N-GCN: Multi-scale Graph Convolution for Semi-supervised Node Classification

Sami Abu-El-Haija
Google Research
Mountain View, CA
haija@google.com

Amol Kapoor*
Google Research
Mountain View, CA
amol.kapoor@columbia.edu

Bryan Perozzi
Google Research
New York City, NY
bperozzi@acm.org

Joonseok Lee
Google Research
Mountain View, CA
joonseok@google.com

ABSTRACT

Graph Convolutional Networks (GCNs) have shown significant improvements in semi-supervised learning on graph-structured data. Concurrently, unsupervised learning of graph embeddings has benefited from the information contained in random walks. In this paper, we propose a model: Network of GCNs (N-GCN), which marries these two lines of work. At its core, N-GCN trains multiple instances of GCNs over node pairs discovered at different distances in random walks, and learns a combination of the instance outputs which optimizes the classification objective. Our experiments show that our proposed N-GCN model improves state-of-the-art baselines on all of the challenging node classification tasks we consider: Cora, Citeseer, Pubmed, and PPI. In addition, our proposed method has other desirable properties, including generalization to recently proposed semi-supervised learning methods such as GraphSAGE, allowing us to propose N-SAGE, and resilience to adversarial input perturbations.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Information systems** → **Social networks**;

KEYWORDS

Graph, Convolution, Spectral, Semi-Supervised Learning, Deep Learning

1 INTRODUCTION

Semi-supervised learning on graphs is important in many real-world applications, where the goal is to recover labels for all nodes given only a fraction of labeled ones. Some applications include social networks, where one wishes to predict user interests, or in health care, where one wishes to predict whether a patient should be screened for cancer. In many such cases, collecting node labels can be prohibitive. However, edges between nodes can be easier to obtain, either using an explicit graph (e.g. social network) or implicitly by calculating pairwise similarities [e.g. using a patient-patient similarity kernel, 19].

Convolutional Neural Networks [16] learn location-invariant hierarchical filters, enabling significant improvements on Computer Vision tasks [13, 15, 23]. This success has motivated researchers [8] to extend convolutions from spatial (i.e. regular lattice) domains to graph-structured (i.e. irregular) domains, yielding a class of algorithms known as Graph Convolutional Networks (GCNs).

Formally, we are interested in semi-supervised learning where we are given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $N = |\mathcal{V}|$ nodes; adjacency matrix A; and matrix $X \in \mathbb{R}^{N \times F}$ of node features. Labels for only a subset of nodes $\mathcal{V}_L \subset \mathcal{V}$ are observed. In general, $|\mathcal{V}_L| \ll |\mathcal{V}|$. Our goal is to recover labels for all unlabeled nodes $\mathcal{V}_U = \mathcal{V} - \mathcal{V}_L$, using the feature matrix X, the known labels for nodes in $\mathcal{V}_L$, and the graph G. In this setting, one treats the graph as the "unsupervised" and labels of $\mathcal{V}_L$ as the "supervised" portions of the data.

Depicted in Figure 1, our model for semi-supervised node classification builds on the GCN module proposed by Kipf and Welling [14], which operates on the normalized adjacency matrix $\hat{A}$, as in $\text{GCN}(\hat{A})$, where $\hat{A} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$, and D is diagonal matrix of node degrees. Our proposed extension of GCNs is inspired by the recent advancements in random walk based graph embeddings [e.g. 2, 10, 22]. We make a Network of GCN modules (N-GCN), feeding each module a different power of $\hat{A}$, as in $\{\text{GCN}(\hat{A}^0), \text{GCN}(\hat{A}^1), \text{GCN}(\hat{A}^2), \dots\}$. The k-th power contains statistics from the k-th step of a random walk on the graph. Therefore, our N-GCN model is able to combine information from various step-sizes (i.e. graph scales). We then combine the output of all GCN modules into a classification sub-network, and we jointly train all GCN modules and the classification sub-network on the upstream objective for semi-supervised node classification. Weights of the classification sub-network give us insight on how the N-GCN model works. For instance, in the presence of input perturbations, we observe that the classification sub-network weights shift towards GCN modules utilizing higher powers of the adjacency matrix, effectively widening the "receptive field" of the (spectral) convolutional filters. We achieve state-of-the-art on several semi-supervised graph learning tasks, showing that explicit random walks enhance the representational power of vanilla GCN's.

The rest of this paper is organized as follows. Section 2 reviews background work that provides the foundation for this paper. In

*Work was done while Amol was an intern at Google Research. He is returning to Google Research, full time, after completing his degree from Columbia University.

Section 3, we describe our proposed method, followed by experimental evaluation in Section 4. We compare our work with recent closely-related methods in Section 5. Finally, we conclude with our contributions and future work in Section 6.

2 BACKGROUND

2.1 Semi-Supervised Node Classification

Traditional label propagation algorithms [6, 24] learn a model that transforms node features into node labels and uses the graph to add a regularizer term:

$$\mathcal{L}_{\text{label.propagation}} = \mathcal{L}_{\text{classification}}\left(f(X), \mathcal{V}_L\right) + \lambda f(X)^T \Delta f(X), \quad (1)$$

The first term $\mathcal{L}_{\text{classification}}$ trains the model $f : \mathbb{R}^{N \times d_0} \rightarrow \mathbb{R}^{N \times C}$ to to predict the known labels $\mathcal{V}_L$. The second term is the graph-based regularizer, ensuring that connected nodes have a similar model output, with Δ being the graph Laplacian and $\lambda \in \mathbb{R}$ is the regularization coefficient hyperparameter.

