Hierarchical Graph Representation Learning with Differentiable Pooling

Rex Ying
rexying@stanford.edu
Stanford University

Jiaxuan You
jiaxuan@stanford.edu
Stanford University

Christopher Morris
christopher.morris@udo.edu
TU Dortmund University

Xiang Ren
xiangren@usc.edu
University of Southern California

William L. Hamilton
wleif@stanford.edu
Stanford University

Jure Leskovec
jure@cs.stanford.edu
Stanford University

Abstract

Recently, graph neural networks (GNNs) have revolutionized the field of graph representation learning through effectively learned node embeddings, and achieved state-of-the-art results in tasks such as node classification and link prediction. However, current GNN methods are inherently *flat* and do not learn *hierarchical* representations of graphs—a limitation that is especially problematic for the task of graph classification, where the goal is to predict the label associated with an entire graph. Here we propose DIFFPOOL, a differentiable graph pooling module that can generate hierarchical representations of graphs and can be combined with various graph neural network architectures in an end-to-end fashion. DIFFPOOL learns a differentiable soft cluster assignment for nodes at each layer of a deep GNN, mapping nodes to a set of clusters, which then form the coarsened input for the next GNN layer. Our experimental results show that combining existing GNN methods with DIFFPOOL yields an average improvement of 5–10% accuracy on graph classification benchmarks, compared to all existing pooling approaches, achieving a new state-of-the-art on four out of five benchmark data sets.

1 Introduction

In recent years there has been a surge of interest in developing graph neural networks (GNNs)—general deep learning architectures that can operate over graph structured data, such as social network data [16, 21, 36] or graph-based representations of molecules [7, 11, 15]. The general approach with GNNs is to view the underlying graph as a computation graph and learn neural network primitives that generate individual node embeddings by passing, transforming, and aggregating node feature information across the graph [15, 16]. The generated node embeddings can then be used as input to any differentiable prediction layer, e.g., for node classification [16] or link prediction [32], and the whole model can be trained in an end-to-end fashion.

However, a major limitation of current GNN architectures is that they are inherently *flat* as they only propagate information across the edges of the graph and are unable to infer and aggregate the information in a *hierarchical* way. For example, in order to successfully encode the graph structure of organic molecules, one would ideally want to encode the local molecular structure (e.g., individual

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

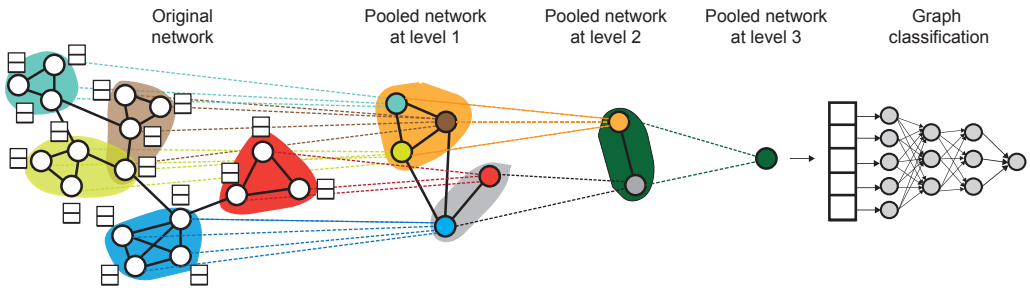


Figure 1: High-level illustration of our proposed method DIFFPOOL. At each hierarchical layer, we run a GNN model to obtain embeddings of nodes. We then use these learned embeddings to cluster nodes together and run another GNN layer on this coarsened graph. This whole process is repeated for L layers and we use the final output representation to classify the graph.

atoms and their direct bonds) as well as the coarse-grained structure of the molecular graph (e.g., groups of atoms and bonds representing functional units in a molecule). This lack of hierarchical structure is especially problematic for the task of graph classification, where the goal is to predict the label associated with an *entire graph*. When applying GNNs to graph classification, the standard approach is to generate embeddings for all the nodes in the graph and then to *globally pool* all these node embeddings together, e.g., using a simple summation or neural network that operates over sets [7, 11, 15, 25]. This global pooling approach ignores any hierarchical structure that might be present in the graph, and it prevents researchers from building effective GNN models for predictive tasks over entire graphs.

Here we propose DIFFPOOL, a differentiable graph pooling module that can be adapted to various graph neural network architectures in an hierarchical and end-to-end fashion (Figure 1). DIFFPOOL allows for developing deeper GNN models that can learn to operate on hierarchical representations of a graph. We develop a graph analogue of the spatial pooling operation in CNNs [23], which allows for deep CNN architectures to iteratively operate on coarser and coarser representations of an image. The challenge in the GNN setting—compared to standard CNNs—is that graphs contain no natural notion of spatial locality, i.e., one cannot simply pool together all nodes in a "$m \times m$ patch" on a graph, because the complex topological structure of graphs precludes any straightforward, deterministic definition of a "patch". Moreover, unlike image data, graph data sets often contain graphs with varying numbers of nodes and edges, which makes defining a general graph pooling operator even more challenging.

In order to solve the above challenges, we require a model that learns how to cluster together nodes to build a hierarchical multi-layer scaffold on top of the underlying graph. Our approach DIFFPOOL learns a differentiable soft assignment at each layer of a deep GNN, mapping nodes to a set of clusters based on their learned embeddings. In this framework, we generate deep GNNs by "stacking" GNN layers in a hierarchical fashion (Figure 1): the input nodes at the layer l GNN module correspond to the clusters learned at the layer $l-1$ GNN module. Thus, each layer of DIFFPOOL coarsens the input graph more and more, and DIFFPOOL is able to generate a hierarchical representation of any input graph after training. We show that DIFFPOOL can be combined with various GNN approaches, resulting in an average 7% gain in accuracy and a new state of the art on four out of five benchmark graph classification tasks. Finally, we show that DIFFPOOL can learn interpretable hierarchical clusters that correspond to well-defined communities in the input graphs.

2 Related Work

Our work builds upon a rich line of recent research on graph neural networks and graph classification.

General graph neural networks. A wide variety of graph neural network (GNN) models have been proposed in recent years, including methods inspired by convolutional neural networks [5, 8, 11, 16, 21, 24, 29, 36], recurrent neural networks [25], recursive neural networks [1, 30] and loopy belief propagation [7]. Most of these approaches fit within the framework of "neural message passing" proposed by Gilmer *et al.* [15]. In the message passing framework, a GNN is viewed as a

message passing algorithm where node representations are iteratively computed from the features of their neighbor nodes using a differentiable aggregation function. Hamilton *et al.* [17] provides a conceptual review of recent advancements in this area, and Bronstein *et al.* [4] outlines connections to spectral graph convolutions.

Graph classification with graph neural networks. GNNs have been applied to a wide variety of tasks, including node classification [16, 21], link prediction [31], graph classification [7, 11, 40], and chemoinformatics [28, 27, 14, 19, 32]. In the context of graph classification—the task that we study here—a major challenge in applying GNNs is going from node embeddings, which are the output of GNNs, to a representation of the entire graph. Common approaches to this problem include simply summing up or averaging all the node embeddings in a final layer [11], introducing a "virtual node" that is connected to all the nodes in the graph [25], or aggregating the node embeddings using a deep learning architecture that operates over sets [15]. However, all of these methods have the limitation that they do not learn hierarchical representations (i.e., all the node embeddings are globally pooled together in a single layer), and thus are unable to capture the natural structures of many real-world graphs. Some recent approaches have also proposed applying CNN architectures to the concatenation of all the node embeddings [29, 40], but this requires a specifying (or learning) a canonical ordering over nodes, which is in general very difficult and equivalent to solving graph isomorphism.

Lastly, there are some recent works that learn hierarchical graph representations by combining GNNs with deterministic graph clustering algorithms [8, 35, 13], following a two-stage approach. However, unlike these previous approaches, we seek to *learn* the hierarchical structure in an end-to-end fashion, rather than relying on a deterministic graph clustering subroutine.

