Learning Steady-States of Iterative Algorithms over Graphs

Hanjun Dai [1] **Zornitsa Kozareva** [2] **Bo Dai** [1] **Alexander J. Smola** [2] **Le Song** [1 3]

Abstract

Many graph analytics problems can be solved via iterative algorithms where the solutions are often characterized by a set of steady-state conditions. Different algorithms respect to different set of fixed point constraints, so instead of using these traditional algorithms, can we learn an algorithm which can obtain the same steady-state solutions automatically from examples, in an effective and scalable way? How to represent the meta learner for such algorithm and how to carry out the learning? In this paper, we propose an embedding representation for iterative algorithms over graphs, and design a learning method which alternates between updating the embeddings and projecting them onto the steady-state constraints. We demonstrate the effectiveness of our framework using a few commonly used graph algorithms, and show that in some cases, the learned algorithm can handle graphs with more than 100,000,000 nodes in a single machine.

1. Introduction

Graphs and networks arise in various real-world applications and machine learning problems, such as social network analysis (Hamilton et al., 2017b), molecule screening (Hachmann et al., 2011; Duvenaud et al., 2015; Lei et al., 2017) and knowledge base reasoning (Trivedi et al., 2017). Many graph analytics problems can be solved via iterative algorithms according to the graph structure, and the solutions of the algorithms are often characterized by a set of steady-state conditions. For instance, the PageRank (Page et al., 1999) score of a node in a graph can be computed iteratively by averaging the scores of its neighbors, until the node score and this neighbor averaging are approximately equal. Mean field inference for the posterior distribution of a variable in a graphical model can be updated iteratively by aggregating the messages from its neighbors until the posterior is approximately equal to the results of the aggregation operator. More generally, the intermediate representation h_v for each node v in the node set $\mathcal{V}$ is updated iteratively according to an operator $\mathcal{T}$ as

$$h_v^{(t+1)} \leftarrow \mathcal{T}\Big(\{h_u^{(t)}\}_{u\in\mathcal{N}(v)}\Big),\ \forall t \geqslant 1, \text{ and}$$
$$h_v^{(0)} \leftarrow \text{constant},\ \forall v \in \mathcal{V} \tag{1}$$

until the steady-state conditions are met

$$h_v^* = \mathcal{T}\big(\{h_u^*\}_{u\in\mathcal{N}(v)}\big),\ \forall v \in \mathcal{V}. \tag{2}$$

Variants of graph neural network (GNN) (Scarselli et al., 2009), like GCN (Kipf & Welling, 2016), neural message passing network (Gilmer et al., 2017), GATs (Veličković et al., 2017) *etc.*, perform fixed T rounds of updates to Eq (1) without respecting the steady state. Thus for learning algorithms like PageRank or mean field inference, a large T is required. In such case, both the computational cost and gradient updates will become problematic. Also note that due to the batch-update nature of GNN family models, multiple rounds of update over all nodes are needed. These two limitations make them not scalable and effective enough, regarding the computational cost and convergence.

In this paper, instead of designing algorithms for each individual graph problem, we take a different perspective, and ask the question:

> Can we design a learning framework for a diverse range of graph problems that learns the algorithm over large graphs achieving the steady-state solutions efficiently and effectively?

Furthermore, how to represent the meta learner for such algorithm and how to carry out the learning of these algorithms? In this paper we propose a stochastic learning framework of algorithm design based on the idea of embedding the intermediate representation of an iterative algorithm over graphs into vector spaces, and then learn such algorithms using example outputs from the desired algorithms to be learned.

More specifically, in our framework, each node in the graph will maintain an embedding vector, and these embedding vectors will be updated using a parameterized operator $\mathcal{T}_\theta$ where the parameters θ will be learned. Furthermore, following each embedding update step, the embedding will also be projected towards the steady state constraint space, gradually enforcing the steady-state conditions. As illustrated in Figure 1, both of the two steps are stochastic,

[1]Georgia Institute of Technology [2]Amazon [3]Ant Financial. Correspondence to: Hanjun Dai <hanjundai@gatech.edu>.

Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018. Copyright 2018 by the author(s).

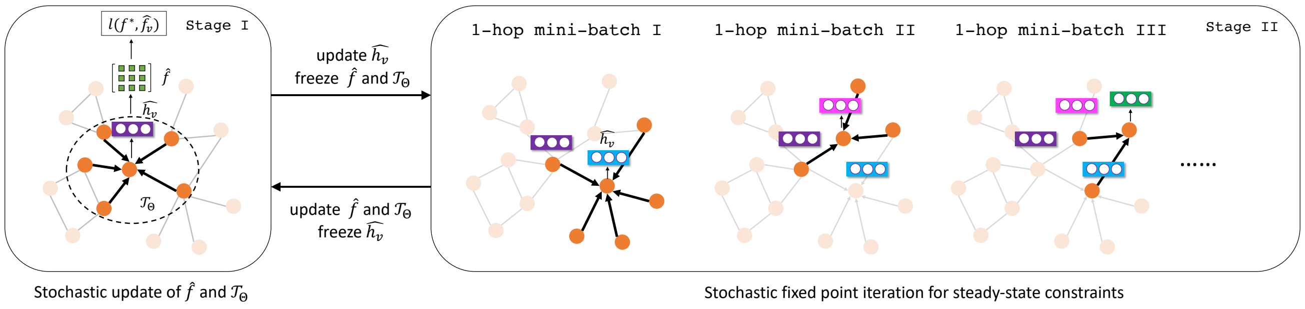

$l(f^*, \widehat{f}_v)$
Stage I
$\widehat{f}$
$\widehat{h_v}$
$\mathcal{T}_\Theta$
update $\widehat{h_v}$
freeze $\widehat{f}$ and $\mathcal{T}_\Theta$
update $\widehat{f}$ and $\mathcal{T}_\Theta$
freeze $\widehat{h_v}$
1-hop mini-batch I
1-hop mini-batch II
1-hop mini-batch III
Stage II
......
Stochastic update of $\widehat{f}$ and $\mathcal{T}_\Theta$
Stochastic fixed point iteration for steady-state constraints