2.2 Graph Convolutional Networks

Graph Convolution [8] generalizes convolution from Euclidean domains to graph-structured data. Convolving a "filter" over a signal on graph nodes can be calculated by transforming both the filter and the signal to the Fourier domain, multiplying them, and then transforming the result back into the discrete domain. The signal transform is achieved by multiplying with the eigenvectors of the graph Laplacian. The transformation requires a quadratic eigendecomposition of the symmetric Laplacian; however, the low-rank approximation of the eigendecomposition can be calculated using truncated Chebyshev polynomials [12]. For instance, [14] calculates a rank-1 approximation of the decomposition. They propose a multi-layer Graph Convolutional Networks (GCNs) for semi-supervised graph learning. Every layer computes the transformation:

$$H^{(l+1)} = \sigma\left(\hat{A} H^{(l)} W^{(l)}\right), \quad (2)$$

where $H^{(l)} \in \mathbb{R}^{N \times d_l}$ is the input activation matrix to the l-th hidden layer with row $H_i^{(l)}$ containing a d_l-dimensional feature vector for vertex $i \in \mathcal{V}$, and $W^{(l)} \in \mathbb{R}^{d_l \times d_{l+1}}$ is the layer's trainable weights. The first hidden layer $H^{(0)}$ is set to the input features X. A softmax on the last layer is used to classify labels. All layers use the same "normalized adjacency" $\hat{A}$, obtained by the "renormalization trick" utilized by [14], as $\hat{A} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$.[1]

Eq. (2) is a first order approximation of convolving filter $W^{(l)}$ over signal $H^{(l)}$ [12, 14]. The left-multiplication with $\hat{A}$ averages node features with their direct neighbors; this signal is then passed through a non-linearity function $\sigma(\cdot)$ (e.g, $\text{ReLU}(z) = \max(0, z)$). Successive layers effectively *diffuse* signals from nodes to neighbors.

Two-layer GCN model can be defined in terms of vertex features X and normalized adjacency $\hat{A}$ as:

$$\text{GCN}_{\text{2-layer}}(\hat{A}, X; \theta) = \text{softmax}\left(\hat{A}\sigma(\hat{A}XW^{(0)})W^{(1)}\right), \quad (3)$$

where the GCN parameters $\theta = \left\{W^{(0)}, W^{(1)}\right\}$ are trained to minimize the cross-entropy error over labeled examples. The output of the GCN model is a matrix $\mathbb{R}^{N \times C}$, where N is the number of nodes and C is the number of labels. Each row contains the label scores for one node, assuming there are C classes.

[1] with added self-connections added as $A_{ii} = 1$, similar to [14]

2.3 Graph Embeddings

Node Embedding methods represent graph nodes in a continuous vector space. They learn a dictionary $Z \in \mathbb{R}^{N \times d}$, with one d-dimensional embedding per node. Traditional methods use the adjacency matrix to learn embeddings. For example, Eigenmaps [5] performs the following constrained optimization:

$$\sum_{i,j} ||A_{ij}(Z_i - Z_J)|| \text{ s.t. } Z^T D Z = I, \quad (4)$$

where I is identity vector. Skipgram models on text corpora [20] inspired modern graph embedding methods, which simulate random walks to learn node embeddings [10, 22]. Each random walk generates a sequence of nodes. Sequences are converted to textual paragraphs, and are passed to a word2vec-style embedding learning algorithm [20]. As shown in [2], this learning-by-simulation is equivalent, in expectation, to the decomposition of a random walk co-occurrence statistics matrix $\mathcal{D}$. The expectation on $\mathcal{D}$ can be written as:

$$\mathbb{E}[\mathcal{D}] \propto \mathbb{E}_{q \sim \mathcal{Q}}\left[(\mathcal{T})^q\right] = \mathbb{E}_{q \sim \mathcal{Q}}\left[\left(D^{-1}A\right)^q\right], \quad (5)$$

where $\mathcal{T} = D^{-1}A$ is the row-normalized transition matrix (a.k.a right-stochastic adjacency matrix), and $\mathcal{Q}$ is a "context distribution" that is determined by random walk hyperparameters, such as the length of the random walk. The expectation therefore weights the importance of one node on another as a function of how well-connected they are, and the distance between them. The main difference between traditional node embedding methods and random walk methods is the optimization criteria: the former minimizes a loss on representing the adjacency matrix A (see Eq. 4), while the latter minimizes a loss on representing random walk co-occurrence statistics $\mathcal{D}$.

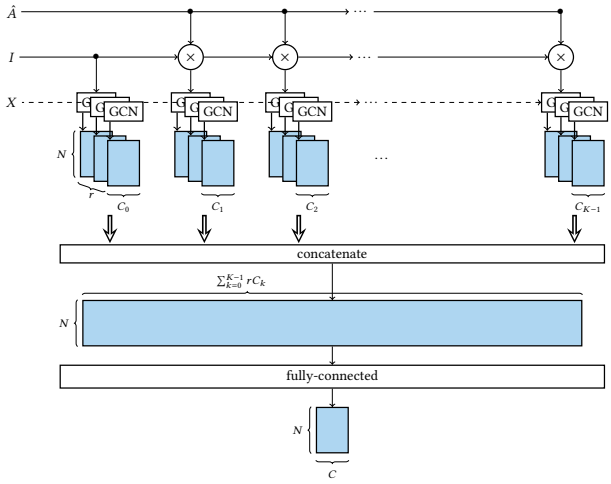
3 OUR METHOD

3.1 Motivation

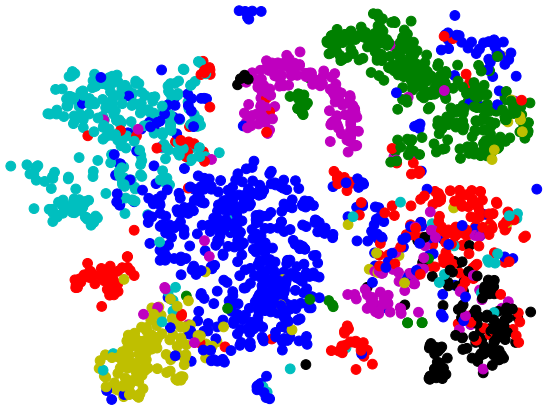
Graph Convolutional Networks and random walk graph embeddings are individually powerful. [14] uses GCNs for semi-supervised node classification. Instead of following traditional methods that use the graph for regularization (e.g. Eq. 4), [14] use the adjacency matrix for training and inference, effectively diffusing information across edges at all GCN layers (see Eq. 3). Separately, recent work has showed that random walk statistics can be very powerful for learning an unsupervised representation of nodes that can preserve the structure of the graph [2, 10, 22].

