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 a node in a graph can be computed iteratively by averaging 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 $T$ as

$$h_v^{(t+1)} \leftarrow T (\{ h_u^{(t)} \}_{u \in \mathcal{N}(v)}), \quad \forall t \geq 1,$$

and

$$h_v^{(0)} \leftarrow \text{constant}, \quad \forall v \in \mathcal{V}$$

until the steady-state conditions are met

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

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 $T_{\theta}$ where the parameters $\theta$ 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,
Learning Steady-States of Iterative Algorithms over Graphs

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 $O(T(|V| + |E|))$ 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 stochastic samples; in stage II, the embeddings $\hat{h}_v$ are updated by performing stochastic fixed point iterations.

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, $G = (V, E)$, with node set $V$ and edge set $E$, 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 $s \in V$. This task can be solved by iteratively propagating the label at node $s$ to other nodes
  
  $y_u^{t+1} = \max_{v \in N(v)} y_v^{(t)}, \quad y_s^{(0)} = 1, \forall v \in V$

  where $N(v)$ denotes the set of neighbors of $v$. At algorithm step $t = 0$, the label $y_v^{(0)}$ 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 $y_v^* = \max_{u \in N(v)} y_u^*$.

- **PageRank scores for node importance.** We want to estimate the importance of each node in a graph. The scores can be initialized to 0 ($r_v^{(0)} \leftarrow 0, \forall v \in V$) and updated iteratively as

  $$r_v^{(t+1)} \leftarrow \frac{(1 - \lambda)}{|V|} + \frac{\lambda}{|N(v)|} \sum_{u \in N(v)} r_v^{(0)}, \quad \forall v \in V.$$ 

  The steady-state scores $r_v^*$ will satisfy the relation $r_v^* = \frac{(1 - \lambda)}{|V|} + \frac{\lambda}{|N(v)|} \sum_{u \in N(v)} r_u^*$.

- **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 $G$. That is

  $$p(\{x_v\}_{v \in V}) \propto \prod_{v \in V} \phi(x_v) \prod_{(u,v) \in E} \phi(x_u, x_v)$$

  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

  $$q^{(t+1)}(x_v) \leftarrow \phi(x_v) \prod_{u \in N(v)} \exp \left( \int_u q^{(t)}(x_u) \log \phi(x_u, x_v) du \right),$$

  and the steady-state solution satisfies $q^*(x_v) = \phi(x_v) \prod_{u \in N(v)} \exp(\int_u q^*(x_u) \log \phi(x_u, x_v) du)$. 
• Compute long range graph convolution features.

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

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

\[
\text{output: } \{ \hat{f}_v := \hat{f}(\hat{h}_v) \}_{v \in \mathcal{V}} \quad (6)
\]

\[
\text{s.t. } \hat{h}_v = \mathcal{T}_\Theta \left[ \{ \hat{h}_u \}_{u \in \mathcal{N}(v)} \right] \quad (7)
\]

In our model, the steady-state operator \( \mathcal{T}_\Theta \) and the linking function \( \hat{f} \) is not fixed before hand, and their parameters will be learned from dataset \( \mathcal{D} \) in Eq (3). Furthermore, the vector embeddings \( \hat{h}_v \) need to be found from Eq (7), after which the embeddings are obtained by solving the steady-state operator stochastically, making it very efficient for large scale graph problems.

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 \( \hat{h}_v \) from some constant, and then update the embedding one at a time. That is

\[
\hat{h}_v \leftarrow \text{constant} \text{ for all } v \in \mathcal{V}
\]
We note that in this randomized scheme, the embeddings \( \{ \hat{h}_v \}_{v \in V} \) 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 \( O(T(|V|+|E|)) \) which quickly becomes prohibitive for large graphs. Instead, our steady-state finding algorithm is carried out using mini-batches.