3 Proposed Method

The key idea of DIFFPOOL is that it enables the construction of deep, multi-layer GNN models by providing a differentiable module to hierarchically pool graph nodes. In this section, we outline the DIFFPOOL module and show how it is applied in a deep GNN architecture.

3.1 Preliminaries

We represent a graph G as (A, F), where $A \in \{0,1\}^{n \times n}$ is the adjacency matrix, and $F \in \mathbb{R}^{n \times d}$ is the node feature matrix assuming each node has d features.[1] Given a set of labeled graphs $\mathcal{D} = \{(G_1, y_1), (G_2, y_2), ...\}$ where $y_i \in \mathcal{Y}$ is the label corresponding to graph $G_i \in \mathcal{G}$, the goal of graph classification is to learn a mapping $f : \mathcal{G} \to \mathcal{Y}$ that maps graphs to the set of labels. The challenge—compared to standard supervised machine learning setup—is that we need a way to extract useful feature vectors from these input graphs. That is, in order to apply standard machine learning methods for classification, e.g., neural networks, we need a procedure to convert each graph to an finite dimensional vector in $\mathbb{R}^D$.

Graph neural networks. In this work, we build upon graph neural networks in order to learn useful representations for graph classification in an end-to-end fashion. In particular, we consider GNNs that employ the following general "message-passing" architecture:

$$H^{(k)} = M(A, H^{(k-1)}; \theta^{(k)}), \tag{1}$$

where $H^{(k)} \in \mathbb{R}^{n \times d}$ are the node embeddings (i.e., "messages") computed after k steps of the GNN and M is the message propagation function, which depends on the adjacency matrix, trainable parameters $\theta^{(k)}$, and the node embeddings $H^{(k-1)}$ generated from the previous message-passing step.[2] The input node embeddings $H^{(0)}$ at the initial message-passing iteration $(k = 1)$, are initialized using the node features on the graph, $H^{(0)} = F$.

There are many possible implementations of the propagation function M [15, 16]. For example, one popular variant of GNNs—Kipf's *et al.* [21] Graph Convolutional Networks (GCNs)—implements M using a combination of linear transformations and ReLU non-linearities:

$$H^{(k)} = M(A, H^{(k-1)}; W^{(k)}) = \text{ReLU}(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(k-1)} W^{(k-1)}), \tag{2}$$

[1] We do not consider edge features, although one can easily extend the algorithm to support edge features using techniques introduced in [35].

[2] For notational convenience, we assume that the embedding dimension is d for all $H^{(k)}$; however, in general this restriction is not necessary.

where $\tilde{A} = A + I$, $\tilde{D} = \sum_j \tilde{A}_{ij}$ and $W^{(k)} \in \mathbb{R}^{d \times d}$ is a trainable weight matrix. The differentiable pooling model we propose can be applied to any GNN model implementing Equation (1), and is agnostic with regards to the specifics of how M is implemented.

A full GNN module will run K iterations of Equation (1) to generate the final output node embeddings $Z = H^{(K)} \in \mathbb{R}^{n \times d}$, where K is usually in the range 2-6. For simplicity, in the following sections we will abstract away the internal structure of the GNNs and use $Z = \text{GNN}(A, X)$ to denote an arbitrary GNN module implementing K iterations of message passing according to some adjacency matrix A and initial input node features X.

Stacking GNNs and pooling layers. GNNs implementing Equation (1) are inherently flat, as they only propagate information across edges of a graph. The goal of this work is to define a general, end-to-end differentiable strategy that allows one to *stack* multiple GNN modules in a hierarchical fashion. Formally, given $Z = \text{GNN}(A, X)$, the output of a GNN module, and a graph adjacency matrix $A \in \mathbb{R}^{n \times n}$, we seek to define a strategy to output a new coarsened graph containing $m < n$ nodes, with weighted adjacency matrix $A^{'} \in \mathbb{R}^{m \times m}$ and node embeddings $Z^{'} \in \mathbb{R}^{m \times d}$. This new coarsened graph can then be used as input to another GNN layer, and this whole process can be repeated L times, generating a model with L GNN layers that operate on a series of coarser and coarser versions of the input graph (Figure 1). Thus, our goal is to learn how to cluster or pool together nodes using the output of a GNN, so that we can use this coarsened graph as input to another GNN layer. What makes designing such a pooling layer for GNNs especially challenging—compared to the usual graph coarsening task—is that our goal is not to simply cluster the nodes in one graph, but to provide a general recipe to hierarchically pool nodes across a broad set of input graphs. That is, we need our model to learn a pooling strategy that will generalize across graphs with different nodes, edges, and that can adapt to the various graph structures during inference.

3.2 Differentiable Pooling via Learned Assignments

Our proposed approach, DIFFPOOL, addresses the above challenges by learning a cluster assignment matrix over the nodes using the output of a GNN model. The key intuition is that we stack L GNN modules and learn to assign nodes to clusters at layer l in an end-to-end fashion, using embeddings generated from a GNN at layer $l - 1$. Thus, we are using GNNs to both extract node embeddings that are useful for graph classification, as well to extract node embeddings that are useful for hierarchical pooling. Using this construction, the GNNs in DIFFPOOL learn to encode a general pooling strategy that is useful for a large set of training graphs. We first describe how the DIFFPOOL module pools nodes at each layer given an assignment matrix; following this, we discuss how we generate the assignment matrix using a GNN architecture.

Pooling with an assignment matrix. We denote the learned cluster assignment matrix at layer l as $S^{(l)} \in \mathbb{R}^{n_l \times n_{l+1}}$. Each row of $S^{(l)}$ corresponds to one of the n_l nodes (or clusters) at layer l, and each column of $S^{(l)}$ corresponds to one of the n_{l+1} clusters at the next layer $l + 1$. Intuitively, $S^{(l)}$ provides a soft assignment of each node at layer l to a cluster in the next coarsened layer $l + 1$.

Suppose that $S^{(l)}$ has already been computed, i.e., that we have computed the assignment matrix at the l-th layer of our model. We denote the input adjacency matrix at this layer as $A^{(l)}$ and denote the input node embedding matrix at this layer as $Z^{(l)}$. Given these inputs, the DIFFPOOL layer $(A^{(l+1)}, X^{(l+1)}) = \text{DIFFPOOL}(A^{(l)}, Z^{(l)})$ coarsens the input graph, generating a new coarsened adjacency matrix $A^{(l+1)}$ and a new matrix of embeddings $X^{(l+1)}$ for each of the nodes/clusters in this coarsened graph. In particular, we apply the two following equations:

$$X^{(l+1)} = {S^{(l)}}^T Z^{(l)} \in \mathbb{R}^{n_{l+1} \times d}, \tag{3}$$

$$A^{(l+1)} = {S^{(l)}}^T A^{(l)} S^{(l)} \in \mathbb{R}^{n_{l+1} \times n_{l+1}}. \tag{4}$$

Equation (3) takes the node embeddings $Z^{(l)}$ and aggregates these embeddings according to the cluster assignments $S^{(l)}$, generating embeddings for each of the n_{l+1} clusters. Similarly, Equation (4) takes the adjacency matrix $A^{(l)}$ and generates a coarsened adjacency matrix denoting the connectivity strength between each pair of clusters.

Through Equations (3) and (4), the DIFFPOOL layer coarsens the graph: the next layer adjacency matrix $A^{(l+1)}$ represents a coarsened graph with n_{l+1} nodes or *cluster nodes*, where each individual

cluster node in the new coarsened graph corresponds to a cluster of nodes in the graph at layer l. Note that $A^{(l+1)}$ is a real matrix and represents a fully connected edge-weighted graph; each entry $A^{(l+1)}_{ij}$ can be viewed as the connectivity strength between cluster i and cluster j. Similarly, the i-th row of $X^{(l+1)}$ corresponds to the embedding of cluster i. Together, the coarsened adjacency matrix $A^{(l+1)}$ and cluster embeddings $X^{(l+1)}$ can be used as input to another GNN layer, a process which we describe in detail below.

