasets and a proprietary dataset. For the public datasets, we use the node classification datasets from the Heterogeneous Graph Benchmark (HGB) [32] curated from widely-recognized medium-scale datasets which include DBLP, ACM, IMDB, and freebase. For DBLP, ACM and freebase, the task is

TABLE III: Dataset Characteristics

Name	Nodes	NodeTypes	Edges	Edge Types	Target	Classes
ACM	19,942	4	547,872	8	paper	4
IMDB	21,420	4	86,642	6	movie	5
DBLP	26,128	4	239,566	6	author	4
freebase	180,098	8	1,057,688	36	book	7
FraudDataset	96,632,621	17	325,887,168	39	user	2

multi-class node classification and for IMDB, the task is multi-label node classification. We retain the data split configuration of 24% for training, 6% for validation and 70% used in HGB. We do not use the node features in the datasets.

The proprietary dataset, FraudDataset, is a graph sampled from 2 weeks of user activity in an online service marketplace. The users are labelled as fraudulent or non-fraudulent and the task is to predict these labels in an inductive setting. The training data contains labelled users from the first 10 days of the interval and the validation and test data are from the penultimate and final day of the interval respectively. We run experiments on two variants of the FraudDataset - one where there are no node features on the graph, and another where where 3 of the 17 node types have natural features.

Table III provides a summary of the dataset statistics.

B. Baselines

We compare **PropInit** to other baselines for initializing heterogeneous node representations for node classification. The first baseline is using a Learnable Embedding layer that learns an embedding vector for each node and is trained jointly with the downstream model. Another baseline is TransE [33]. We use TransE as the representative of the set of KG techniques that generate embeddings which are ontology aware by modeling the relation types in the graph. For experiments, we use the TransE implementation provided by DGL-KE [34]. The final baseline is DeepWalk [8]. We use DeepWalk as the baseline representative of self supervised network embeddings methods. We use the DGL [35] implementation of DeepWalk for our experiments. Both TransE and DeepWalk are first run on the graph to generate the embeddings after which the embeddings are then used as node features for the downstream model.

C. Implementation details

The version of **PropInit** that we use in all the baseline comparison experiments sets $P = 4$, $K = 3$ and uses GAT [28] to parametrize g_ψ during the embedding propagation stage. We use DGL [35] with the PyTorch [36] framework to implement the GNN models and METIS [37] for partitioning the graph. We use **PropInit** as described in Algorithm 1 for the the ACM and IMDB datasets and we use the stochastic variant of **PropInit** for DBLP, freebase and FraudDataset.

For all experiments, we use RGCN [29] as the downstream GNN model for node classification. For the experiments on the public open-source datasets, we use the hyperparameters reported in Heterogeneous Graph Benchmark (HGB) [32]. For the FraudDataset, we set $d = 8$ and $L = 3$. All models are optimized using the Pytorch Adam Optimizer with a learning rate of 0.001. We do not perform any hyperparameter tuning but we select the model that performs the best on the validation dataset at any point during training as the model to evaluate on the test dataset. We run each experiment five times and report the average metric and standard error.

D. Results

Table VI shows that **PropInit** is effective across the board. **PropInit** outperforms the baselines on most datasets. For IMDB, our method is within 6% points of the top baseline performance - Node Embedding, while using less than 3% of the parameters required for trainable node embeddings. For the DBLP dataset which has no features, the classification task requires the test node embeddings generalize and only **PropInit** and DeepWalk generates node initializations that perform well in this setting.

We also compare **PropInit** and the baselines above on two variants of the proprietary FraudDataset described earlier. In the first variant, the node types which have node features are initialized by concatenating the node embedding from **PropInit** or the baseline and the natural features of the node. We then project the resulting vector for each node type using an MLP so all nodes have have uniform dimensionality. The results for these sets of experiments are shown in Table V. **PropInit** performs better than the baseline approaches on all performance metrics in this setting of the FraudDataset.