Under special conditions, it is possible for the GCN model to learn random walks. In particular, consider a two-layer GCN defined in Eq. 3 with the assumption that first-layer activation is identity as $\sigma(z) = z$, and weight $W^{(0)}$ is an identity matrix (either explicitly set or learned to satisfy the upstream objective). Under these two identity conditions, the model reduces to:

$$\begin{aligned}\text{GCN}_{\text{2-layer-special}}(\hat{A}, X) &= \text{softmax}\left(\hat{A}\hat{A}XW^{(1)}\right) \\ &= \text{softmax}\left(\hat{A}^2 XW^{(1)}\right)\end{aligned}$$



(a) N-GCN Architecture



(b) t-SNE visualization of fully-connected (fc) hidden layer of NGCN when trained over Cora graph.

Figure 1: Left: Model architecture, where $\hat{A}$ is the normalized normalized adjacency matrix, I is the identity matrix, X is node features matrix, and $\times$ is matrix-matrix multiply operator. We calculate K powers of the $\hat{A}$, feeding each power into r GCNs, along with X. The output of all $K \times r$ GCNs can be concatenated along the column dimension, then fed into fully-connected layers, outputting C channels per node, where C is size of label space. We calculate cross entropy error, between rows *prediction* $N \times C$ with known labels, and use them to update parameters of classification sub-network and all GCNs. Right: pre-relu activations after the first fully-connected layer of a 2-layer classification sub-network. Activations are PCA-ed to 50 dimensions then visualized using t-SNE.

where $\hat{A}^2$ can be expanded as:

$$\hat{A}^2 = \left(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}\right)\left(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}\right) = D^{-\frac{1}{2}}A\left[D^{-1}A\right]D^{-\frac{1}{2}}$$
$$= D^{-\frac{1}{2}}A\mathcal{T}D^{-\frac{1}{2}} \tag{6}$$

By multiplying the adjacency A with the transition matrix $\mathcal{T}$ before, the $\text{GCN}_{\text{2-layer-special}}$ is effectively doing a one-step random walk i.e. diffusing signals from nodes to neighbors, without non-linearities, then applying a non-linear Graph Conv layer.

3.2 Explicit Random Walks

The special conditions described above are not true in practice. Although stacking hidden GCN layers allows information to flow through graph edges, this flow is *indirect* as the information goes through feature reduction (matrix multiplication) and a non-linearity (activation function $\sigma(\cdot)$). Therefore, the vanilla GCN cannot directly learn high powers of $\hat{A}$, and could struggle with modeling information across distant nodes. We hypothesize that making the GCN directly operate on random walk statistics will allow the network to better utilize information across distant nodes, in the same way that node embedding methods (e.g. DeepWalk, [22]) operating on $\mathcal{D}$ are superior to traditional embedding methods operating on the adjacency matrix (e.g. Eigenmaps, [5]). Therefore, in addition to feeding only $\hat{A}$ to the GCN model as proposed by [14] (see Eq. 3), we propose to feed a K-degree polynomial of $\hat{A}$ to K instantiations of GCN. Generalizing Eq. (6) to arbitrary power k gives:

$$\hat{A}^k = D^{-\frac{1}{2}}A\mathcal{T}^{k-1}D^{-\frac{1}{2}}. \tag{7}$$

We also define $\hat{A}^0$ to be the identity matrix. Similar to [14], we add self-connections and convert directed graphs to undirected ones, making $\hat{A}$ and hence $\hat{A}^k$ symmetric matrices. The eigendecomposition of symmetric matrices is real. Therefore, the low-rank approximation of the eigendecomposition [12] is still valid, and a one layer of [14] utilizing $\hat{A}^k$ should still approximate multiplication in the Fourier domain.

3.3 Network of GCNs

Consider K instantiations of $\{\text{GCN}(\hat{A}^0, X), \text{GCN}(\hat{A}^1, X), \ldots, \text{GCN}(\hat{A}^{K-1}, X)\}$. Each GCN outputs a matrix $\mathbb{R}^{N \times C_k}$, where the v-th row describes a latent representation of that particular GCN for node $v \in \mathcal{V}$, and C_k is the latent dimensionality. Though C_k can be different for each GCN, we set all C_k to be the same for simplicity. We then combine the output of all K GCN and feed them into a classification sub-network, allowing us to jointly train all GCNs and the classification sub-network via backpropagation. This should allow the classification sub-network to choose features from the various GCNs, effectively allowing the overall model to learn a combination of features using the raw (normalized) adjacency, different steps of random walks (i.e. graph scales), and the input features X (as they are multiplied by identity $\hat{A}^0$).