Figure 1. Overview of proposed graph steady-state learning algorithm. In stage I, we update the classifier $\widehat{f}_v$ and steady-state operator $\mathcal{T}_\Theta$ with 1-hop neighborhood of stochastic samples; in stage II, the embeddings $\widehat{h}_v$ are updated by performing stochastic fixed point iterations.

which only requires 1-hop neighborhood for the update. We argue that such 1-hop stochasticity is key to the efficiency and effectiveness. Most of the GNN variants (e.g., Li et al. (2015)) need $O(T(|\mathcal{V}|+|\mathcal{E}|))$ computational cost and memory consumption per each round of parameter update. For large graphs, this would be quite expensive. Hamilton et al. (2017a) attempts the mini-batch update using T-hops neighborhood of sampled mini-batch of nodes. However, the neighborhood size grows exponentially with respect to T. As in the idea of six degrees of separation, $T=6$ would already include all the nodes in the social network.

We note that this new algorithm is significantly different from the traditional graph embedding settings where the goal is to learn representations (or features) for nodes in a graph for classification. In contrast, our goal is to efficiently learn an *algorithm* which can run in a large graph and can respect specific condition with physical meaning. The successive stochastic projection of the embeddings onto the steady-state condition, which is not present in previous graph embedding methods, is a crucial step in our algorithm, and creates an important inductive bias which allows us to generalize the learned steady-state algorithm output to the entire network and even to a different network.

We showed that our framework can be adapted to learn the steady-state of a few commonly used graph algorithms, namely the detection of connected components, PageRank scores, mean field inference, and node labeling problem over graphs. We conducted systematical comparison between the learned algorithms and several existing algorithms to demonstrate the benefits in terms of both effectiveness and scalability on both randomly generated graphs and real-world graphs. In particular, in the PageRank problem, the learned algorithm can easily handle graphs with more than 100,000,000 nodes in a single machine.

2. Iterative Algorithms over Graphs

Many iterative algorithms over graphs can be formulated into the form of Eq (1) and the solutions satisfy a requirement of the form of Eq (2). More specifically, for a graph, $\mathcal{G}=(\mathcal{V},\mathcal{E})$, with node set $\mathcal{V}$ and edge set $\mathcal{E}$, the iterative algorithm framework can be instantiated as follows

- **Graph component detection problem.** We want to find all nodes within the same connected component as source node $s\in\mathcal{V}$. This task can be solved by iteratively propagating the label at node s to other nodes
$$y_v^{(t+1)} = \max_{u\in\mathcal{N}(v)} y_u^{(t)},\ y_s^{(0)}=1, y_v^{(0)}=0, \forall v\in\mathcal{V}$$
where $\mathcal{N}(v)$ denotes the set of neighbors of v. At algorithm step $t=0$, the label $y_s^{(0)}$ at node s are set to 1 (infected) and 0 for all other nodes. The steady state is achieved when nodes in the same connected component as s are infected. That is $y_v^*=\max_{u\in\mathcal{N}(v)} y_u^*$.
- **PageRank scores for node importance.** We want to estimate the importance of each node in a graph. The scores can be initialized to 0 ($r_v^{(0)} \leftarrow 0, \forall v\in\mathcal{V}$) and updated iteratively as
$$r_v^{(t+1)} \leftarrow \frac{(1-\lambda)}{|\mathcal{V}|} + \frac{\lambda}{|\mathcal{N}(v)|}\sum_{u\in\mathcal{N}(v)} r_u^{(t)},\ \forall v\in\mathcal{V}.$$
The steady-state scores r_v^* will satisfy the relation $r_v^* = \frac{(1-\lambda)}{|\mathcal{V}|} + \frac{\lambda}{|\mathcal{N}(v)|}\sum_{u\in\mathcal{N}(v)} r_u^*$.
- **Mean field inference in graphical model.** We want to approximate the marginal distributions of a set of variables x_v in a graph model defined on $\mathcal{G}$. That is $p(\{x_v\}_{v\in\mathcal{V}}) \propto \prod_{v\in\mathcal{V}}\phi(x_v)\prod_{(u,v)\in\mathcal{E}}\phi(x_u,x_v)$ where $\phi(x_v)$ and $\phi(x_u,x_v)$ are the node and edge potential respectively. The marginal approximation $q(x_v)$ can be obtained in an iterative fashion by the following mean field update
$$q^{(t+1)}(x_v) \leftarrow \phi(x_v)\prod_{u\in\mathcal{N}(v)} \exp\left(\int_u q^{(t)}(x_u)\log\phi(x_u,x_v)\mathrm{d}u\right),$$
and the steady-state solution satisfies $q^*(x_v) = \phi(x_v)\prod_{u\in\mathcal{N}(v)}\exp\left(\int_u q^*(x_u)\log\phi(x_u,x_v)\mathrm{d}u\right)$.

- **Compute long range graph convolution features.** We want to extract long range features from graph and use that figure to capture the relation between graph topology and external labels. One possible parametrization of graph convolution features h_v can be updated from zeros initialization as
$$h_v^{(t+1)} \leftarrow \sigma\left(W_1 x_v + W_2 \sum_{u \in \mathcal{N}(v)} h_u^{(t)}\right)$$
where σ is a nonlinear elementwise operation, and W_1, W_2 are the parameters of the operator. The steady state is characterized as $h_v^* \leftarrow \sigma\Big(W_1 x_v + W_2 \sum_{u \in \mathcal{N}(v)} h_u^*\Big)$. Then the labeling function $f(h_v^*)$ for each node is determined by the state-steady feature h_v^*.