Learning the assignment matrix. In the following we describe the architecture of DIFFPOOL, i.e., how DIFFPOOL generates the assignment matrix $S^{(l)}$ and embedding matrices $Z^{(l)}$ that are used in Equations (3) and (4). We generate these two matrices using two separate GNNs that are both applied to the input cluster node features $X^{(l)}$ and coarsened adjacency matrix $A^{(l)}$. The *embedding GNN* at layer l is a standard GNN module applied to these inputs:

$$Z^{(l)} = \text{GNN}_{l,\text{embed}}(A^{(l)}, X^{(l)}), \tag{5}$$

i.e., we take the adjacency matrix between the cluster nodes at layer l (from Equation 4) and the pooled features for the clusters (from Equation 3) and pass these matrices through a standard GNN to get new embeddings $Z^{(l)}$ for the cluster nodes. In contrast, the *pooling GNN* at layer l, uses the input cluster features $X^{(l)}$ and cluster adjacency matrix $A^{(l)}$ to generate an assignment matrix:

$$S^{(l)} = \text{softmax}\left(\text{GNN}_{l,\text{pool}}(A^{(l)}, X^{(l)})\right), \tag{6}$$

where the softmax function is applied in a row-wise fashion. The output dimension of $\text{GNN}_{l,\text{pool}}$ corresponds to a pre-defined maximum number of clusters in layer l, and is a hyperparameter of the model.

Note that these two GNNs consume the same input data but have distinct parameterizations and play separate roles: The embedding GNN generates new embeddings for the input nodes at this layer, while the pooling GNN generates a probabilistic assignment of the input nodes to n_{l+1} clusters.

In the base case, the inputs to Equations (5) and Equations (6) at layer $l = 0$ are simply the input adjacency matrix A and the node features F for the original graph. At the penultimate layer $L-1$ of a deep GNN model using DIFFPOOL, we set the assignment matrix $S^{(L-1)}$ be a vector of 1's, i.e., all nodes at the final layer L are assigned to a single cluster, generating a final embedding vector corresponding to the entire graph. This final output embedding can then be used as feature input to a differentiable classifier (e.g., a softmax layer), and the entire system can be trained end-to-end using stochastic gradient descent.

Permutation invariance. Note that in order to be useful for graph classification, the pooling layer should be invariant under node permutations. For DIFFPOOL we get the following positive result, which shows that any deep GNN model based on DIFFPOOL is permutation invariant, as long as the component GNNs are permutation invariant.

Proposition 1. *Let $P \in \{0,1\}^{n \times n}$ be any permutation matrix, then* $\text{DIFFPOOL}(A, Z) = \text{DIFFPOOL}(PAP^T, PX)$ *as long as* $\textit{GNN}(A, X) = \textit{GNN}(PAP^T, X)$ *(i.e., as long as the GNN method used is permutation invariant).*

Proof. Equations (5) and (6) are permutation invariant by the assumption that the GNN module is permutation invariant. And since any permutation matrix is orthogonal, applying $P^T P = I$ to Equation (3) and (4) finishes the proof. □

3.3 Auxiliary Link Prediction Objective and Entropy Regularization

In practice, it can be difficult to train the pooling GNN (Equation 4) using only gradient signal from the graph classification task. Intuitively, we have a non-convex optimization problem and it can be difficult to push the pooling GNN away from spurious local minima early in training. To alleviate this issue, we train the pooling GNN with an auxiliary link prediction objective, which encodes the intuition that nearby nodes should be pooled together. In particular, at each layer l, we minimize $L_{\text{LP}} = ||A^{(l)}, S^{(l)} S^{(l)^T}||_F$, where $||\cdot||_F$ denotes the Frobenius norm. Note that the adjacency matrix $A^{(l)}$ at deeper layers is a function of lower level assignment matrices, and changes during training.

Another important characteristic of the pooling GNN (Equation 4) is that the output cluster assignment for each node should generally be close to a one-hot vector, so that the membership for each cluster or subgraph is clearly defined. We therefore regularize the entropy of the cluster assignment by minimizing $L_{\text{E}} = \frac{1}{n}\sum_{i=1}^{n} H(S_i)$, where H denotes the entropy function, and S_i is the i-th row of S.

During training, L_{LP} and L_{E} from each layer are added to the classification loss. In practice we observe that training with the side objective takes longer to converge, but nevertheless achieves better performance and more interpretable cluster assignments.

4 Experiments

We evaluate the benefits of DIFFPOOL against a number of state-of-the-art graph classification approaches, with the goal of answering the following questions:

- **Q1** How does DIFFPOOL compare to other pooling methods proposed for GNNs (e.g., using sort pooling [40] or the SET2SET method [15])?
- **Q2** How does DIFFPOOL combined with GNNs compare to the state-of-the-art for graph classification task, including both GNNs and kernel-based methods?
- **Q3** Does DIFFPOOL compute meaningful and interpretable clusters on the input graphs?

Data sets. To probe the ability of DIFFPOOL to learn complex hierarchical structures from graphs in different domains, we evaluate on a variety of relatively large graph data sets chosen from benchmarks commonly used in graph classification [20]. We use protein data sets including ENZYMES, PROTEINS [3, 12], D&D [10], the social network data set REDDIT-MULTI-12K [39], and the scientific collaboration data set COLLAB [39]. See Appendix A for statistics and properties. For all these data sets, we perform 10-fold cross-validation to evaluate model performance, and report the accuracy averaged over 10 folds.

Model configurations. In our experiments, the GNN model used for DIFFPOOL is built on top of the GRAPHSAGE architecture, as we found this architecture to have superior performance compared to the standard GCN approach as introduced in [21]. We use the "mean" variant of GRAPHSAGE [16] and apply a DIFFPOOL layer after every two GRAPHSAGE layers in our architecture. A total of 2 DIFFPOOL layers are used for the datasets. For small datasets such as ENZYMES, PROTEINS and COLLAB, 1 DIFFPOOL layer can achieve similar performance. After each DIFFPOOL layer, 3 layers of graph convolutions are performed, before the next DIFFPOOL layer, or the readout layer. The embedding matrix and the assignment matrix are computed by two separate GRAPHSAGE models respectively. In the 2 DIFFPOOL layer architecture, the number of clusters is set as 25% of the number of nodes before applying DIFFPOOL, while in the 1 DIFFPOOL layer architecture, the number of clusters is set as 10%. Batch normalization [18] is applied after every layer of GRAPHSAGE. We also found that adding an ℓ_2 normalization to the node embeddings at each layer made the training more stable. In Section 4.2, we also test an analogous variant of DIFFPOOL on the STRUCTURE2VEC [7] architecture, in order to demonstrate how DIFFPOOL can be applied on top of other GNN models. All models are trained for 3 000 epochs with early stopping applied when the validation loss starts to drop. We also evaluate two simplified versions of DIFFPOOL:

- DIFFPOOL-DET, is a DIFFPOOL model where assignment matrices are generated using a deterministic graph clustering algorithm [9].
- DIFFPOOL-NOLP is a variant of DIFFPOOL where the link prediction side objective is turned off.

4.1 Baseline Methods

In the performance comparison on graph classification, we consider baselines based upon GNNs (combined with different pooling methods) as well as state-of-the-art kernel-based approaches.

GNN-based methods.

- GRAPHSAGE with global mean-pooling [16]. Other GNN variants such as those proposed in [21] are omitted as empirically GraphSAGE obtained higher performance in the task.
- STRUCTURE2VEC (S2V) [7] is a state-of-the-art graph representation learning algorithm that combines a latent variable model with GNNs. It uses global mean pooling.
- Edge-conditioned filters in CNN for graphs (ECC) [35] incorporates edge information into the GCN model and performs pooling using a graph coarsening algorithm.

Table 1: Classification accuracies in percent. The far-right column gives the relative increase in accuracy compared to the baseline GRAPHSAGE approach.