3.3.1 Fully-Connected Classification Network. From a deep learning prospective, it is intuitive to represent the classification network as a fully-connected layer. We can concatenate the output of the K GCNs along the column dimension, i.e. concatenating all $\text{GCN}(X, \hat{A}^k)$, each $\in \mathbb{R}^{N \times C_k}$ into matrix $\in \mathbb{R}^{N \times C_K}$ where $C_K = \sum_k C_k$. We add a fully-connected layer $f_{\text{fc}} : \mathbb{R}^{N \times C_K} \to \mathbb{R}^{N \times C}$,

with trainable parameter matrix $W_{\text{fc}} \in \mathbb{R}^{CK \times C}$, written as:

$$\text{N-GCN}_{\text{fc}}(\hat{A}, A; W_{\text{fc}}, \theta) = \text{softmax}\Big(\Big[\text{GCN}(\hat{A}^0, X; \theta^{(0)}) \;\vdots\; \text{GCN}(\hat{A}^1, X; \theta^{(1)}) \;\vdots\; \dots \Big] W_{\text{fc}} \Big). \tag{8}$$

The classifier parameters W_{fc} are jointly trained with GCN parameters $\theta = \{\theta^{(0)}, \theta^{(1)}, \dots\}$. We use subscript fc on N-GCN to indicate the classification network is a fully-connected layer.

3.3.2 Attention Classification Network. We also propose a classification network based on "softmax attention", which learns a convex combination of the GCN instantiations. Our attention model (N-GCN_{a}) is parametrized by vector $\widetilde{m} \in \mathbb{R}^K$, one scalar for each GCN. It can be written as:

$$\text{N-GCN}_{\text{a}}(\hat{A}, X; m, \theta) = \sum_k m_k \text{GCN}(\hat{A}^k, X; \theta^{(k)}) \tag{9}$$

where m is output of a softmax: $m = \text{softmax}(\widetilde{m})$.

This softmax attention is similar to "Mixture of Experts" model, especially if we set the number of output channels for all GCNs equal to the number of classes, as in $C_0 = C_1 = \cdots = C$. This allows us to add cross entropy loss terms on all GCN outputs in addition to the loss applied at the output NGCN, forcing all GCN's to be independently useful. It is possible to set the $m \in \mathbb{R}^K$ parameter vector "by hand" using the validation split, especially for reasonable K such as $K \leq 6$. One possible choice might be setting m_0 to some small value and remaining $m_1, \dots, m_{K-1}$ to the harmonic series $\frac{1}{k}$; another choice may be linear decay $\frac{K-k}{K-1}$. These are respectively similar to the context distributions of GloVe [21] and word2vec [17, 20]. We note that if on average a node's information is captured by its direct or nearby neighbors, then the output of GCNs consuming lower powers of $\hat{A}$ should be weighted highly.

3.4 Training

We minimize the cross entropy between our model output and the known training labels Y as:

$$\min \text{diag}(\mathcal{V}_L) \left[Y \circ \log \text{N-GCN}(X, \hat{A}) \right], \tag{10}$$

where $\circ$ is Hadamard product, and $\text{diag}(\mathcal{V}_L)$ denotes a diagonal matrix, with entry at (i, i) set to 1 if $i \in \mathcal{V}_L$ and 0 otherwise. In addition, we can apply intermediate supervision for the NGCN_{a} to attempt make all GCN become independently useful, yielding minimization objective:

$$\min_{m,\theta} \text{diag}(\mathcal{V}_L) \Big[Y \circ \log \text{N-GCN}_{\text{a}}(\hat{A}, X; m, \theta) + \sum_k Y \circ \log \text{GCN}(\hat{A}^k, X; \theta^{(k)}) \Big]. \tag{11}$$

3.5 GCN Replication Factor r

To simplify notation, our N-GCN derivations (e.g. Eq. 8) assume that there is one GCN per $\hat{A}$ power. However, our implementation feeds every $\hat{A}$ to r GCN modules, as shown in Fig. 1.

3.6 Generalization to other Graph Models

In addition to vanilla GCNs [e.g. 14], our derivation also applies to other graph models including GraphSAGE [SAGE, 11]. Algorithm 1 shows a generalization that allows us to make a network of arbitrary graph models (e.g. GCN, SAGE, or others). Algorithms 2 and 3, respectively, show pseudo-code for the vanilla GCN [14] and GraphSAGE[2] [11]. Finally, Algorithm 4 defines our full Network of GCN model (N-GCN) by plugging Algorithm 2 into Algorithm 1. Similarly, Algorithm 5 defines our N-SAGE model by plugging Algorithm 3 in Algorithm 1.

We can recover the original algorithms GCN [14] and SAGE [11], respectively, by using Algorithms 4 (N-GCN) and 5 (N-SAGE) with $r = 1$, $K = 1$, identity ClassifierFn, and modifying line 2 in Algorithm 1 to $P \leftarrow \hat{A}$. Moreover, we can recover original DCNN [3] by calling Algorithm 4 with $L = 1$, $r = 1$, modifying line 3 to $\hat{A} \leftarrow D^{-1}A$, and keeping $K > 1$ as their proposed model operates on the power series of the transition matrix i.e. *unmodified* random walks, like ours.

4 EXPERIMENTS

4.1 Datasets

We experiment on three citation graph datasets: Pubmed, Citeseer, Cora, and a biological graph: Protein-Protein Interactions (PPI). We choose the aforementioned datasets because they are available online and are used by our baselines. The citation datasets are prepared by [25], and the PPI dataset is prepared by [11]. Table 1 summarizes dataset statistics.

Each node in the citation datasets represents an article published in the corresponding journal. An edge between two nodes represents a citation from one article to another, and a label represents the subject of the article. Each dataset contains a binary Bag-of-Words (BoW) feature vector for each node. The BoW are extracted from the article abstract. Therefore, the task is to predict the subject of articles, given the BoW of their abstract and the citations to other (possibly labeled) articles. Following [25] and [14], we use 20 nodes per class for training, 500 (overall) nodes for validation, and 1000 nodes for evaluation. We note that the validation set is larger than training $|\mathcal{V}_L|$ for these datasets!