3.3. Specific parameterization for \( T_\Theta \) and \( g \)

The operator \( T_\Theta \) and link function \( g \) can come from general nonlinear function class. The operator \( T_\Theta \) 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:

\[
T_\Theta \left[ \{ \hat{h}_u \}_{u \in N(v)} \right] = W_1 \sigma \left( W_2 [x_v, \sum_{u \in N(v)} \hat{h}_u, x_u] \right)
\]

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.,

\[
g(\hat{h}_v) = \sigma(V_1^\top \text{ReLU}(V_2^\top \hat{h}_v)),
\]

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

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

s.t. \( \hat{h}_v = T_\Theta \left[ \{ \hat{h}_u \}_{u \in N(v)} \right], \forall v \in V. \)

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 \( \hat{h}_v \) 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 \( T_\Theta \) 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 \( \{ \hat{h}^k_v \}_{v \in V} \) satisfying the steady-state equation, i.e., \( \hat{h}^k_v = T_\Theta \left[ \{ \hat{h}^k_u \}_{u \in N(v)} \right], \forall v \in V \), we have the gradient estimators as

\[
\frac{\partial \mathcal{L}}{\partial W_i} = \mathbb{E} \left[ \frac{\partial f_v^*}{\partial \hat{h}_v} \frac{\partial \hat{h}_v}{\partial W_i} \right],
\]

\[
\frac{\partial \mathcal{L}}{\partial V_i} = \mathbb{E} \left[ \frac{\partial f_v^*}{\partial \hat{h}_v} \frac{\partial \hat{h}_v}{\partial V_i} \right],
\]

where the expectation \( \mathbb{E} \left[ \cdot \right] \) is taken w.r.t. uniform distribution over labeled nodes \( \mathcal{Y}^{(v)} \). 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 $W_1, W_2, V_1, V_2, \{\hat{h}_v\}_{v \in V}$ randomly
2: for $k = 1$ to $K$ do
3:   for $t_h = 1$ to $n_h$ do
4:     Sample $\overline{V} = \{v_1, v_2, ..., v_N\} \in V$
5:     Use Eq. (12) to update embedding $\hat{h}_{v_i}, \forall v_i \in \overline{V}$
6:   end for
7: for $t_f = 1$ to $n_f$ do
8:   Sample $\overline{V}^{(y)} = \{v_1, v_2, ..., v_M\} \in \mathcal{V}^{(y)}$
9:   $\{W_i \leftarrow W_i - \eta \frac{\partial L}{\partial W_i} \}_{i=1}^T, \{V_i \leftarrow V_i - \eta \frac{\partial L}{\partial V_i} \}_{i=1}^T$
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 $\mathcal{V} = \{v_1, v_2, ..., v_N\} \in V$ 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:

$$\hat{h}_{v_i} \leftarrow (1-\alpha)\hat{h}_{v_i} + \alpha T_\theta \left[ \{\hat{h}_u\}_{u \in N(v_i)} \right] \cdot \forall v_i \in \overline{V}. \quad (12)$$

The overall algorithm is summarized in Algorithm 1. The whole iterative process will run $K$ steps or until 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 $\Theta(M|\mathcal{V}^{(y)})$, 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 $\Theta(N|\mathcal{V}^{(y)})$. So in summary, the computational cost in each iteration is just proportional to the number of edges in each mini-batch.
Learning Steady-States of Iterative Algorithms over Graphs

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 parameter $m = 1$ 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 $128 \times 128$ 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.

<table>
<thead>
<tr>
<th># nodes</th>
<th>m=1</th>
<th>m=4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S2V-degree</td>
<td>SSE</td>
</tr>
<tr>
<td>l k</td>
<td>0.0652</td>
<td>0.0041</td>
</tr>
<tr>
<td>1 l</td>
<td>0.0843</td>
<td>0.0054</td>
</tr>
<tr>
<td>10 k</td>
<td>0.1444</td>
<td>0.0075</td>
</tr>
<tr>
<td>100 k</td>
<td>0.4012</td>
<td>0.0088</td>
</tr>
<tr>
<td>1 m</td>
<td>0.4954</td>
<td>0.0162</td>
</tr>
<tr>
<td>10 m</td>
<td>0.4954</td>
<td>0.0162</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th># nodes</th>
<th>m=1</th>
<th>m=4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S2V-degree</td>
<td>SSE</td>
</tr>
<tr>
<td>l k</td>
<td>0.0783</td>
<td>0.0062</td>
</tr>
<tr>
<td>1 l</td>
<td>0.0956</td>
<td>0.0074</td>
</tr>
<tr>
<td>10 k</td>
<td>0.1931</td>
<td>0.0073</td>
</tr>
<tr>
<td>100 k</td>
<td>0.4532</td>
<td>0.0097</td>
</tr>
<tr>
<td>1 m</td>
<td>0.5254</td>
<td>0.0202</td>
</tr>
<tr>
<td>10 m</td>
<td>0.5254</td>
<td>0.0202</td>
</tr>
</tbody>
</table>