Typically, to learn these iterative algorithms with GNN family models, we need to run many iterations in order for them to converge to the steady-state solutions. Especially when the graph scale gets large, a large number of iterations are needed, making the GNNs very computationally intensive and slow. In the following, we will formulate a generic learning problem for designing a faster algorithms for these scenarios.

3. The Algorithm Learning Problem

In this section we propose a framework of algorithm design based on the idea of embedding the intermediate representation of an iterative algorithm over graphs into vector spaces, and then learn such algorithms using example outputs from the desired algorithms to be learned.

More specially, we assume that we have collected the output of an iterative algorithm $\mathcal{T}$ over a single large graph[1]. The training dataset consists of the input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and the output of the algorithm for a subset of nodes, $\mathcal{V}^{(y)} \subseteq \mathcal{V}$ from the graph:

$$\mathcal{D} = \left\{ f_v^* := f(h_v^*) \mid h_v^* = \mathcal{T}\left[\{h_u^*\}_{u \in \mathcal{N}(v)}\right], v \in \mathcal{V}^{(y)} \right\}. \quad (3)$$

In the dataset, h_v^* is the quantity in the algorithm which satisfies the steady-state conditions, and $f(\cdot)$ is an additional labeling function which takes the steady-state quantity and produces the final label for each node. In the case where h_v^* is the output of an algorithm, we can think of $f(\cdot)$ is the identity function.

Given the above dataset $\mathcal{D}$ from previous run of the algorithm, the goal is to learn a parameterized algorithm $\mathcal{A}_\Theta$ such that the output of the algorithm can mimic the output of the original algorithm $\mathcal{T}$. That is the learned algorithm $\mathcal{A}_\Theta$ produces $\{\widehat{f}_v\}_{v \in \mathcal{V}^{(y)}} = \mathcal{A}_\Theta[\mathcal{G}]$, which are close to f_v^* according to some loss function $\ell(f_v^*, \widehat{f}_v)$.

[1]Our method can also be used for the cases where data are collected from multiple graphs. In this case, we can view multiple graphs as a single big graph with a collection of connected components.

Overall, the algorithm learning problem for $\mathcal{A}_\Theta$ can be formulated into the following optimization problem

$$\min_{\Theta} \sum_{v \in \mathcal{V}^{(y)}} \ell(f_v^*, \widehat{f}_v) \quad (4)$$
$$\text{s.t. } \{\widehat{f}_v\}_{v \in \mathcal{V}^{(y)}} = \mathcal{A}_\Theta[\mathcal{G}] \quad (5)$$

In the above general statement of the learning problem, we have not specified the actual form of the algorithm and the parametrization of the algorithm step. In the following section we will explain our design of fast iterative algorithm which can be learned.

The design goal of our model will focus on two key aspects: respect steady-state conditions and learn fast. Thus the core of our model is a steady-state operator $\mathcal{T}_\Theta$ between vector embedding representation of nodes, and a link function mapping the embedding to the algorithm output. Furthermore, the embeddings are obtained by solving the steady-state operator stochastically, making it very efficient for large scale graph problems.

3.1. Steady-state operator and linking function

We will associate each node in the graph with an unknown vector embedding representation $\widehat{h}_v \in \mathbb{R}^d$, and the core of our algorithm is a parameterized operator, $\mathcal{T}_\Theta$, for enforcing steady-state relations between these embeddings. Given a link function $\widehat{f}(h_v)$, our model makes predictions on the algorithm outputs by the following operations

$$\text{output}: \left\{\widehat{f}_v := \widehat{f}(\widehat{h}_v)\right\}_{v \in \mathcal{V}} \quad (6)$$
$$\text{s.t. } \widehat{h}_v = \mathcal{T}_\Theta\left[\{\widehat{h}_u\}_{u \in \mathcal{N}(v)}\right] \quad (7)$$

In our model, the steady-state operator $\mathcal{T}_\Theta$ and the linking function $\widehat{f}$ is not fixed before hand, and their parameters will be learned from dataset $\mathcal{D}$ in Eq (3). Furthermore, the vector embeddings $\widehat{h}_v$ need to be found from Eq (7), after which the embeddings are used for making predictions about the algorithm outputs via $\widehat{f}$. Thus, we need an algorithm for finding the (approximate) steady-state of Eq (7).

3.2. Finding steady-state

Here we use an iterative algorithm to find the steady-state of Eq (7). The algorithm will execute in a similar fashion as randomized Gauss-Seidel method which updates one unknown variable at the time according to the steady-state equation. Adapting the scheme to our case, we will start all $\{\widehat{h}_v\}_{v \in \mathcal{V}}$ from some constant, and then update the embedding one at at time. That is

$$\widehat{h}_v \leftarrow \text{constant for all } v \in \mathcal{V}$$

for v sampled from $\mathcal{V}$:

$$\widehat{h}_v \leftarrow \mathcal{T}_\Theta\left[\{\widehat{h}_u\}_{u\in\mathcal{N}(v)}\right] \tag{8}$$

We note that in this randomized scheme, the embeddings $\{\widehat{h}_v\}_{v\in\mathcal{V}}$ are updated in an asynchronous fashion. Furthermore, each time the update is also carried out only one hop for the sampled node v. This makes it very efficient compared to synchronous update over the entire graph for T hops. For comparison, the synchronous update will amount to a computational complexity of $O(T(|\mathcal{V}|+|\mathcal{E}|))$ which quickly becomes prohibitive for large graphs. Instead, our steady-state finding algorithm is carried out using mini-batches.