	Method	Data Set					
		ENZYMES	D&D	REDDIT-MULTI-12K	COLLAB	PROTEINS	Gain
Kernel	GRAPHLET	41.03	74.85	21.73	64.66	72.91	
	SHORTEST-PATH	42.32	78.86	36.93	59.10	76.43	
	1-WL	53.43	74.02	39.03	78.61	73.76	
	WL-OA	60.13	79.04	44.38	80.74	75.26	
GNN	PATCHYSAN	–	76.27	41.32	72.60	75.00	4.17
	GRAPHSAGE	54.25	75.42	42.24	68.25	70.48	–
	ECC	53.50	74.10	41.73	67.79	72.65	0.11
	SET2SET	60.15	78.12	43.49	71.75	74.29	3.32
	SORTPOOL	57.12	79.37	41.82	73.76	75.54	3.39
	DIFFPOOL-DET	58.33	75.47	46.18	**82.13**	75.62	5.42
	DIFFPOOL-NOLP	61.95	79.98	46.65	75.58	76.22	5.95
	DIFFPOOL	**62.53**	**80.64**	**47.08**	75.48	**76.25**	**6.27**

- PATCHYSAN [29] defines a receptive field (neighborhood) for each node, and using a canonical node ordering, applies convolutions on linear sequences of node embeddings.
- SET2SET replaces the global mean-pooling in the traditional GNN architectures by the aggregation used in SET2SET [38]. Set2Set aggregation has been shown to perform better than mean pooling in previous work [15]. We use GRAPHSAGE as the base GNN model.
- SORTPOOL [40] applies a GNN architecture and then performs a single layer of soft pooling followed by 1D convolution on sorted node embeddings.

For all the GNN baselines, we use 10-fold cross validation numbers reported by the original authors when possible. For the GRAPHSAGE and SET2SET baselines, we use the base implementation and hyperparameter sweeps as in our DIFFPOOL approach. When baseline approaches did not have the necessary published numbers, we contacted the original authors and used their code (if available) to run the model, performing a hyperparameter search based on the original author's guidelines.

Kernel-based algorithms. We use the GRAPHLET [34], the SHORTEST-PATH [2], WEISFEILER-LEHMAN kernel (WL) [33], and WEISFEILER-LEHMAN OPTIMAL ASSIGNMENT KERNEL (WL-OA) [22] as kernel baselines. For each kernel, we computed the normalized gram matrix. We computed the classification accuracies using the C-SVM implementation of LIBSVM [6], using 10-fold cross validation. The C parameter was selected from $\{10^{-3}, 10^{-2}, \ldots, 10^{2}, 10^{3}\}$ by 10-fold cross validation on the training folds. Moreover, for WL and WL-OA we additionally selected the number of iteration from $\{0, \ldots, 5\}$.

4.2 Results for Graph Classification

Table 1 compares the performance of DIFFPOOL to these state-of-the-art graph classification baselines. These results provide positive answers to our motivating questions **Q1** and **Q2**: We observe that our DIFFPOOL approach obtains the highest average performance among all pooling approaches for GNNs, improves upon the base GRAPHSAGE architecture by an average of 6.27%, and achieves state-of-the-art results on 4 out of 5 benchmarks. Interestingly, our simplified model variant, DIFFPOOL-DET, achieves state-of-the-art performance on the COLLAB benchmark. This is because many collaboration graphs in COLLAB show only single-layer community structures, which can be captured well with pre-computed graph clustering algorithm [9]. One observation is that despite significant performance improvement, DIFFPOOL could be unstable to train, and there is significant variation in accuracy across different runs, even with the same hyperparameter setting. It is observed that adding the link predictioin objective makes training more stable, and reduces the standard deviation of accuracy across different runs.

Differentiable Pooling on STRUCTURE2VEC. DIFFPOOL can be applied to other GNN architectures besides GRAPHSAGE to capture hierarchical structure in the graph data. To further support answering **Q1**, we also applied DIFFPOOL on Structure2Vec (S2V). We ran experiments using S2V with three layer architecture, as reported in [7]. In the first variant, one DIFFPOOL layer is applied after the first layer of S2V, and two more S2V layers are stacked on top of the output of DIFFPOOL.

The second variant applies one DIFFPOOL layer after the first and second layer of S2V respectively. In both variants, S2V model is used to compute the embedding matrix, while GRAPHSAGE model is used to compute the assignment matrix.

Table 2: Accuracy results of applying DIFFPOOL to S2V.

Data Set	Method		
	S2V	S2V WITH 1 DIFFPOOL	S2V WITH 2 DIFFPOOL
ENZYMES	61.10	62.86	**63.33**
D&D	78.92	80.75	**82.07**

The results in terms of classification accuracy are summarized in Table 2. We observe that DIFFPOOL significantly improves the performance of S2V on both ENZYMES and D&D data sets. Similar performance trends are also observed on other data sets. The results demonstrate that DIFFPOOL is a general strategy to pool over hierarchical structure that can benefit different GNN architectures.

Running time. Although applying DIFFPOOL requires additional computation of an assignment matrix, we observed that DIFFPOOL did not incur substantial additional running time in practice. This is because each DIFFPOOL layer reduces the size of graphs by extracting a coarser representation of the graph, which speeds up the graph convolution operation in the next layer. Concretely, we found that GRAPHSAGE with DIFFPOOL was 12$\times$ faster than the GRAPHSAGE model with SET2SET pooling, while still achieving significantly higher accuracy on all benchmarks.

4.3 Analysis of Cluster Assignment in DIFFPOOL

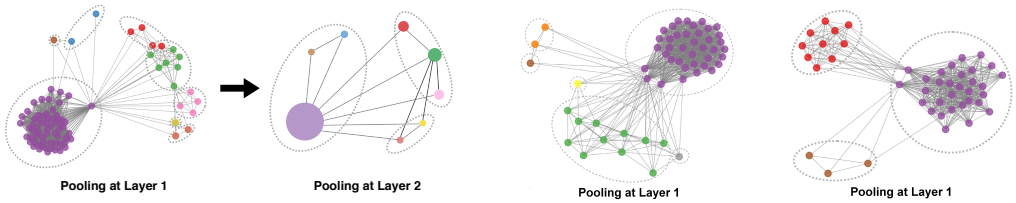
Hierarchical cluster structure. To address **Q3**, we investigated the extent to which DIFFPOOL learns meaningful node clusters by visualizing the cluster assignments in different layers. Figure 2 shows such a visualization of node assignments in the first and second layers on a graph from COLLAB data set, where node color indicates its cluster membership. Node cluster membership is determined by taking the argmax of its cluster assignment probabilities. We observe that even when learning cluster assignment based solely on the graph classification objective, DIFFPOOL can still capture the hierarchical community structure. We also observe significant improvement in membership assignment quality with link prediction auxiliary objectives.

Dense vs. sparse subgraph structure. In addition, we observe that DIFFPOOL learns to collapse nodes into soft clusters in a non-uniform way, with a tendency to collapse densely-connected subgraphs into clusters. Since GNNs can efficiently perform message-passing on dense, clique-like subgraphs (due to their small diameters) [26], pooling together nodes in such a dense subgraph is not likely to lead to any loss of structural information. This intuitively explains why collapsing dense subgraphs is a useful pooling strategy for DIFFPOOL. In contrast, sparse subgraphs may contain many interesting structures, including path-, cycle- and tree-like structures, and given the high-diameter induced by sparsity, GNN message-passing may fail to capture these structures. Thus, by separately pooling distinct parts of a sparse subgraph, DIFFPOOL can learn to capture the meaningful structures present in sparse graph regions (e.g., as in Figure 2).

Assignment for nodes with similar representations. Since the assignment network computes the soft cluster assignment based on features of input nodes and their neighbors, nodes with both similar input features and neighborhood structure will have similar cluster assignment. In fact, one can construct synthetic cases where 2 nodes, although far away, have exactly the same neighborhood structure and features for self and all neighbors. In this case the pooling network is forced to assign them into the same cluster, which is different from the concept of pooling in other architectures such as image ConvNets. In some cases we do observe that disconnected nodes are pooled together.

In practice we rely on the identifiability assumption similar to Theorem 1 in GraphSAGE [16], where nodes are identifiable via their features. This holds in many real datasets [3]. The auxiliary link prediction objective is observed to also help discouraging nodes that are far away to be pooled together. Furthermore, it is possible to use more sophisticated GNN aggregation function such as

[3] However, some chemistry molecular graph datasets contain many nodes that are structurally similar, and assignment network is observed to pool together nodes that are far away.