In the second setting, we do not use any features on any of the node types. The embedding from **PropInit** or the baseline is used as the node initialization for a downstream RGCN model. The results for those experiments are also shown in Table V. Here again, **PropInit** has the best F1 score. In particular, **PropInit** generalizes better than learnable node embeddings which actually achieve perfect performance on the training set but drops significantly by 20% on the test set.

E. Encoder Models

To evaluate the robustness of our framework to different downstream GNN and Graph ML encoders, we perform some experiments using different GNNs and ML models for f_ϕ for the downstream node classification tasks. We implement Relation Graph Convolutional Network (RGCN), Graph Attention Network (GAT) and a Multi Layer Perceptron (MLP) as the different variants of f_ϕ. The results which can be seen in Table VI show **PropInit** and an MLP node classification model can match learnable node embeddings with a GNN node classification model.

TABLE IV: Node Classification results on open-source heterogeneous graph datasets

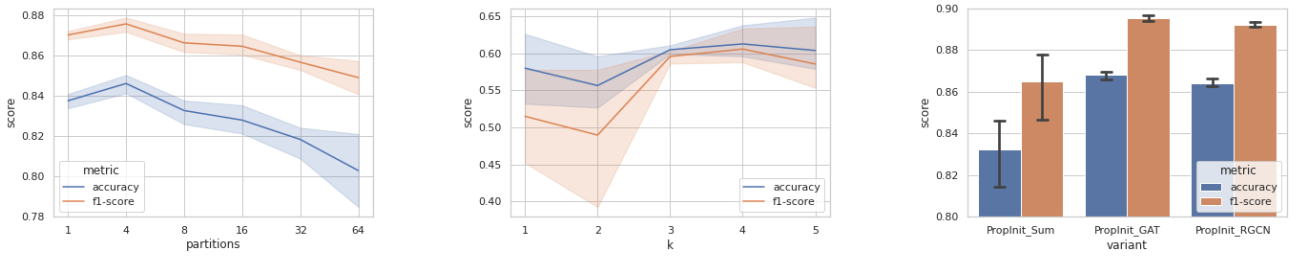
Dataset	ACM		IMDB		DBLP		Freebase	
Method	Micro F1	Macro F1	Micro F1	Macro F1	Micro F1	Macro F1	Micro F1	Macro F1
TransE	52.07 ± 0.80	42.29 ± 3.21	49.30 ± 0.78	40.00 ± 1.80	54.30 ± 6.80	60.70 ± 9.60	56.66 ± 0.50	24.39 ± 1.11
DeepWalk	54.49 ± 1.22	50.93 ± 2.19	56.20 ± 0.58	50.20 ± 0.82	94.01 ± 0.48	93.70 ± 0.53	57.00 ± 1.21	25.90 ± 0.79
Embedding	53.24 ± 0.69	43.38 ± 3.18	**59.20 ± 0.73**	**54.30 ± 0.90**	45.30 ± 0.85	45.90 ± 1.41	**58.80 ± 0.22**	26.30 ± 0.66
PropInit	**60.48 ± 0.35**	**59.57 ± 0.52**	53.29 ± 1.20	48.69 ± 1.14	**94.78 ± 0.28**	**94.54 ± 0.26**	57.99 ± 0.56	**28.15 ± 0.99**

TABLE V: Node Classification results on FraudDataset

Variant	With Node Features				Without Node Features			
Method	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1
TransE	85.61 ± 0.13	85.35 ± 0.22	92.18 ± 0.26	88.63 ± 0.10	83.40 ± 0.14	83.37 ± 0.14	90.90 ± 0.19	86.90 ± 0.11
DeepWalk	86.50 ± 0.11	86.43 ± 0.23	92.33 ± 0.14	89.27 ± 0.06	85.20 ± 0.12	84.80 ± 0.19	**92.30 ± 0.12**	88.40 ± 0.08
Embedding	82.51 ± 0.28	82.13 ± 0.24	91.09 ± 0.25	86.37 ± 0.21	79.86 ± 0.38	79.97 ± 0.29	89.27 ± 0.34	84.36 ± 0.29
PropInit	**86.79 ± 0.11**	**86.42 ± 0.17**	**92.89 ± 0.12**	**89.54 ± 0.08**	**86.04 ± 0.25**	**85.85 ± 0.17**	92.26 ± 0.21	**88.94 ± 0.22**