3.3. Specific parameterization for $\mathcal{T}_\Theta$ and g

The operator $\mathcal{T}_\Theta$ and link function g can come from general nonlinear function class. The operator $\mathcal{T}_\Theta$ enforces the steady-state condition of node embeddings based on 1-hop local neighborhood information. Due to the variety of graph structures, this function should be able to handle different number of inputs (*i.e.*, different number of neighbor nodes) and be invariant to the ordering of these neighbors. In our work, we use the following parameterization:

$$\mathcal{T}_\Theta\left[\{\widehat{h}_u\}_{u\in\mathcal{N}(v)}\right] = W_1\sigma\left(W_2\left[x_v, \sum_{u\in\mathcal{N}(v)}[\widehat{h}_u, x_u]\right]\right) \tag{9}$$

where $\sigma(\cdot)$ is element-wise activation function, such as commonly used Sigmoid or ReLU. W_1 and W_2 are the weight matrices. x_v is the optional feature representation of nodes, such as observations in Markov Random Field (MRF). In general, a two-layer neural network formulation as above would be enough for most cases. But one can also use problem-specific parameterization for better performance.

For prediction function g, it takes the node embeddings as inputs, and predicts the corresponding algorithm outputs. We also adopt a two-layer neural network, *i.e.*,

$$g(\widehat{h}_v) = \sigma\left(V_2^\top \text{ReLU}(V_1^\top \widehat{h}_v)\right), \tag{10}$$

where V_1, V_2 are parameters of $g(\cdot)$. $\sigma(\cdot)$ is a task-specific activation function. For linear regression this is the identity function $\sigma(x) = x$. For multi-class classification problem, $\sigma(\cdot)$ is softmax which would output a probabilistic simplex.

3.4. The optimization problem

Thus the overall optimization problem for learning our model can be formulated as

$$\min_{\{W_i,V_i\}_{i=1}^2} \mathcal{L}\left(\{W_i,V_i\}_{i=1}^2\right) := \frac{1}{|\mathcal{V}^y|}\sum_{v\in\mathcal{V}^{(y)}} \ell(f_v^*, g(\widehat{h}_v))$$

$$\text{s.t. } \widehat{h}_v = \mathcal{T}_\Theta\left[\{\widehat{h}_u\}_{u\in\mathcal{N}(v)}\right], \forall v\in\mathcal{V}. \tag{11}$$

In the next section, we will introduce an alternating algorithm to solve the above optimization problem. The algorithm will alternate between using most current model to find the embeddings and make prediction, and using the gradient of the loss with respect to $\{W_1, W_2, V_1, V_2\}$ for update these parameters.

4. Learning Algorithm

It should be emphasized that directly applying the vanilla stochastic gradient descent requires visiting *all* the nodes in the graph many times due to the constraints in Eq. (11), making the reduction of the cost via stochastic gradient computation in vain. As we discussed in Section 3.2, this step is actually the computation bottleneck. In this section, we present a scalable algorithm which exploits the stochasticity in both equilibrium constraints and the objective in Eq. (11) to learn the parameters. Then, we provide the analysis of the computational and memory complexity in detail to show how our proposed approach could save the computation in Section 4.2.

4.1. Stochastic Fixed-Point Gradient Descent

In fact, the optimization Eq. (11) can be understood as improving the policy which minimizing the cost that is proportional to f^*. The fix-point equation characterizes the dynamic programming whose solution is steady state $\widehat{h}_v$ for each node. Comparing to the reinforcement learning (RL), it plays a similar role as "value function". With these estimations of the steady states, we minimize the cost by updating the parameters in $\mathcal{T}_\Theta$ and g, which can be understood as a similar role as "policy" in RL. Based on such understanding, we design our algorithm inspired by the policy iteration in reinforcement learning (Sutton & Barto, 1998). Furthermore, to reduce the complexity in the first stage for estimating, we introduce an extra randomness over the constraints and solve it approximately through *stochastic fixed point iteration*.

Stochastic gradient descent for "policy" improvement. Specifically, at k-th round in the stochastic optimization, once we have $\left\{\widehat{h}_v^k\right\}_{v\in\mathcal{V}}$ satisfying the steady-state equation, *i.e.*, $\widehat{h}_v^k = \mathcal{T}_\Theta\left[\{\widehat{h}_u^k\}_{u\in\mathcal{N}(v)}\right], \forall v \in \mathcal{V}$, we have the gradient estimators as

$$\frac{\partial\mathcal{L}}{\partial W_i} = \widehat{\mathbb{E}}\left[\frac{\partial\ell\left(f_v^*, g\left(\widehat{h}_v^k\right)\right)}{\partial g\left(\widehat{h}_v^k\right)}\frac{\partial g\left(\widehat{h}_v^k\right)}{\partial W_i}\right],$$

$$\frac{\partial\mathcal{L}}{\partial V_i} = \widehat{\mathbb{E}}\left[\frac{\partial\ell\left(f_v^*, g\left(\widehat{h}_v^k\right)\right)}{\partial \widehat{h}_v^k}\frac{\partial\mathcal{T}_\Theta\left[\{\widehat{h^k}_u\}_{u\in\mathcal{N}(v)}\right]}{\partial V_i}\right],$$

where the expectation $\widehat{\mathbb{E}}[\cdot]$ is taken w.r.t. uniform distribution over labeled nodes $\mathcal{V}^{(y)}$. With such treatment, we can update the parameters, *i.e.*, $\{W_1, W_2, V_1, V_2\}$, as vanilla stochastic gradient descent.