(a) (b) (c)

Figure 2: Visualization of hierarchical cluster assignment in DIFFPOOL, using example graphs from COLLAB. The left figure (a) shows hierarchical clustering over two layers, where nodes in the second layer correspond to clusters in the first layer. (Colors are used to connect the nodes/clusters across the layers, and dotted lines are used to indicate clusters.) The right two plots (b and c) show two more examples first-layer clusters in different graphs. Note that although we globally set the number of clusters to be 25% of the nodes, the assignment GNN automatically learns the appropriate number of meaningful clusters to assign for these different graphs.

high-order moments [37] to distinguish nodes that are similar in structure and feature space. The overall framework remains unchanged.

Sensitivity of the Pre-defined Maximum Number of Clusters. We found that the assignment varies according to the depth of the network and C, the maximum number of clusters. With larger C, the pooling GNN can model more complex hierarchical structure. The trade-off is that very large C results in more noise and less efficiency. Although the value of C is a pre-defined parameter, the pooling net learns to use the appropriate number of clusters by end-to-end training. In particular, some clusters might not be used by the assignment matrix. Column corresponding to unused cluster has low values for all nodes. This is observed in Figure 2(c), where nodes are assigned predominantly into 3 clusters.

5 Conclusion

We introduced a differentiable pooling method for GNNs that is able to extract the complex hierarchical structure of real-world graphs. By using the proposed pooling layer in conjunction with existing GNN models, we achieved new state-of-the-art results on several graph classification benchmarks. Interesting future directions include learning hard cluster assignments to further reduce computational cost in higher layers while also ensuring differentiability, and applying the hierarchical pooling method to other downstream tasks that require modeling of the entire graph structure.

Acknowledgement

This research has been supported in part by DARPA SIMPLEX, Stanford Data Science Initiative, Huawei, JD and Chan Zuckerberg Biohub. Christopher Morris is funded by the German Science Foundation (DFG) within the Collaborative Research Center SFB 876 “Providing Information by Resource-Constrained Data Analysis”, project A6 “Resource-efficient Graph Mining”. The authors also thank Marinka Zitnik for help in visualizing the high-level illustration of the proposed methods.

References

[1] M. Bianchini, M. Gori, and F. Scarselli. Processing directed acyclic graphs with recursive neural networks. *IEEE Transactions on Neural Networks*, 12(6):1464–1470, 2001.

[2] K. M. Borgwardt and H.-P. Kriegel. Shortest-path kernels on graphs. In *IEEE International Conference on Data Mining*, pages 74–81, 2005.

[3] K. M. Borgwardt, C. S. Ong, S. Schönauer, S. V. N. Vishwanathan, A. J. Smola, and H.-P. Kriegel. Protein function prediction via graph kernels. *Bioinformatics*, 21(Supplement 1):i47–i56, 2005.

[4] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[5] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and deep locally connected networks on graphs. In *International Conference on Learning Representations*, 2014.

[6] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[7] H. Dai, B. Dai, and L. Song. Discriminative embeddings of latent variable models for structured data. In *International Conference on Machine Learning*, pages 2702–2711, 2016.

[8] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pages 3844–3852, 2016.

[9] I. S. Dhillon, Y. Guan, and B. Kulis. Weighted graph cuts without eigenvectors a multilevel approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(11):1944–1957, 2007.

[10] P. D. Dobson and A. J. Doig. Distinguishing enzyme structures from non-enzymes without alignments. *Journal of Molecular Biology*, 330(4):771 – 783, 2003.

[11] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in Neural Information Processing Systems*, pages 2224–2232, 2015.

[12] A. Feragen, N. Kasenburg, J. Petersen, M. D. Bruijne, and K. M. Borgwardt. Scalable kernels for graphs with continuous attributes. In *Advances in Neural Information Processing Systems*, pages 216–224, 2013. Erratum available at http://image.diku.dk/aasa/papers/graphkernels_nips_erratum.pdf.

[13] M. Fey, J. E. Lenssen, F. Weichert, and H. Müller. SplineCNN: Fast geometric deep learning with continuous B-spline kernels. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[14] A. Fout, J. Byrd, B. Shariat, and A. Ben-Hur. Protein interface prediction using graph convolutional networks. In *Advances in Neural Information Processing Systems*, pages 6533–6542, 2017.

[15] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pages 1263–1272, 2017.

[16] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1025–1035, 2017.

[17] W. L. Hamilton, R. Ying, and J. Leskovec. Representation learning on graphs: Methods and applications. *IEEE Data Engineering Bulletin*, 40(3):52–74, 2017.

[18] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[19] W. Jin, C. W. Coley, R. Barzilay, and T. S. Jaakkola. Predicting organic reaction outcomes with Weisfeiler-Lehman network. In *Advances in Neural Information Processing Systems*, pages 2604–2613, 2017.

[20] K. Kersting, N. M. Kriege, C. Morris, P. Mutzel, and M. Neumann. Benchmark data sets for graph kernels, 2016.

[21] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

[22] N. M. Kriege, P.-L. Giscard, and R. Wilson. On valid optimal assignment kernels and applications to graph classification. In *Advances in Neural Information Processing Systems*, pages 1623–1631, 2016.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[24] T. Lei, W. Jin, R. Barzilay, and T. S. Jaakkola. Deriving neural architectures from sequence and graph kernels. In *International Conference on Machine Learning*, pages 2024–2033, 2017.

[25] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. In *International Conference on Learning Representations*, 2016.

[26] R. Liao, M. Brockschmidt, D. Tarlow, A. L. Gaunt, R. Urtasun, and R. Zemel. Graph partition neural networks for semi-supervised classification. In *International Conference on Learning Representations (Workshop Track)*, 2018.

[27] A. Lusci, G. Pollastri, and P. Baldi. Deep architectures and deep learning in chemoinformatics: The prediction of aqueous solubility for drug-like molecules. *Journal of Chemical Information and Modeling*, 53(7):1563–1575, 2013.

[28] C. Merkwirth and T. Lengauer. Automatic generation of complementary descriptors with molecular graph networks. *Journal of Chemical Information and Modeling*, 45(5):1159–1168, 2005.

[29] M. Niepert, M. Ahmed, and K. Kutzkov. Learning convolutional neural networks for graphs. In *International Conference on Machine Learning*, pages 2014–2023, 2016.

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

[31] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling. Modeling relational data with graph convolutional networks. In *Extended Semantic Web Conference*, 2018.

[32] K. Schütt, P. J. Kindermans, H. E. Sauceda, S. Chmiela, A. Tkatchenko, and K. R. Müller. SchNet: A continuous-filter convolutional neural network for modeling quantum interactions. In *Advances in Neural Information Processing Systems*, pages 992–1002, 2017.

[33] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-Lehman graph kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011.

[34] N. Shervashidze, S. V. N. Vishwanathan, T. H. Petri, K. Mehlhorn, and K. M. Borgwardt. Efficient graphlet kernels for large graph comparison. In *International Conference on Artificial Intelligence and Statistics*, pages 488–495, 2009.

[35] M. Simonovsky and N. Komodakis. Dynamic edge-conditioned filters in convolutional neural networks on graphs. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 29–38, 2017.

[36] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

[37] S. Verma and Z.-L. Zhang. Graph capsule convolutional neural networks. *arXiv preprint arXiv:1805.08090*, 2018.

[38] O. Vinyals, S. Bengio, and M. Kudlur. Order matters: Sequence to sequence for sets. In *International Conference on Learning Representations*, 2015.

[39] P. Yanardag and S. V. N. Vishwanathan. A structural smoothing framework for robust graph comparison. In *Advances in Neural Information Processing Systems*, pages 2134–2142, 2015.

[40] M. Zhang, Z. Cui, M. Neumann, and Y. Chen. An end-to-end deep learning architecture for graph classification. In *AAAI Conference on Artificial Intelligence*, 2018.