Random Field graphical model:

\[
P(\{H_v\}, \{x_v\}) \propto \prod_{v \in V} \Phi(H_v, x_v) \prod_{(u, v) \in E} \Psi(H_u, H_v)
\]

where \(x_v\) is the observation and \(H_v\) is the latent variable. The mean-field score for each \(H_v\) is a vector calculated using the UGM toolset. The task is to learn the mean-field score \(q(H_v)\) for each node over a 128 \(\times\) 128 lattice with \(x_v\) 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 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. (2017), which contains 56,944 nodes (for proteins) and 818,716 edges (for their interactions). It is a multi-label classification task, 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. (2017).

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Micro-F1/°%</th>
<th>Macro-F1/°%</th>
</tr>
</thead>
<tbody>
<tr>
<td>structure2vec</td>
<td>66.62/70.07</td>
<td>77.62/77.11</td>
</tr>
<tr>
<td>GCN</td>
<td>66.16/71.01</td>
<td>78.97/80.5</td>
</tr>
<tr>
<td>SSE</td>
<td>75.07/77.67</td>
<td>79.86/81.14</td>
</tr>
</tbody>
</table>

### Table 4. Inductive node classification using PPI dataset.

<table>
<thead>
<tr>
<th>Method</th>
<th>Micro-F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphSAGE-GCN</td>
<td>0.500</td>
</tr>
<tr>
<td>GraphSAGE-mean</td>
<td>0.598</td>
</tr>
<tr>
<td>GraphSAGE-LSTM</td>
<td>0.612</td>
</tr>
<tr>
<td>GraphSAGE-pool</td>
<td>0.600</td>
</tr>
<tr>
<td>SSE</td>
<td>0.836</td>
</tr>
</tbody>
</table>

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.

#### 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.

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.
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


Appendix

A. More on experiments

A.1. Dataset information

Table 5. Multi-class node classification Dataset statistics as reported in Kipf & Welling (2016).

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Nodes</th>
<th># Edges</th>
<th># Classes</th>
<th># Features</th>
<th>Label rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Citeseer</td>
<td>3,327</td>
<td>4,732</td>
<td>6</td>
<td>3,703</td>
<td>3.6%</td>
</tr>
<tr>
<td>Cora</td>
<td>2,708</td>
<td>5,429</td>
<td>7</td>
<td>1,433</td>
<td>5.2%</td>
</tr>
<tr>
<td>Pubmed</td>
<td>19,717</td>
<td>44,338</td>
<td>3</td>
<td>500</td>
<td>0.3%</td>
</tr>
</tbody>
</table>

Table 6. Multi-label node classification Dataset statistics.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Nodes</th>
<th># Edges</th>
<th># Labels</th>
<th>Label type</th>
<th>Graph type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlogCatalog</td>
<td>10,312</td>
<td>331,983</td>
<td>39</td>
<td>membership</td>
<td>social network</td>
</tr>
<tr>
<td>PPI(transductive)</td>
<td>3,890</td>
<td>76,584</td>
<td>50</td>
<td>Bio-states</td>
<td>protein</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>4,777</td>
<td>184,812</td>
<td>40</td>
<td>POS-tag</td>
<td>word-net</td>
</tr>
<tr>
<td>Amazon</td>
<td>334,863</td>
<td>925,872</td>
<td>58</td>
<td>product type</td>
<td>co-purchasing</td>
</tr>
<tr>
<td>PPI(inductive)</td>
<td>56,944</td>
<td>818,716</td>
<td>121</td>
<td>Bio-states</td>
<td>protein</td>
</tr>
</tbody>
</table>

The real-world dataset used are shown in Table 5 and Table 6. The multi-class classification datasets are from Kipf & Welling (2016), where the multi-label classification datasets are from Grover & Leskovec (2016), Hamilton et al. (2017a) and SNAP website. Datasets in Table 5 and also the inductive PPI dataset have extra node features. When available, we use the same train/valid/test split as in original paper.

A.2. Experiments on small graphs

In this section, we compare with baseline algorithms on small benchmark datasets. We show that in the graphs where the diameter is small, existing algorithms can do pretty good, since local information is almost equivalent to global-range information. Nonetheless, our SSE can still achieve comparable performance in this scenario.