Stochastic fixed-point iteration for "value" estimation. However, it is prohibitive to solve the steady-state equation

Algorithm 1 Learning with Stochastic Fixed Point Iteration

```
1:  Initialize $W_1, W_2, V_1, V_2, \{\widehat{h}_v\}_{v\in\mathcal{V}}$ randomly
2:  for $k=1$ to $K$ do
3:      for $t_h=1$ to $n_h$ do
4:          Sample $\widetilde{\mathcal{V}}=\{v_1,v_2,...,v_N\}\in\mathcal{V}$
5:          Use Eq. (12) to update embedding $\widehat{h}_{v_i}, \forall v_i\in\widetilde{\mathcal{V}}$
6:      end for
7:      for $t_f=1$ to $n_f$ do
8:          Sample $\widetilde{\mathcal{V}}^{(y)}=\{v_1,v_2,...,v_M\}\in\mathcal{V}^{(y)}$
9:          $\{W_i\leftarrow W_i-\eta\frac{\partial\mathcal{L}}{\partial W_i}\}_{i=1}^2$, $\{V_i\leftarrow V_i-\eta\frac{\partial\mathcal{L}}{\partial V_i}\}_{i=1}^2$
10:     end for
11: end for
```

exactly in large-scale graph with millions of vertices since it requires visiting all the nodes in the graph. Therefore, we introduce the extra randomness on the constraints for sampling the constraints to tackle the groups of equations approximately. This technique is very effective in dealing with infinite constraints in approximately solving MDP (De Farias & Van Roy, 2003; 2004).

Specifically, in k-th step, we first sample a set of nodes $\widetilde{\mathcal{V}}=\{v_1,v_2,...,v_N\}\in\mathcal{V}$ from the entire node set rather of the labeled set. For stability, we update the new embedding of v_i by moving average in following form:

$$\widehat{h}_{v_i}\leftarrow(1-\alpha)\widehat{h}_{v_i}+\alpha\mathcal{T}_\Theta\left[\{\widehat{h}_u\}_{u\in\mathcal{N}(v_i)}\right], \forall v_i\in\widetilde{\mathcal{V}}. \quad (12)$$

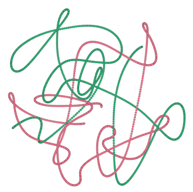
The overall algorithm is summarized in Algorithm 1. The whole iterative process will run K steps or untill convergence. During each macro iteration, the two stages can also have multiple inner loops. Specifically, let n_f be the number of inner loops for "policy" improvement, and n_h be the number of inner loops in "value" estimation.During the experiment we found that, having more fixed point iterations, *i.e.*, $n_h > n_f$ helps the model converge faster and achieve better generalization.

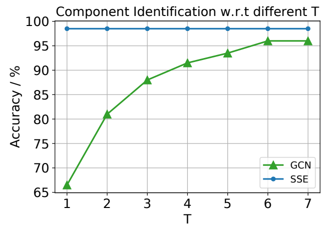
We name our algorithm Stochastic Steady-state Embedding (SSE), due to its stochasticity nature and steady-state enforcement.

4.2. Complexity analysis

In this section, we briefly analyze the computation and memory complexity of Algorithm 1.

In "policy" improvement stage, assume the labeled set $\mathcal{V}^{(y)}$ is an unbiased sample from $\mathcal{V}$, then the computational cost is $\Theta(M\frac{|\mathcal{E}|}{|\mathcal{V}|})$, since we only need 1-hop nodes to update. Here we use the average node degree in graph to calculate the expected number of edges in each mini-batch. Similarly, in "value" estimation stage, we have $\Theta(N\frac{|\mathcal{E}|}{|\mathcal{V}|})$. So in summary, the computational cost in each iteration is just proportional to the number of edges in each mini-batch.


(a) Graph consists of two disjoint chains.



(b) Accuracy *w.r.t.* different T.

Figure 2. Graph connectivity experiment.

The memory cost of our algorithm is also smaller compared to the existing graph neural networks. Regardless of necessary memory held by parameters $W_{1,2}$, $V_{1,2}$ and node/edge features, the dominating part is the persistent node embedding matrix $\{\widehat{h}_v\}_{v\in\mathcal{V}}$ which takes $O(|\mathcal{V}|)$ space. This is also much cheaper than most GNN-family models which take $O(T|\mathcal{V}|)$ space, due to the requirement of storing intermediate embeddings for back-propagation use.

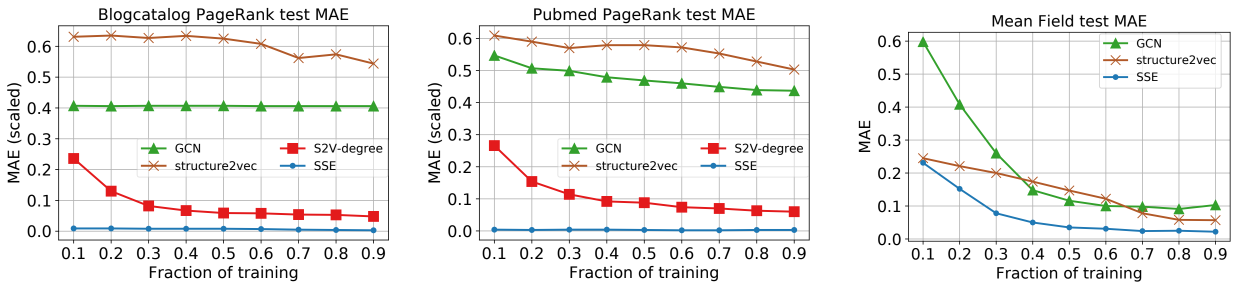
5. Experiments

In this section, we experimentally demonstrate the effectiveness and efficiency of learning various graph algorithms. We compare our proposed algorithm with some of the GNN variants who have the fixed finite number of propagations T, using experiments with both transductive and inductive settings. In transductive setting, we compare with GCN (Kipf & Welling, 2016), a localized first-order approximation of spectral graph convolutions and structure2vec (Dai et al., 2016) which mimics the graphical model inference algorithms to obtain feature representation. The number of propagation steps is tuned in $T\in\{1,...,7\}$ for them. In inductive setting, we compare with GraphSage (Hamilton et al., 2017a) and its variants. For our proposed algorithm, We tune the number of inner loops for SGD and fixed point iterations $n_f, n_h\in\{1,5,8\}$, to balance the parameter learning and fixed point constraint satisfaction.