Relational Pooling for Graph Representations

Ryan L. Murphy [1] **Balasubramaniam Srinivasan** [2] **Vinayak Rao** [1] **Bruno Ribeiro** [2]

Abstract

This work generalizes graph neural networks (GNNs) beyond those based on the Weisfeiler-Lehman (WL) algorithm, graph Laplacians, and diffusions. Our approach, denoted Relational Pooling (RP), draws from the theory of finite partial exchangeability to provide a framework with maximal representation power for graphs. RP can work with existing graph representation models and, somewhat counterintuitively, can make them even more powerful than the original WL isomorphism test. Additionally, RP allows architectures like Recurrent Neural Networks and Convolutional Neural Networks to be used in a theoretically sound approach for graph classification. We demonstrate improved performance of RP-based graph representations over state-of-the-art methods on a number of tasks.

1. Introduction

Applications with relational graph data, such as molecule classification, social and biological network prediction, first order logic, and natural language understanding, require an effective representation of graph structures and their attributes. While representation learning for graph data has made tremendous progress in recent years, current schemes are unable to produce so-called *most-powerful* representations that can provably distinguish all distinct graphs up to graph isomorphisms. Consider for instance the broad class of Weisfeiler-Lehman (WL) based Graph Neural Networks (WL-GNNs) (Duvenaud et al., 2015; Kipf & Welling, 2017; Gilmer et al., 2017; Hamilton et al., 2017a; Velickovic et al., 2018; Monti et al., 2017; Ying et al., 2018; Xu et al., 2019; Morris et al., 2019). These are unable to distinguish pairs of nonisomorphic graphs on which the standard WL isomorphism heuristic fails (Cai et al., 1992; Xu et al., 2019; Morris et al., 2019). As graph neural networks (GNNs) are applied to increasingly more challenging problems, having a most-powerful framework for graph representation learning would be a key development in geometric deep learning (Bronstein et al., 2017).

In this work we introduce *Relational Pooling* (RP), a novel framework with maximal representation power for any graph input. In RP, we specify an idealized most-powerful representation for graphs and a framework for tractably approximating this ideal. The ideal representation can distinguish pairs of nonisomorphic graphs even when the WL isomorphism test fails, which motivates a straightforward procedure using approximate RP – we call this *RP-GNN* – for making GNNs more powerful.

A key inductive bias for graph representations is invariance to permutations of the adjacency matrix (graph isomorphisms), see Aldous (1981); Diaconis & Janson (2008); Orbanz & Roy (2015). Our work differs in its focus on learning representations of *finite but variable-size* graphs. In particular, given a finite but arbitrary-sized graph G potentially endowed with vertex or edge features, RP outputs a representation $\overline{\overline{f}}(G) \in \mathbb{R}^{d_h}$, $d_h > 0$, that is invariant to graph isomorphisms. RP can learn representations for each vertex in a graph, though to simplify the exposition, we focus on learning one representation of the entire graph.

Contributions. We make the following contributions: (1) We introduce *Relational Pooling* (RP), a novel framework for graph representation that can be combined with any existing neural network architecture, including ones not generally associated with graphs such as Recurrent Neural Networks (RNNs). (2) We prove that RP has maximal representation power for graphs and show that combining WL-GNNs with RP can increase their representation power. In our experiments, we classify graphs that cannot be distinguished by a state-of-the-art WL-GNN (Xu et al., 2019). (3) We introduce approximation approaches that make RP computationally tractable. We demonstrate empirically that these still lead to strong performance and can be used with RP-GNN to speed up graph classification when compared to traditional WL-GNNs.

2. Relational Pooling

Notation. We consider graphs endowed with vertex and edge features. That is, let $G = (V, E, \boldsymbol{X}^{(v)}, \mathbf{X}^{(e)})$ be a graph with vertices V, edges $E \subseteq V \times V$, vertex fea-

[1]Department of Statistics, and [2]Department of Computer Science, Purdue University, West Lafayette, Indiana, USA. Correspondence to: Ryan L. Murphy <murph213@purdue.edu>.

Proceedings of the 36^{th} International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019.

tures stored in a $|V| \times d_v$ matrix $\boldsymbol{X}^{(v)}$, and edge features stored in a $|V| \times |V| \times d_e$ tensor $\mathbf{X}^{(e)}$. W.l.o.g, we let $V := \{1, \ldots, n\}$, choosing some arbitrary ordering of the vertices. Unlike the vertex features $\boldsymbol{X}^{(v)}$, these vertex *labels* do not represent any meaningful information about the vertices, and learned graph representations should not depend upon the choice of ordering. Formally, there always exists a bijection on V (called a permutation or isomorphism) between orderings so we desire *permutation-invariant*, or equivalently, *isomorphic-invariant* functions.

In this work, we encode G by two data structures: (1) a $|V| \times |V| \times (1 + d_e)$ tensor that combines G's adjacency matrix with its edge features and (2) a $|V| \times d_v$ matrix representing node features $\boldsymbol{X}^{(v)}$. The tensor is defined as $\mathbf{A}_{v,u,\cdot} = \left[\mathbb{1}_{(v,u)\in E} \bowtie \boldsymbol{X}^{(e)}_{v,u}\right]$ for $v, u \in V$ where $[\cdot \bowtie \cdot]$ denotes concatenation along the 3rd mode of the tensor, $\mathbb{1}_{(\cdot)}$ denotes the indicator function, and $\boldsymbol{X}^{(e)}_{v,u}$ denotes the feature vector of edge (v, u) by a slight abuse of notation. A permutation is bijection $\pi : V \to V$ from the label set V to itself. If vertices are relabeled by a permutation π, we represent the new adjacency tensor by $\mathbf{A}_{\pi,\pi}$, where $(\mathbf{A}_{\pi,\pi})_{\pi(i),\pi(j),k} = \boldsymbol{A}_{i,j,k}\ \forall i, j \in V,\ k \in \{1, \ldots, 1 + d_e\}$; the index k over edge features is not permuted. Similarly, the vertex features are represented by $\boldsymbol{X}^{(v)}_{\pi}$ where $(\boldsymbol{X}^{(v)}_{\pi})_{\pi(i),l} = X^{(v)}_{i,l}$, $\forall i \in V$ and $l \in \{1, \ldots, d_v\}$. The Supplementary Material shows a concrete example and Kearnes et al. (2016) use a similar representation.

For bipartite graphs (e.g., consumers $\times$ products), V is partitioned by $V^{(\mathrm{r})}$ and $V^{(\mathrm{c})}$ and a separate permutation function can be defined on each. Their encoding is similar to the above and we define RP for the two different cases below.

Joint RP. Inspired by joint exchangeability (Aldous, 1981; Diaconis & Janson, 2008; Orbanz & Roy, 2015), we define a *joint RP* permutation-invariant function of non-bipartite graphs, whether directed or undirected, as

$$\overline{\overline{f}}(G) = \frac{1}{|V|!} \sum_{\pi \in \Pi_{|V|}} \vec{f}(\mathbf{A}_{\pi,\pi}, \boldsymbol{X}^{(v)}_{\pi}), \tag{1}$$

where $\Pi_{|V|}$ is the set of all distinct permutations of V and $\vec{f}$ is an arbitrary (possibly *permutation-sensitive*) function of the graph with codomain $\mathbb{R}^{d_h}$. Following Murphy et al. (2019), we use the notation $\overline{\cdot}$ to denote permutation-invariant function. Since Equation 1 averages over all permutations of the labels V, $\overline{\overline{f}}$ is a permutation-invariant function and can theoretically represent any such function $\overline{\overline{g}}$ (consider $\vec{f} = \overline{\overline{g}}$). We can compose $\overline{\overline{f}}$ with another function ρ (outside the summation) to capture additional signal in the graph. This can give a maximally expressive, albeit intractable, graph representation (Theorem 2.1). We later discuss tractable approximations for $\overline{\overline{f}}$ and neural network architectures for $\vec{f}$.