A.2.1. Multi-label classification

Here we compare our proposed method on the multi-label benchmark datasets. We include all the datasets used in Grover & Leskovec (2016) (namely, BlogCatalog (Zafarani & Liu, 2009), Protein-Protein Interactions (PPI) (Breitkreutz et al., 2007) and Wikipedia (Mahoney, 2011)). All the statistics of the dataset can be found in Table 6.

The evaluation metric we used here is Micro-F1 and Macro-F1 score. We tuned the hyperparameters for all the algorithms on 10% of training nodes, and then trained the model on full training set. The dimension of the embedding is set to 128. The results are shown in Table 7. We achieve the best results in Wikipedia, while getting comparable performances on the other two. In dataset like Blogcatalog, a small local neighborhood would be enough to infer the group membership of users in this friendship network, thus our approach would not benefit from taking global-range of information into account. However, in the Wikipedia dataset where we achieves the best Micro-F1 and Macro-F1 scores, it is important to know long range information to get a consensus among POS-tag labeling.

A.2.2. Document classification

In this section, we evaluate the performances on several benchmark citation graphs, namely Citeseer, Cora an Pubmed (Sen et al., 2008). The task is to do document classification, where each node in the citation graph represents the corresponding document. Different from the experiment in Section A.2.1, here the documents have auxiliary bag-of-words features. Since the document classes are mutually exclusive, we train all the models with Cross Entropy loss.

The statistics of the datasets are shown in Table 5. The number of features corresponds to the vocabulary size in each dataset. The edges (undirected) are formed by the citation relationship between articles. We use the same training/validation/test splits as in Kipf & Welling (2016). During training, only 20 instances per class are provided with corresponding labels.

We report the test classification accuracy in Figure 5. When possible, we include the baselines’ performances directly from previously published results (Kipf & Welling, 2016). From the figure we can see, the proposed SSE performs the best in Citeseer dataset, while being slightly worse than other GNN models in Cora and Pubmed dataset, respectively. We’ve also include the results that using the node feature with multi-layer perceptron (MLP). The MLP doesn’t consider any graph structure into consideration, which serves a sanity check.
Table 7. Multi-label classification in small datasets. We report both Micro-F1 and Macro-F1 on held-out test set.

<table>
<thead>
<tr>
<th></th>
<th>Blogcatalog</th>
<th>Wikipedia</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Micro-F1%</td>
<td>Macro-F1%</td>
</tr>
<tr>
<td>Methods</td>
<td>10% 20% 30% 40% 50% 60% 70% 80% 90%</td>
<td>10% 20% 30% 40% 50% 60% 70% 80% 90%</td>
</tr>
<tr>
<td>structure2vec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blogcatalog</td>
<td>Micro-F1%</td>
<td>Macro-F1%</td>
</tr>
<tr>
<td>Methods</td>
<td>10% 20% 30% 40% 50% 60% 70% 80% 90%</td>
<td>10% 20% 30% 40% 50% 60% 70% 80% 90%</td>
</tr>
<tr>
<td>structure2vec</td>
<td>35.05 36.65 38.43 39.35 40.48 40.89 42.56 42.58 42.61</td>
<td>19.78 22.39 23 25.16 25.89 26.96 26.86 27.46 27.69</td>
</tr>
<tr>
<td>GCN</td>
<td>36.80 38.42 39.47 40.88 40.88 41.69 42.06 42.43 42.50</td>
<td>19.31 20.96 20.43 22.3 21.86 22.14 23.06 23.2 23.43</td>
</tr>
<tr>
<td>SSE</td>
<td>33.90 36.42 36.80 37.39 37.91 37.92 38.58 39.10 40.28</td>
<td>19.88 22.68 22.88 23.89 23.9 24.08 24.38 25.12 24.99</td>
</tr>
<tr>
<td>PPI</td>
<td>Micro-F1%</td>
<td>Macro-F1%</td>
</tr>
<tr>
<td>Methods</td>
<td>10% 20% 30% 40% 50% 60% 70% 80% 90%</td>
<td>10% 20% 30% 40% 50% 60% 70% 80% 90%</td>
</tr>
<tr>
<td>structure2vec</td>
<td>19.86 23.19 24.73 25.46 25.29 27.79 27.75 28.32 28.99</td>
<td>15.14 15.94 18.32 18.41 19.04 20.41 20.56 22.01 23.83</td>
</tr>
</tbody>
</table>