We demonstrate the effectiveness of the proposed algorithm in capturing steady-state information with learning graph algorithms, *i.e.*, the graph connectivity detection, PageRank, and Mean Field Inference on graphical model, where the global-range steady information is the key for success, in Section 5.1, 5.2 and 5.3, respectively. We also show the comparison on benchmark datasets in Section 5.4, where we can achieve comparable or better accuracy. Finally we show our advantage in terms of scalability in Section 5.5.

5.1. Algorithm-learning: connectivity

The graph we constructed contains 2 disjoint chains. Each chain is a connected component which contains 1,000 nodes and 999 edges. Figure 2a illustrates the graph we created.



(a) PageRank on BlogCatalog (b) PageRank on Pubmed (c) Mean-Field Inference for MRF

Figure 3. Algorithm learning for PageRank and Mean Field Inference. Error is measured using Mean Absolute Error (MAE).

The algorithm needs to know the multi-hop structure, in order to identify the component ID for a certain node. In this transductive setting, we use 10% nodes with labels for training, and the rest for testing. With proper parameter tuning, the GCN and structure2vec can achieve 96% accuracy in distinguishing two components, while our SSE gets 99%.

In Figure 2b, we vary the T of GCN, and report its test performance. Since our proposed algorithm doesn't have the dependency over T, we simply include it as a reference. We can see as T gets larger, the GCN model converges to better solution by taking longer range of information, while the computational cost increases linearly with T. Also through this experiment we find it is not only computationally more efficient, but also experimentally more effective in learning the steady-states.

5.2. Algorithm Learning: PageRank

In this task, we learn to predict the PageRank scores for each node in the network graph. In our experiment we use the default value (which is 0.85) for the damping factor.

Real-world graphs: We take the Blogcatalog and Pubmed graphs for evaluation (graph statistics can be found in Table 6 and Table 5 in Appendix). The dataset contains about $10k \sim 20k$ nodes. For each dataset, we first run the PageRank algorithm using networkx (Hagberg et al., 2008). Since the raw PageRank scores are normalized to a probabilistic simplex, we rescale it by multiplying the total number of nodes. This avoids some precision issue of the float numbers. In transductive setting, we reserve 10% nodes for held-out evaluation, and vary the training set size from 10% to 90% of the total nodes. We also modify the vanilla structure2vec model to use degree-weighted message aggregation, denoted as S2V-degree, for better performance in PageRank prediction task.

The quantitative results are shown in Figure 3a and 3b, respectively. We can see from the figure that, our proposed algorithm can achieve almost perfect fitting results on all the two datasets, even with only 10% nodes for training. However, although we've shown that with larger T the GCN can match our performance in Section 5.1, it is not effective in current experiment. Simply making T larger will cause problem for both gradient propagation and memory consumption, and thus it is not effective. The modified baseline S2V-degree performs the second best, so we compare with it in detail on Barabasi-Albert random graphs in next part.

Barabasi-Albert random graphs: To evaluate how the performance varies as graph size grows, we further carry out experiments on Barabasi-Albert (BA) graphs. We vary the number of nodes $n \in \{1k, 10k, 100k, 1m, 10m\}$, and use two different parameters $m=1$ and $m=4$ for BA model. It is known that when $m=1$ the graph has diameter of $O(\log n)$ and for $m \geq 2$ it is $O(\log n / \log\log n)$ (Bollobás & Riordan, 2004). Thus for $m=1$, it is more challenging since the number of hops of information need is larger.

In transductive setting, we split the nodes equally into training and test set; in inductive setting, the training is performed in a single graph, while the algorithm is asked to generalize to new graphs from the same distribution. For S2V-degree we set $T=5$ due to the consideration of feasibility. The transductive and inductive results are shown in Table 1 and 2, respectively. As is expected, the MAE in $m=4$ setting is lower than that in $m=1$ setting. Our proposed algorithm achieves almost perfect MAE and increases slightly when the prediction task becomes more and more challenging as the size of graph increasing to 10m nodes. In comparison, the performance of S2V-degree is significantly worse, especially when graph size grows. This is because $T=5$ propagations cannot capture enough long range information. We emphasize that for large graphs with 10m nodes, it is also hard for batch algorithm like S2V-degree to converge and generalize well.

5.3. Algorithm Learning: mean-field inference

To further evaluate the ability of our proposed algorithm in capturing the steady-state information, we design a task to fit the posteriors from the mean-field (MF) inference algorithm. Here we define a lattice graph over a 128×128 grid. Specifically, we focus on the pair-wise Markov

Table 1. Transductive learning of PageRank on Barabasi-Albert graphs with different sizes and hyperparameters ($m = 1,4$). We report MAE on 50% held-out nodes.

	# nodes	1k	10k	100k	1m	10m
m=1	S2V-degree	0.0652	0.0843	0.1444	0.4012	0.4954
	SSE	0.0041	0.0054	0.0075	0.0088	0.0162
	# nodes	1k	10k	100k	1m	10m
m=4	S2V-degree	0.0138	0.0165	0.0347	0.0944	0.1223
	SSE	0.0043	0.0051	0.0056	0.0065	0.0083

Table 2. Inductive learning of PageRank on Barabasi-Albert graphs, trained on graph with same hyper-parameters.

	# nodes	1k	10k	100k	1m	10m
m=1	S2V-degree	0.0783	0.0956	0.1931	0.4532	0.5254
	SSE	0.0062	0.0074	0.0073	0.0097	0.0202
	# nodes	1k	10k	100k	1m	10m
m=4	S2V-degree	0.0172	0.0193	0.0394	0.1243	0.1527
	SSE	0.0057	0.0063	0.0066	0.0079	0.0101

Random Field graphical model:

$$P(\{\mathbf{H}_v\},\{\mathbf{x}_v\}) \propto \prod_{v\in\mathcal{V}} \Phi(\mathbf{H}_v,\mathbf{x}_v) \prod_{(u,v)\in\mathcal{E}} \Psi(\mathbf{H}_u,\mathbf{H}_v) \quad (13)$$

where $\mathbf{x}_v$ is the observation and $\mathbf{H}_v$ is the latent variable. The mean-field score for each $\mathbf{H}_v$ is a vector calculated using the UGM toolset [2]. The task is to learn the mean-field score $q(\mathbf{H}_v)$ for each node over a 128×128 lattice with $\mathbf{x}_v$ set to be binary with a Gaussian perturbation. The posterior in this case can be understood as steady-state that is expressed by nonlinear fixed point equation. We test the learned mean-field scores on the 10% of the vertices and vary the size of training set sampled from the remaining vertices.