Separate RP. RP for bipartite graphs is motivated by *separate* exchangeability (Diaconis & Janson, 2008; Orbanz & Roy, 2015) and is defined as

$$\overline{\overline{f}}(G) = C \sum_{\pi \in \Pi_{|V^{(\mathrm{r})}|}} \sum_{\sigma \in \Pi_{|V^{(\mathrm{c})}|}} \vec{f}\left(\mathbf{A}_{\pi,\sigma}, \boldsymbol{X}^{(\mathrm{r},v)}_{\pi}, \boldsymbol{X}^{(\mathrm{c},v)}_{\sigma}\right) \tag{2}$$

where $C = (|V^{(\mathrm{r})}|!|V^{(\mathrm{c})}|!)^{-1}$ and π, σ are permutations of $V^{(\mathrm{r})}$, $V^{(\mathrm{c})}$, respectively. Results that apply to joint RP apply to separate RP.

2.1. Representation Power of RP

Functions $\overline{\overline{f}}$ should be *expressive* enough to learn distinct representations of nonisomorphic graphs or graphs with distinct features. We say $\overline{\overline{f}}(G)$ is *most-powerful* or *most-expressive* when $\overline{\overline{f}}(G) = \overline{\overline{f}}(G')$ iff G and G' are isomorphic and have the same vertex/edge features up to permutation. If $\overline{\overline{f}}$ is not most-powerful, a downstream function ρ may struggle to predict different classes for nonisormorphic graphs.

Theorem 2.1. *If node and edge attributes come from a finite set, then the representation $\overline{\overline{f}}(G)$ in Equation 1 is the most expressive representation of G, provided $\vec{f}$ is sufficiently expressive (e.g., a universal approximator).*

All proofs are shown in the Supplementary Material. This result provides a key insight into RP; one can focus on building expressive functions $\vec{f}$ that need not be permutation-invariant as the summation over permutations assures that permutation-invariance is satisfied.

2.2. Neural Network Architectures

Since $\vec{f}$ may be permutation sensitive, RP allows one to use a wide range of neural network architectures.

RNNs, MLPs. A valid architecture is to vectorize the graph (concatenating node and edge features, as illustrated in the Supplementary Material) and learn $\vec{f}$ over the resulting sequence. $\vec{f}$ can be an RNN, like an LSTM (Hochreiter & Schmidhuber, 1997) or GRU (Cho et al., 2014), or a feed-forward neural network (multilayer perceptron, MLP) with padding if different graphs have different sizes. Concretely,

$$\overline{\overline{f}}(G) = \frac{1}{|V|!} \sum_{\pi \in \Pi_{|V|}} \vec{f}\left(\mathrm{vec}(\mathbf{A}_{\pi,\pi}, \boldsymbol{X}^{(v)}_{\pi})\right).$$

CNNs. Convolutional neural networks (CNNs) can also be directly applied over the tensor $\mathbf{A}_{\pi,\pi}$ and combined with the node features $\boldsymbol{X}^{(v)}_{\pi}$, as in

$$\overline{\overline{f}}(G) = \frac{1}{|V|!} \sum_{\pi \in \Pi_{|V|}} \mathrm{MLP}\left(\left[\mathrm{CNN}(\mathbf{A}_{\pi,\pi}) \bowtie \mathrm{MLP}(\boldsymbol{X}^{(v)}_{\pi})\right]\right), \tag{3}$$

where CNN denotes a 2D (LeCun et al., 1989; Krizhevsky et al., 2012) if there are no edge features and a 3D CNN (Ji et al., 2013) if there are edge features, $[\cdot \bowtie \cdot]$ is a concatenation of the representations, and MLP is a multilayer perceptron. Multi-resolution 3D convolutions (Qi et al., 2016) can be used to map variable-sized graphs into the same sized representation for downstream layers.

GNNs. The function $\vec{f}$ can also be a graph neural network (GNN), a broad class of models that use the graph G itself to define the computation graph. These are permutation-invariant by design but we will show that their integration into RP can (1) make them more powerful and (2) speed up their computation via theoretically sound approximations. The GNNs we consider follow a message-passing (Gilmer et al., 2017) scheme defined by the recursion

$$\mathbf{h}_u^{(l)} = \phi^{(l)}\left(\mathbf{h}_u^{(l-1)}, \text{JP}\big((\mathbf{h}_v^{(l-1)})_{v\in\mathcal{N}(u)}\big)\right), \qquad (4)$$

where $\phi^{(l)}$ is a learnable function with distinct weights at each layer $1 \le l \le L$ of the computation graph, JP is a general (learnable) permutation-invariant function (Murphy et al., 2019), $\mathcal{N}(u)$ is the set of neighbors of $u \in V$, and $\mathbf{h}_u^{(l)} \in \mathbb{R}^{d_h^{(l)}}$ is a vector describing the embedding of node u at layer l. $\mathbf{h}_u^{(0)}$ is the feature vector of node u, $(\boldsymbol{X}^{(v)})_{u,\cdot}$ or can be assigned a constant c if u has no features. Under this framework, node embeddings can be used directly to predict node-level targets, or all node embeddings can be aggregated (via a learnable function) to form an embedding $\mathbf{h}_G$ used for graph-wide tasks.

There are several variations of Equation 4 in the literature. Duvenaud et al. (2015) proposed using embeddings from all layers $l \in \{1, 2, \ldots, L\}$ for graph classification. Hamilton et al. (2017a) used a similar framework for node classification and link prediction tasks, using the embedding at the last layer, while Xu et al. (2018) extend Hamilton et al. (2017a) to once again use embeddings at all layers for node and link prediction tasks. Other improvements include attention (Velickovic et al., 2018). This approach can be derived from spectral graph convolutions (e.g., (Kipf & Welling, 2017)). More GNNs are discussed in Section 3.

Recently, Xu et al. (2019); Morris et al. (2019) showed that these architectures are at most as powerful as the Weisfeiler-Lehman (WL) algorithm for testing graph isomorphism (Weisfeiler & Lehman, 1968), which itself effectively follows a message-passing scheme. Accordingly, we will broadly refer to models defined by Equation 4 as WL-GNNs. Xu et al. (2019) proposes a WL-GNN called *Graph Isomorphism Network* (GIN) which is as powerful as the WL test in graphs with discrete features.

Can a WL-GNN be more powerful than the WL test? WL-GNNs inherit a shortcoming from the WL test (Cai et al., 1992; Arvind et al., 2017; Fürer, 2017; Morris et al., 2019); node representations $\mathbf{h}_u^{(l)}$ do not encode whether two nodes have the *same neighbor* or *distinct neighbors with the same features*, limiting their ability to learn an expressive representation of the entire graph. Consider a task where graphs represent molecules, where node features indicate atom type and edges denote the presence or absence of bonds. Here, the first WL-GNN layer cannot distinguish that two (say) carbon atoms have a bond with the same carbon atom or a bond to two distinct carbon atoms. Successive layers of the WL-GNN update node representations and the hope is that nodes eventually get unique representations (up to isomorphisms), and thus allow the WL-GNN to detect whether two nodes have the same neighbor based on the representations of their neighbors. However, if there are too few WL-GNN layers or complex cycles in the graph, the graph and its nodes will not be adequately represented.

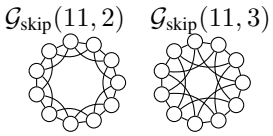


Figure 1: The WL test incorrectly deems these isomorphic.

To better understand this challenge, consider the extreme case illustrated by the two graphs in Figure 1. These are cycle graphs with $M = 11$ nodes where nodes that are $R \in \{2, 3\}$ 'hops' around the circle are connected by an edge. These highly symmetric graphs, which are special cases of circulant graphs (Vilfred, 2004) are formally defined in Definition 2.1 but the key point is that the WL test, and thus WL-GNNs, cannot distinguish these two nonisomorphic graphs.