The PPI graph, as processed and described by [11], consists of 24 disjoint subgraphs, each corresponding to a different human tissue. 20 of those subgraphs are used for training, 2 for validation, and 2 for testing, as partitioned by [11].

4.2 Baseline Methods

For the citation datasets, we copy baseline numbers from [14]. These include label propagation (LP, [26]); semi-supervised embedding (SemiEmb, [24]); manifold regularization (ManiReg, [7]); skip-gram graph embeddings [DeepWalk 22]; Iterative Classification Algorithm [ICA, 18]; Planetoid [25]; vanilla GCN [14]. For PPI, we copy baseline numbers from [11], which include GraphSAGE with LSTM aggregation (SAGE-LSTM) and GraphSAGE with pooling aggregation (SAGE). Further, for all datasets, we use our implementation to run baselines DCNN [3], GCN [14], and SAGE [with pooling

[2] Our implementation assumes mean-pool aggregation by [11], which performs on-par to their top performer max-pool aggregation. In addition, our Algorithm 3 lists a full-batch implementation whereas [11] offer a mini-batch implementation.

Algorithm 1 General Implementation: Network of Graph Models

Require: $\hat{A}$ is a normalization of A

function NETWORK(GRAPHMODELFN, $\hat{A}$, X, L, $r = 4$, $K = 6$, CLASSIFIERFN=FCLAYER)
  $P \leftarrow I$
  GraphModels $\leftarrow$ []
  for $k = 1$ to K **do**
    for $i = 1$ to r **do**
      GraphModels.append(GRAPHMODELFN(P, X, L))
    $P \leftarrow \hat{A}P$
  return CLASSIFIERFN(GraphModels)

Algorithm 2 GCN Model [14]

Require: $\hat{A}$ is a normalization of A

function GCNMODEL($\hat{A}$, X, L)
  $Z \leftarrow X$
  for $i = 1$ to L **do**
    $Z \leftarrow \sigma(\hat{A}ZW^{(i)})$
  return Z

Algorithm 3 SAGE Model [11]

Require: $\hat{A}$ is a normalization of A

function SAGEMODEL($\hat{A}$, X, L)
  $Z \leftarrow X$
  for $i = 1$ to L **do**
    $Z \leftarrow \sigma(\left[\begin{array}{c:c} Z & \hat{A}Z \end{array}\right] W^{(i)})$
    $Z \leftarrow$ L2NORMALIZEROWS(Z)
  return Z

Algorithm 4 N-GCN

function NGCN(A, X, $L = 2$)
  $D \leftarrow$ **diag**($A\mathbf{1}$) ▷ Sum rows
  $\hat{A} \leftarrow D^{-1/2}AD^{-1/2}$
  return NETWORK(GCNMODEL, $\hat{A}, X, L$)

Algorithm 5 N-SAGE

function NSAGE(A, X)
  $D \leftarrow$ **diag**($A\mathbf{1}$) ▷ Sum rows
  $\hat{A} \leftarrow D^{-1}A$
  return NETWORK(SAGEMODEL, $\hat{A}, X, 2$)

aggregation, 11], as these baselines can be recovered as special cases of our algorithm, as explained in Section 3.6.

4.3 Implementation

We use TensorFlow[1] to implement our methods, which we use to also measure the performance of baselines GCN, SAGE, and DCNN. For our methods and baselines, all GCN and SAGE modules that we train are 2 layers[3], where the first outputs 16 dimensions per node and the second outputs the number of classes (dataset-dependent). DCNN baseline has one layer and outputs 16 dimensions per node, and its channels (one per transition matrix power) are concatenated into a fully-connected layer that outputs the number of classes. We use 50% dropout and L2 regularization of 10^{-5} for all of the aforementioned models.

4.4 Node Classification Accuracy

Table 2 shows node classification accuracy results. We run 20 different random initializations for every model (baselines and ours), train using Adam optimizer [4] with learning rate of 0.01 for 600 steps, capturing the model parameters at peak validation accuracy to avoid overfitting. For our models, we sweep our hyperparameters r, K, and choice of classification sub-network $\in \{\text{fc}, \text{a}\}$. For baselines and our models, we choose the model with the highest accuracy on validation set, and use it to record metrics on the test set in Table 2.

Table 2 shows that N-GCN outperforms GCN [14] and N-SAGE improves on SAGE for all datasets, showing that *unmodified* random walks indeed help in semi-supervised node classification. Finally, our proposed models acheive state-of-the-art on all datasets.

4.5 Sensitivity Analysis

We analyze the impact of random walk length K and replication factor r on classification accuracy in Figure 2. In general, model performance improves when increasing K and r. We note utilizing random walks by setting $K > 1$ improves model accuracy due to the additional information, not due to increased model capacity: Contrast $K = 1, r > 1$ (i.e. mixture of GCNs, no random walks) with $K > 1, r = 1$ (i.e. N-GCN on random walks) – in both scenarios, the model has more capacity, but the latter shows better performance. The same holds for SAGE.

4.6 Tolerance to feature noise

We test our method under feature noise perturbations by removing node features at random. This is practical, as article authors might forget to include relevant terms in the article abstract, and more generally not all nodes will have the same amount of detailed information. Figure 3 shows that when features are removed, methods utilizing unmodified random walks: N-GCN, N-SAGE, and DCNN,

[3] except as clearly indicated in Table 4

Dataset	Type	Nodes $\lvert\mathcal{V}\rvert$	Edges $\lvert\mathcal{E}\rvert$	Classes C	Features F	Labeled nodes $\lvert\mathcal{V}_L\rvert$
Citeseer	citaction	3,327	4,732	6 (single class)	3,703	120
Cora	citaction	2,708	5,429	7 (single class)	1,433	140
Pubmed	citaction	19,717	44,338	3 (single class)	500	60
PPI	biological	56,944	818,716	121 (multi-class)	50	44,906