From Figure 3c we can see, our proposed algorithm still works best regarding the MAE metric, and can achieve better results with fewer labeled vertices. Here the fixed point equations are nonlinear, which is different from the PageRank experiment. The baseline algorithms can also achieve good performance with more supervision.

5.4. Application: node classification

Transductive setting:

To demonstrate the effectiveness of addressing steady-state information, we conduct experiments on a large graph dataset, namely the Amazon product co-purchasing network dataset (Yang & Leskovec, 2015)[3]. Among the 75,149 product types, we select those with at least 5,000 products. This results in 58 labels finally.The statistics of dataset can be found in Table 6 in Appendix.

From Table 3 we can see the SSE outperforms the baselines by a large margin. We also observed that in Amazon dataset, GNN-family models benefit more from more supervision, due to the larger model capacity. Our proposed method achieves the best performance, regardless of the amount of supervision available. This suggests that our algorithm can effectively utilize the global-range information of graph structure.

To make the comparison comprehensive, we also conduct experiments on small benchmark datasets that are commonly used in the literature. Details can be found in Appendix A.2.2 and Appendix A.2.1, for multi-class citation network classification and multi-label classification, respectively. Since the graphs are small, typically for GNN family models, $T = 2$ would be enough to get good prediction. Nonetheless, the SSE still achieves comparable results.

Inductive setting:

In this setting, we use the PPI dataset from GraphSage Hamilton et al. (2017a), which contains 56,944 nodes (for proteins) and 818,716 edges (for their interactions). It is a multi-label classification tasks, where each protein can have at most 121 labels. Each protein is associated with additional 50-dimensional features. We use the same train/valid/test split as in Hamilton et al. (2017a).

Table 4 shows the results. The GraphSage results are taken from the original paper, since we are using the exactly same setting. We can see regarding the Micro-F1 metric, our proposed SSE achieves much better performance. We show that, since GraphSage is trained using mini-batch of nodes within T-hops, it is not effective enough to capture the steady-state information, which in this case seems essential.

5.5. Scalability

In this section, we demonstrate that the proposed algorithm is very efficient for large-scale graphs in terms of both convergence speed and execution time.

5.5.1. Time per update

All the algorithms are executed on a 16-core cluster with 256GB memory. We evaluate the wall-clock time cost per update. For baselines GCN and structure2vec, this corresponds to one feedforward and back propagation round with T-step embedding propagation on entire graph; for our method, this corresponds to $n_f + n_h$ mini-batch updates. Here we focus on models in GNN family. For GCN and structure2vec, we compare with $T = 1$ and $T = 5$; while in our method, $n_f = 1$ and $n_h = 5$.

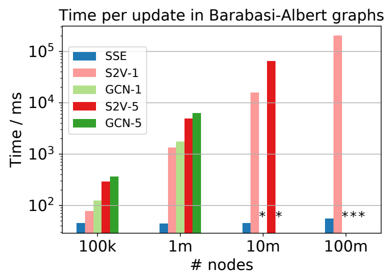
The task we choose here is PageRank in Section 5.2. The graphs we evaluate on are generated using Barabasi-Albert

[2]https://www.cs.ubc.ca/~schmidtm/Software/UGM.html

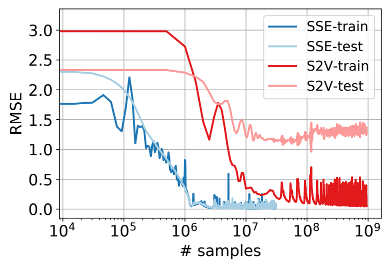
[3]http://snap.stanford.edu/data/com-Amazon.html.

Table 3. Multi-label classification in Amazon product dataset. We report both Micro-F1 and Macro-F1 on held-out test set.

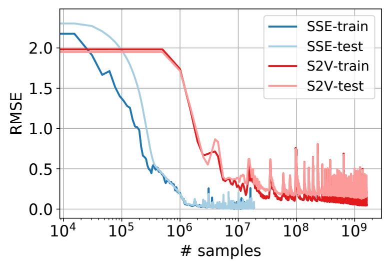
Amazon	Micro-F1/%									Macro-F1/%								
Methods	1%	2%	3%	4%	5%	6%	7%	8%	9%	1%	2%	3%	4%	5%	6%	7%	8%	9%
structure2vec	70.27	74.54	77.18	79.95	80.97	81.58	82.71	83.27	83.55	66.62	70.07	74.74	76.43	77.62	78.65	79.92	80.13	80.11
GCN	70.39	73.58	77.61	80.34	82.03	83.23	84.25	85.1	85.68	66.16	71.01	74.56	77.11	78.97	80.5	81.36	82.15	82.75
SSE	**78.36**	**81.06**	**82.61**	**83.79**	**84.59**	**85.08**	**85.68**	**86.57**	**87.13**	**75.07**	**77.67**	**79.03**	**79.86**	**81.14**	**81.59**	**82.39**	**83.13**	**84.03**



(a) Wall-clock time per round of update. The (*) in the figure denotes the out-of-memory error.



(b) Convergence on BA graphs with # nodes=1,000,000 and m=1.



(c) Convergence on BA graphs with # nodes=1,000,000 and m=4.