TABLE VI: Node Classification results on FraudDataset using **PropInit** with different downstream Encoder Models

Node Classification Encoder Model	Accuracy	Precision	Recall	F1
MLP	80.74 ± 0.23	83.60 ± 0.48	85.06 ± 0.46	84.32 ± 0.16
GAT	83.66 ± 0.11	84.12 ± 0.30	90.17 ± 0.44	87.04 ± 0.10
RGCN	86.79 ± 0.11	86.42 ± 0.17	92.89±0.12	89.54 ± 0.08



(a) Effect of number of graph partitions (b) Effect of propagation steps K. (c) Comparing **PropInit** functions g_ψ

Fig. 2: Effect of various components of **PropInit**

F. Effect of number of graph partitions

To study the effect of the number of partitions we run experiments using different total number of partitions P. The results shown in Figure 2a demonstrate that there is a tradeoff between the expressivity of **PropInit** afforded by more granular partitions and the generalization performance of the learned node embeddings. Setting 4 as the number of partitions balances this tradeoff for the FraudDataset.

G. Effect of number of embedding propagation steps

We also explore how **PropInit**'s performance is affected by the radius of local graph topology used when computing a node's initial embedding. We show the results in Figure 2b. For small k, the embedding of a node may not contain enough information to sufficiently distinguish the node, thus the performance of the downstream model is impacted. For large k, the embedding of the node pulls information from distant nodes and hampers downstream performance -a consequence of the well known oversmoothing problem in GNNs. Setting $k = 3$ in our experiments manages the trade-off between these two extremes.

H. Effect of message passing parametrization for **PropInit**

A major component of **PropInit** is the choice of model for g_ψ. To analyze the effect of choice of g_ψ for the embedding propagation, we compare 3 different variants of **PropInit** in Figure 2c. The figure shows **PropInit**_GAT outperforms the other variants. However, even in the case where **PropInit** has no trainable message passing parameters in **PropInit**_SUM, it outperforms using learnable node embeddings.

VI. Conclusion

In this work, we study the problem of node classification on heterogeneous graphs using GNNs and we focus on the case when there are no available node features on some or all node types in the graph. We summarize the challenges with existing approaches as i) being difficult to extend to the inductive setting and ii) having memory requirements for training and inference that grows linearly with the graph size.

We propose **PropInit** a scalable, inductive node representation learning framework which uses node type embeddings, graph partitioning and embedding propagation to initialize the node embedding for a node for a downstream GNN. We show that our method is effective while requiring significantly less memory than competing baselines. Specifically, our method **PropInit**, achieves similar or better classification performance than all baselines on 3 of the 4 public benchmark datasets and uses less than 1% of the parameters required to learn full node embeddings. On an industrial scale dataset, **PropInit** outperforms all the baselines and while using less than 0.01% of the parameters required for full node embeddings.

REFERENCES

[1] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[2] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018, pp. 974–983.

[3] Z. Liu, C. Chen, X. Yang, J. Zhou, X. Li, and L. Song, "Heterogeneous graph neural networks for malicious account detection," in *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, 2018, pp. 2077–2085.

[4] H. Cai, V. W. Zheng, and K. C.-C. Chang, "A comprehensive survey of graph embedding: Problems, techniques, and applications," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 9, pp. 1616–1637, 2018.

[5] P. Cui, X. Wang, J. Pei, and W. Zhu, "A survey on network embedding," *IEEE transactions on knowledge and data engineering*, vol. 31, no. 5, pp. 833–852, 2018.

[6] P. Goyal and E. Ferrara, "Graph embedding techniques, applications, and performance: A survey," *Knowledge-Based Systems*, vol. 151, pp. 78–94, 2018.

[7] C. Yang, Y. Xiao, Y. Zhang, Y. Sun, and J. Han, "Heterogeneous network representation learning: A unified framework with survey and benchmark," *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[8] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 701–710.