Table 1: Dataset used for experiments. For citation datasets, 20 training nodes per class are observed, with $\lvert\mathcal{V}_L\rvert = 20 \times C$

	Method	Citeseer	Cora	Pubmed	PPI
(a)	ManiReg [7]	60.1	59.5	70.7	–
(b)	SemiEmb [24]	59.6	59.0	71.1	–
(c)	LP [26]	45.3	68.0	63.0	–
(d)	DeepWalk [22]	43.2	67.2	65.3	–
(e)	ICA [18]	69.1	75.1	73.9	–
(f)	Planetoid [25]	64.7	75.7	77.2	–
(g)	GCN [14]	70.3	81.5	79.0	–
(h)	SAGE-LSTM [11]	–	–	–	61.2
(i)	SAGE [11]	–	–	–	60.0
(j)	DCNN (our implementation)	71.1	81.3	79.3	44.0
(k)	GCN (our implementation)	71.2	81.0	78.8	46.2
(l)	SAGE (our implementation)	63.5	77.4	77.6	59.8
(m)	N-GCN (ours)	**72.2**	**83.0**	**79.5**	46.8
(n)	N-SAGE (ours)	71.0	81.8	79.4	**65.0**

Table 2: Node classification performance (% accuracy for the first three, citation datasets, and f1 micro-averaged for multiclass PPI), using data splits of [14, 25] and [11]. We report the test accuracy corresponding to the run with the highest validation accuracy. Results in rows (a) through (g) are copied from [14], rows (h) and (i) from [11], and (j) through (l) are generated using our code since we can recover other algorithms as explained in Section 3.6. Rows (m) and (n) are our models. Entries with "–" indicate that authors from whom we copied results did not run on those datasets. Nonetheless, we run all datasets using our implementation of the most-competitive baselines.

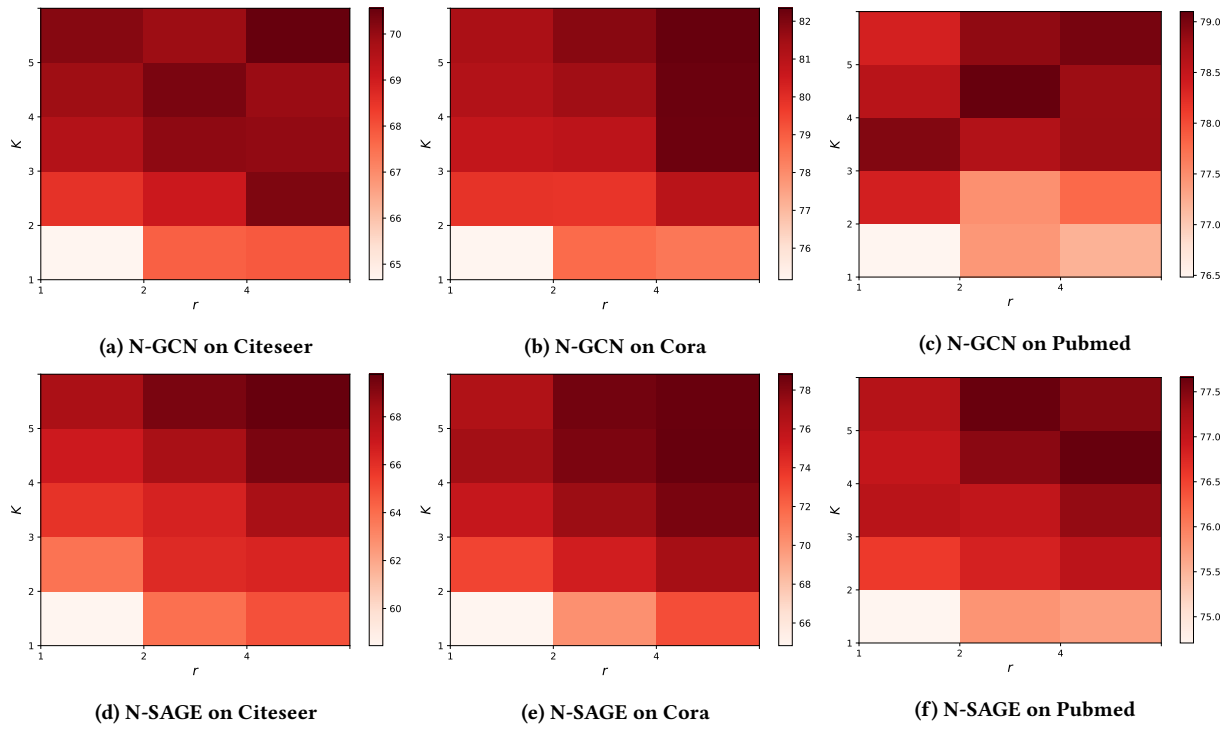
Nodes per class	5	10	20	100
DCNN (our implementation)	63.0 ± 1.0	72.3 ± 0.4	79.2 ± 0.2	82.6 ± 0.3
GCN (our implementation)	64.6 ± 0.3	70.0 ± 3.7	79.1 ± 0.3	81.8 ± 0.3
SAGE (our implementation)	69.0 ± 1.4	72.0 ± 1.3	77.2 ± 0.5	80.7 ± 0.7
N-GCN$_a$ (ours)	65.1 ± 0.7	71.2 ± 1.1	**79.7** ± 0.3	**83.0** ± 0.4
N-GCN$_{fc}$ (ours)	65.0 ± 2.1	71.7 ± 0.7	**79.7** ± 0.4	82.9 ± 0.3
N-SAGE$_a$ (ours)	66.9 ± 0.4	73.4 ± 0.7	79.0 ± 0.3	82.5 ± 0.2
N-SAGE$_{fc}$ (ours)	**70.7** ± 0.4	**74.1** ± 0.8	78.5 ± 1.0	81.8 ± 0.3