Figure 4. Results on scalability experiments. We compare both the time needed per update, as well as number of samples required for convergence in PageRank experiments with large Barabasi-Albert random graphs.

Table 4. Inductive node classification using PPI dataset.

Method	Micro-F1
GraphSAGE-GCN	0.500
GraphSAGE-mean	0.598
GraphSAGE-LSTM	0.612
GraphSAGE-pool	0.600
SSE	**0.836**

model with $m = 4$ as its parameter. We vary the number of nodes in $\{100k, 1m, 10m, 100m\}$, and report the time in milliseconds in Figure 4a.

The results show our algorithm takes almost constant time for each update, due to its stochasticity nature. As graph size grows, the time cost for GCN and structure2vec grows linearly. For graph with $100m$ nodes, storing the intermediate updates and gradients for $T = 5$ in structure2vec is no longer feasible [4].

5.5.2. Convergence

Here we compare the number of samples required for different algorithms to converge to a good solution. Figure 4b and 4c show the curves. We take the Barabasi-Albert graphs with 1,000,000 node and two different settings of $m = 1$ and $m = 4$, and fit with the PageRank scores on 50% nodes. We also visualize the test error convergence curve on the held-out 50% nodes. Both training and test curves report the RMSE (root mean square error), since we use this metric for optimization.

We compare with S2V-degree with $T = 5$, which achieves second best results in Section 5.2. For our algorithm, each round of updates requires $256 \times (n_f + n_h)$ samples. Here 256 is the mini-batch size we used, while S2V-degree needs the whole graph per update.

From the figures we can see our proposed algorithm converges much faster than the S2V, in terms of number of samples. The number of samples required by our algorithm is equivalent to only scanning through the entire training set for 4 or 5 passes. While for S2V-degree, it requires hundreds or thousands of passes to converge. Also note that, S2V-degree with $T = 5$ gets much worse test error in the case when $m = 1$, due to its limited ability for capturing steady-state information.

6. Conclusion

In this paper, we presented SSE, an algorithm that can learn many steady-state algorithms over graphs. Different from graph neural network family models, SSE is trained stochastically which only requires 1-hop information, but can capture fixed point relationships efficiently and effectively. We demonstrate this in both synthetic and real-world benchmark datasets, with transductive and inductive experiments for learning various graph algorithms. The algorithm also scales well up to 100m nodes with much less training effort. Future work includes investigation in learning more complicated graph algorithms, as well as distributed training.

[4]Note that for open source implementation of GCN, the Tensorflow limits the # elements in sparse matrix. That's why it cannot work on graphs with $10m$ nodes.

Acknowledgements

This project was supported in part by NSF IIS-1218749, NIH BIGDATA 1R01GM108341, NSF CAREER IIS-1350983, NSF IIS-1639792 EAGER, NSF CNS-1704701, ONR N00014-15-1-2340, Intel ISTC, NVIDIA and Amazon AWS. We thank the anonymous reviewers who gave useful comments.

References

Bollobás, B. and Riordan, O. The diameter of a scale-free random graph. *Combinatorica*, 24(1):5–34, 2004.

Breitkreutz, B.-J., Stark, C., Reguly, T., Boucher, L., Breitkreutz, A., Livstone, M., Oughtred, R., Lackner, D. H., Bähler, J., Wood, V., et al. The biogrid interaction database: 2008 update. *Nucleic acids research*, 36 (suppl_1):D637–D640, 2007.

Dai, H., Dai, B., and Song, L. Discriminative embeddings of latent variable models for structured data. In *ICML*, 2016.

De Farias, D. P. and Van Roy, B. The linear programming approach to approximate dynamic programming. *Operations research*, 51(6):850–865, 2003.

De Farias, D. P. and Van Roy, B. On constraint sampling in the linear programming approach to approximate dynamic programming. *Mathematics of operations research*, 29 (3):462–478, 2004.

Duvenaud, D. K., Maclaurin, D., Iparraguirre, J., Bombarell, R., Hirzel, T., Aspuru-Guzik, A., and Adams, R. P. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in Neural Information Processing Systems*, pp. 2215–2223, 2015.

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

Grover, A. and Leskovec, J. node2vec: Scalable feature learning for networks. In *KDD*, 2016.

Hachmann, J., Olivares-Amaya, R., Atahan-Evrenk, S., Amador-Bedolla, C., Sánchez-Carrera, R. S., Gold-Parker, A., Vogt, L., Brockway, A. M., and Aspuru-Guzik, A. The harvard clean energy project: large-scale computational screening and design of organic photovoltaics on the world community grid. *The Journal of Physical Chemistry Letters*, 2(17):2241–2251, 2011.

Hagberg, A., Swart, P., and S Chult, D. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Laboratory (LANL), 2008.

Hamilton, W. L., Ying, R., and Leskovec, J. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*, 2017a.

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

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

Lei, T., Jin, W., Barzilay, R., and Jaakkola, T. Deriving neural architectures from sequence and graph kernels. *arXiv preprint arXiv:1705.09037*, 2017.

Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

Mahoney, M. W. Randomized algorithms for matrices and data. *Foundations and Trends® in Machine Learning*, 3(2):123–224, 2011.

Page, L., Brin, S., Motwani, R., and Winograd, T. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. The graph neural network model. *Neural Networks, IEEE Transactions on*, 20(1):61–80, 2009.

Sen, P., Namata, G., Bilgic, M., Getoor, L., Galligher, B., and Eliassi-Rad, T. Collective classification in network data. *AI magazine*, 29(3):93, 2008.

Sutton, R. and Barto, A. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

Trivedi, R., Dai, H., Wang, Y., and Song, L. Know-evolve: Deep temporal reasoning for dynamic knowledge graphs. In *ICML*, 2017.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Yang, J. and Leskovec, J. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

Zafarani, R. and Liu, H. Social computing data repository at ASU, 2009. URL http://socialcomputing.asu.edu.