[9] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *Proceedings of the 24th international conference on world wide web*, 2015, pp. 1067–1077.

[10] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 855–864.

[11] J. Qiu, Y. Dong, H. Ma, J. Li, K. Wang, and J. Tang, "Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec," in *Proceedings of the eleventh ACM international conference on web search and data mining*, 2018, pp. 459–467.

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

[13] C. T. Duong, T. D. Hoang, H. T. H. Dang, Q. V. H. Nguyen, and K. Aberer, "On node features for graph neural networks," *arXiv preprint arXiv:1911.08795*, 2019.

[14] Y. Dong, N. V. Chawla, and A. Swami, "metapath2vec: Scalable representation learning for heterogeneous networks," in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 2017, pp. 135–144.

[15] T.-y. Fu, W.-C. Lee, and Z. Lei, "Hin2vec: Explore meta-paths in heterogeneous information networks for representation learning," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, 2017, pp. 1797–1806.

[16] J. Zhao, X. Wang, C. Shi, Z. Liu, and Y. Ye, "Network schema preserving heterogeneous information network embedding," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2020.

[17] B. Yang, W.-t. Yih, X. He, J. Gao, and L. Deng, "Embedding entities and relations for learning and inference in knowledge bases," *arXiv preprint arXiv:1412.6575*, 2014.

[18] T. Trouillon, J. Welbl, S. Riedel, É. Gaussier, and G. Bouchard, "Complex embeddings for simple link prediction," in *International conference on machine learning*. PMLR, 2016, pp. 2071–2080.

[19] L. Yu, J. Shen, J. Li, and A. Lerer, "Scalable graph neural networks for heterogeneous graphs," *arXiv preprint arXiv:2011.09679*, 2020.

[20] F. Frasca, E. Rossi, D. Eynard, B. Chamberlain, M. Bronstein, and F. Monti, "Sign: Scalable inception graph neural networks," *arXiv preprint arXiv:2004.11198*, 2020.

[21] J. Zhang, X. Shi, S. Zhao, and I. King, "Star-gcn: Stacked and reconstructed graph convolutional networks for recommender systems," *arXiv preprint arXiv:1905.13129*, 2019.

[22] N. Alvarez-Gonzalez, A. Kaltenbrunner, and V. Gómez, "Inductive graph embeddings through locality encodings," *arXiv preprint arXiv:2009.12585*, 2020.

[23] Q. Tan, N. Liu, X. Zhao, H. Yang, J. Zhou, and X. Hu, "Learning to hash with graph neural networks for recommender systems," in *Proceedings of The Web Conference 2020*, 2020, pp. 1988–1998.

[24] M. Kalantzi and G. Karypis, "Position-based hash embeddings for scaling graph neural networks," in *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, 2021, pp. 779–789.

[25] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 2. IEEE, 2005, pp. 729–734.

[26] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2008.

[27] N. K. Thomas and M. Welling, "Semi-supervised classification with graph convolutional networks. int. conf. on learn," *Representations*, vol. 2, 2017.

[28] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[29] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European Semantic Web Conference*. Springer, 2018, pp. 593–607.

[30] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," *arXiv preprint arXiv:1704.01212*, 2017.

[31] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[32] Q. Lv, M. Ding, Q. Liu, Y. Chen, W. Feng, S. He, C. Zhou, J. Jiang, Y. Dong, and J. Tang, "Are we really making much progress? revisiting, benchmarking and refining heterogeneous graph neural networks," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 1150–1160.

[33] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," *Advances in neural information processing systems*, vol. 26, 2013.

[34] D. Zheng, X. Song, C. Ma, Z. Tan, Z. Ye, J. Dong, H. Xiong, Z. Zhang, and G. Karypis, "Dgl-ke: Training knowledge graph embeddings at scale," in *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2020, pp. 739–748.

[35] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma *et al.*, "Deep graph library: Towards efficient and scalable deep learning on graphs." 2019.

[36] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *arXiv preprint arXiv:1912.01703*, 2019.

[37] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," 1997.