Table 3: Node classification accuracy (in %) for our largest dataset (Pubmed) as we vary size of training data $\frac{\lvert\mathcal{V}\rvert}{C} \in \{5, 10, 20, 100\}$. We report mean and standard deviations on 10 runs. We use a different random seed for every run (i.e. selecting different labeled nodes), but the same 10 random seeds across models. Convolution-based methods (e.g. SAGE) work well with few training examples, but *unmodified* random walk methods (e.g. DCNN) work well with more training data. Our methods combine convolution and random walks, making them work well in both conditions.

outperform convolutional methods including GCN and SAGE. Moreover, the performance gap widens as we remove more features. This suggests that our methods can somewhat recover removed features by *directly* pulling-in features from nearby and distant neighbors. We visualize in Figure 4 the attention weights as a function of % features removed. With little feature removal, there is some weight on $\hat{A}^0$, and the attention weights for $\hat{A}^1, \hat{A}^2, \ldots$ follow some decay function. Maliciously dropping features causes our model to shift its attention weights towards higher powers of $\hat{A}$.

4.7 Random Walk Steps Versus GCN Depth

K-step random walk will allow every node to accumulate information from its neighbors, up to distance K. Similarly, a K-layer GCN [14] will do the same. The difference between the two was



(a) N-GCN on Citeseer (b) N-GCN on Cora (c) N-GCN on Pubmed

(d) N-SAGE on Citeseer (e) N-SAGE on Cora (f) N-SAGE on Pubmed

Figure 2: Sensitivity Analysis. Model performance when varying random walk steps K and replication factor r. Best viewed with zoom. Overall, model performance increases with larger values of K and r. In addition, having random walk steps (larger K) boosts performance more than increasing model capacity (larger r).

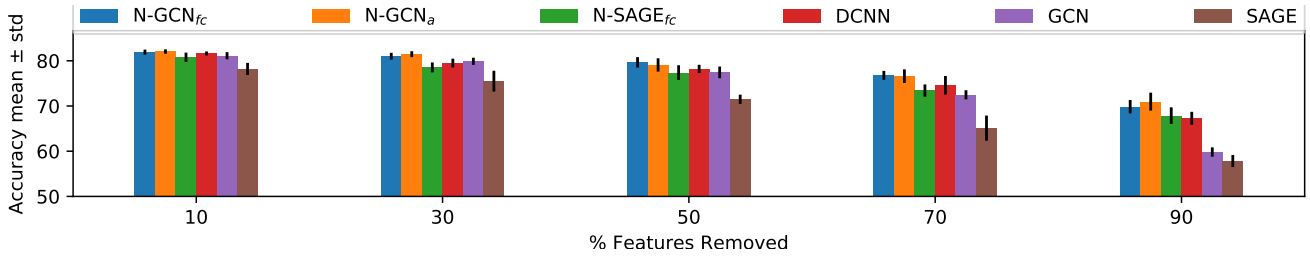


Figure 3: Classification accuracy for the Cora dataset with 20 labeled nodes per class ($|\mathcal{V}| = 20 \times C$), but features removed at random, averaging 10 runs. We use a different random seed for every run (i.e. removing different features per node), but the same 10 random seeds across models.

mathematically explained in Section 3.1. To summarize: the former averages node feature vectors according to the random walk co-visit statistics, whereas the latter creates non-linearities and matrix multiplies at every step. So far, we displayed experiments where our models (N-GCN and N-SAGE) were able to use information from distant nodes (e.g. $K = 5$), but for all GCN and SAGE modules, we used 2 GCN layer for baselines and our models.

Even though the authors of GCN [14] and SAGE [11] suggest using two GCN layers, according by holdout validation, for a fair comparison with our models, we run experiments utilizing deeper GCN and SAGE are models so that its "receptive field" is comparable to ours.

Table 4 shows test accuracies when training deeper GCN and SAGE models, using our implementation. We notice that, unlike our method which benefits from a wider "receptive field", there is no direct correspondence between depth and improved performance.

5 RELATED WORK

The field of graph learning algorithms is quickly evolving. We review work most similar to ours.

Defferrard et al [9] define graph convolutions as a K-degree polynomial of the Laplacian, where the polynomial coefficients are learned. In their setup, the K-th degree Laplacian is a sparse square matrix where entry at (i, j) will be zero if nodes i and j are

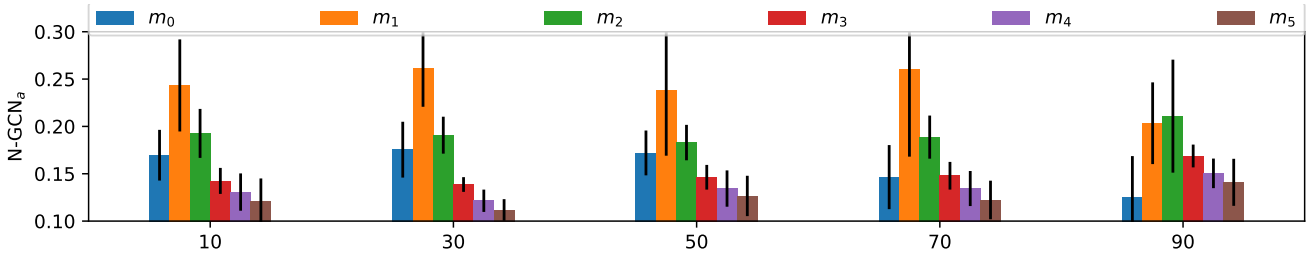


Figure 4: Attention weights (m) for N-GCN$_a$ when trained with feature removal perturbation on the Cora dataset. Removing features shifts the attention weights to the right, suggesting the model is relying more on long range dependencies.

Dataset	Model	$64 \times C$	$64 \times 64 \times C$	$64 \times 64 \times 64 \times C$
Citeseer	GCN	0.699	0.632	0.659
Citeseer	SAGE	0.668	0.660	0.674
Cora	GCN	0.803	0.800	0.780
Cora	SAGE	0.761	0.763	0.757
Pubmed	GCN	0.762	0.771	0.781
Pubmed	SAGE	0.770	0.776	0.775
PPI	GCN	0.460	0.461	0.466
PPI	SAGE	0.658	0.672	0.650

Table 4: Performance of deeper GCN and SAGE models, both using our implementation. Deeper GCN (or SAGE) does not consistently improve classification accuracy, suggesting that N-GCN and N-SAGE are more performant and are easier to train. They use shallower convolution models that operate on multiple scales of the graph.

more than K hops apart. Their sparsity analysis also applies here. A minor difference is the adjacency normalization. We use $\hat{A}$ whereas they use the Laplacian defined as $I - \hat{A}$. Raising $\hat{A}$ to power K will produce a square matrix with entry (i, j) being the probability of random walker ending at node i after K steps from node j. The major difference is the order of random walk versus non-linearity. In particular, their model calculates learns a linear combination of K-degree polynomial and pass through classifier function g, as in $g(\sum_k q_k \widetilde{A}^k)$, while our (e.g. N-GCN) model calculates $\sum_k q_k g(\widetilde{A}^k)$, where $\widetilde{A}$ is $\hat{A}$ in our model and $I - \hat{A}$ in theirs, and our g can be a GCN module. In fact, [9] is also similar to work by [2], as they both learn polynomial coefficients to some normalized adjacency matrix.

Atwood and Towsley [3] propose DCNN, which calculates powers of the transition matrix and keeps each power in a separate channel until the classification sub-network at the end. Their model is therefore similar to our work in that it also falls under $\sum_k q_k g(\widetilde{A}^k)$. However, where their model multiplies features with each power $\widetilde{A}^k$ once, our model makes use of GCN's [14] that multiply by $\widetilde{A}^k$ at every GCN layer (see Eq. 2). Thus, DCNN model [3] is a special case of ours, when GCN module contains only one layer, as explained in Section 3.6.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we propose a meta-model that can run arbitrary Graph Convolution models, such as GCN [14] and SAGE [11], on the output of random walks. Traditional Graph Convolution models operate on the normalized adjacency matrix. We make multiple instantiations of such models, feeding each instantiation a power of the adjacency matrix, and then concatenating the output of all instances into a classification sub-network. Each instantiation is therefore operating on different scale of the graph. Our model, Network of GCNs (and similarly, Network of SAGE), is end-to-end trainable, and is able to directly learn information across near or distant neighbors. We inspect the distribution of parameter weights in our classification sub-network, which reveal to us that our model is effectively able to circumvent adversarial perturbations on the input by shifting weights towards model instances consuming higher powers of the adjacency matrix. For future work, we plan to extend our methods to a stochastic implementation and tackle other (larger) graph datasets.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/

[2] Sami Abu-El-Haija, Bryan Perozzi, Rami Al-Rfou, and Alex Alemi. 2017. Watch Your Step: Learning Graph Embeddings Through Attention. In *arxiv*.

[3] James Atwood and Don Towsley. 2016. Diffusion-Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NIPS)*.

[4] Jimmy Ba and Diederik Kingma. 2015. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations*.

[5] Mikhail Belkin and Partha Niyogi. 2003. Laplacian Eigenmaps for Dimensionality Reduction and Data Representation. In *Neural Computation*.

[6] Mikhail Belkin, Partha Niyogi, and Vikas Sindhwani. 2006. Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. In *Journal of machine learning research (JMLR)*.

[7] Mikhail Belkin, Partha Niyogi, and Vikas Sindhwani. 2006. Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. In *Journal of machine learning research (JMLR)*.

[8] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. 2014. Spectral Networks and Locally Connected Networks on Graphs. In *International Conference on Learning Representations*.

[9] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Advances in Neural Information Processing Systems (NIPS)*.

[10] A. Grover and J. Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

[11] W. Hamilton, R. Ying, and J. Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NIPS*.

[12] David K. Hammond, Pierre Vandergheynst, and R. Gribonval. 2011. Wavelets on graphs via spectral graph theory. In *Applied and Computational Harmonic Analysis*.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[14] T. Kipf and M. Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*.

[16] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*.

[17] Omer Levy, Yoav Goldberg, and Ido Dagan. 2015. Improving Distributional Similarity with Lessons Learned from Word Embeddings. In *Transactions of the Association for Computational Linguistics (TACL)*.

[18] Qing Lu and Lise Getoor. 2003. Link-based classification. In *International Conference on Machine Learning (ICML)*.

[19] Selin Merdan, Christine L. Barnett, and Brian T. Denton. 2017. Data Analytics for Optimal Detection of Metastatic Prostate Cancer. (2017).

[20] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*.

[21] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global Vectors for Word Representation. In *Conference on Empirical Methods in Natural Language Processing, EMNLP*.

[22] B. Perozzi, R. Al-Rfou, and S. Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *Knowledge Discovery and Data Mining*.

[23] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[24] Jason Weston, Frederic Ratle, Hossein Mobahi, and Ronan Collobert. 2012. Deeplearning via semi-supervised embedding. In *Neural Networks: Tricks of the Trade*. 639–655.

[25] Z. Yang, W. Cohen, and R. Salakhutdinov. 2016. Revisiting Semi-Supervised Learning with Graph Embeddings. In *International Conference on Machine Learning (ICML)*.

[26] Xiaojin Zhu, Zoubin Ghahramani, and John Lafferty. 2003. Semi-supervised learning using gaussian fields and harmonic functions. In *International Conference on Machine Learning (ICML)*.