Definition 2.1: [Circulant Skip Links (CSL) graphs] Let R and M be co-prime natural numbers[1] such that $R < M - 1$. $\mathcal{G}_{\text{skip}}(M, R)$ denotes an undirected 4-regular graph with vertices $\{0, 1, \ldots, M-1\}$ whose edges form a cycle and have skip links. That is, for the cycle, $\{j, j+1\} \in E$ for $j \in \{0, \ldots, M-2\}$ and $\{M-1, 0\} \in E$. For the skip links, recursively define the sequence $s_1 = 0$, $s_{i+1} = (s_i + R) \bmod M$ and let $\{s_i, s_{i+1}\} \in E$ for any $i \in \mathbb{N}$. $\lozenge$

We will use RP to help WL-GNNs overcome this shortcoming. Let $\vec{f}$ be a WL-GNN that we make permutation sensitive by assigning each node an identifier that depends on π. Permutation sensitive IDs prevent the RP sum from collapsing to just one term but more importantly help distinguish neighbors that otherwise appear identical. In particular, given any $\pi \in \Pi_{|V|}$, we append to the rows of $\boldsymbol{X}_\pi^{(v)}$ one-hot encodings of the row number before computing $\vec{f}$. We can represent this by an augmented vertex attribute ma-

[1] Two numbers are co-primes if their only common factor is 1.

trix $\left[\boldsymbol{X}_\pi^{(v)} \bowtie I_{|V|}\right]$ for every $\pi \in \Pi_{|V|}$, where $I_{|V|}$ is a $|V| \times |V|$ identity matrix and $[B \bowtie C]$ concatenates the columns of matrices B and C. RP-GNN is then given by

$$\overline{\overrightarrow{f}}(G) = \frac{1}{|V|!} \sum_{\pi \in \Pi_{|V|}} \overrightarrow{f}\left(\mathbf{A}_{\pi,\pi}, \left[\boldsymbol{X}_\pi^{(v)} \bowtie I_{|V|}\right]\right) \quad (5)$$
$$= \frac{1}{|V|!} \sum_{\pi \in \Pi_{|V|}} \overrightarrow{f}\left(\mathbf{A}, \left[\boldsymbol{X}^{(v)} \bowtie (I_{|V|})_\pi\right]\right),$$

where the second holds since $\overrightarrow{f}$ a GNN and thus invariant to permutations of the adjacency matrix. The following theorem shows that $\overline{\overrightarrow{f}}(G)$ in Equation 5 is strictly more *expressive* than the original WL-GNN; it can distinguish all nodes and graphs that WL-GNN can in addition to graphs that the original WL-GNN cannot.

Theorem 2.2. *The RP-GNN in Equation 5 is strictly more expressive than the original WL-GNN. Specifically, if $\overrightarrow{f}$ is a GIN (Xu et al., 2019) and the graph has discrete attributes, its RP-GNN is more powerful than the WL test.*

Equation 5 is computationally expensive but can be made tractable while retaining expressive power over standard GNNs. While all approximations discussed in Section 2.3 for RP in general are applicable to RP-GNN, a specific strategy is to assign permutation-sensitive node IDs in a clever way. In particular, if vertex features are available, we only need to assign enough IDs to make all vertices unique and thereby reduce the number of permutations we need to evaluate. For example, in the molecule CH_2O_2, if we create node features with one-hot IDs $(C,0,1)$,$(H,0,1)$,$(H,1,0)$,$(O,0,1)$,$(O,1,0)$, then we need only consider $1! \cdot 2! \cdot 2! = 4$ permutations. For unattributed graphs, we assign $i \bmod m$ to node i; setting m=1 reduces to a GNN and m=$|V|$ is the most expressive. More examples are in the Supplementary Material.

2.3. RP Tractability

2.3.1. Tractability via canonical orientations

Equation 1 is intractable as written and calls for approximations. The most direct approximation is to compose a permutation-sensitive $\overrightarrow{f}$ with a canonical orientation function that re-orders $\mathbf{A}$ such that CANONICAL$(\mathbf{A}, \boldsymbol{X}^{(v)})$= CANONICAL$(\mathbf{A}_{\pi,\pi}, \boldsymbol{X}_\pi^{(v)})$, $\forall \pi \in \Pi_{|V|}$. For instance, vertices can be sorted by centrality scores with some tie-breaking scheme (Montavon et al., 2012; Niepert et al., 2016). This causes the sum over all permutations to collapse to just an evaluation of $\overrightarrow{f} \circ$ CANONICAL. Essentially, this introduces a fixed component into the permutation-invariant function $\overline{\overrightarrow{f}}$ with only the second stage learned from data. This simplifying approximation to the original problem is however only useful if CANONICAL is related to the true function, and can otherwise result in poor representations (Murphy et al., 2019).

A more flexible approach collapses the set of all permutations into a smaller set of *equivalent permutations* which we denote as *poly-canonical orientation*. Depth-First Search (DFS) and Breadth-First Search (BFS) serve as two examples. In a DFS, the nodes of the adjacency matrix/tensor $\mathbf{A}_{\pi,\pi}$ are ordered from 1 to $|V|$ according to the order they are visited by a DFS starting at $\pi(1)$. Thus, if G is a length-three path and we consider permutation functions defined (elementwise) as $\pi(1,2,3) = (1,2,3)$, $\pi'(1,2,3) = (1,3,2)$, DFS or BFS would see respectively ①-②-③ and ①-③-② (where vertices are numbered by permuted indices), start at $\pi(1)$=1 and result in the same 'left-to-right' orientation for both permutations. In disconnected graphs, the search starts at the first node of each connected component. Learning orientations from data is a discrete optimization problem left for future work.

2.3.2. Tractability via π-SGD

A simple approach for making RP tractable is to sample *random* permutations during training. This offers the computational savings of a single canonical ordering but circumvents the need to learn a good canonical ordering for a given task. This approach is only approximately invariant, a tradeoff we make for the increased power of RP.

For simplicity, we analyze a supervised graph classification setting with a single sampled permutation, but this can be easily extended to sampling multiple permutations and unsupervised settings. Further, we focus on joint invariance but the formulation is similar for separate invariance. Consider N training data examples $\mathcal{D} \equiv \{(G(1), \boldsymbol{y}(1)), \ldots, (G(N), \boldsymbol{y}(N))\}$, where $\boldsymbol{y}(i) \in \mathbb{Y}$ is the target output and graph $G(i)$ its corresponding graph input. For a parameterized function $\overrightarrow{f}$ with parameters $\boldsymbol{W}$,

$$\overline{\overrightarrow{f}}(G(i); \boldsymbol{W}) = \frac{1}{|V(i)|!} \sum_{\pi \in \Pi_{|V(i)|}} \overrightarrow{f}(\mathbf{A}_{\pi,\pi}(i), \boldsymbol{X}_\pi^{(v)}(i); \boldsymbol{W}),$$

our (original) goal is to minimize the empirical loss

$$\overline{\overline{L}}(\mathcal{D}; \boldsymbol{W}) = \sum_{i=1}^{N} L\left(\boldsymbol{y}(i), \overline{\overrightarrow{f}}(G(i); \boldsymbol{W})\right), \quad (6)$$

where L is a convex loss function of $\overline{\overrightarrow{f}}(\cdot;\cdot)$ such as cross-entropy or square loss. For each graph $G(i)$, we sample a permutation $\mathbf{s}_i \sim \text{Unif}(\Pi_{|V(i)|})$ and replace the sum in Equation 1 with the estimate

$$\hat{\overline{\overrightarrow{f}}}(G(i); \boldsymbol{W}) = \overrightarrow{f}(\mathbf{A}_{\mathbf{s}_i,\mathbf{s}_i}(i), \boldsymbol{X}_{\mathbf{s}_i}^{(v)}(i); \boldsymbol{W}). \quad (7)$$

For separate invariance, we would sample a distinct permutation for each set of vertices. The estimator in Equation 7

is unbiased: $E_{\mathbf{s}_i}[\hat{\overline{\vec{f}}}(G_{\mathbf{s}_i,\mathbf{s}_i}(i);\boldsymbol{W})] = \overline{\vec{f}}(G(i);\boldsymbol{W})$, where $G_{\mathbf{s}_i,\mathbf{s}_i}$ is shorthand for a graph that has been permuted by $\mathbf{s}_i$. However, this is no longer true when $\overline{\vec{f}}$ is chained with a nonlinear loss L: $E_{\mathbf{s}_i}[L(\boldsymbol{y}(i), \hat{\overline{\vec{f}}}(G_{\mathbf{s}_i,\mathbf{s}_i}(i);\boldsymbol{W}))] \neq L(\boldsymbol{y}(i), E_{\mathbf{s}_i}[\hat{\overline{\vec{f}}}(G_{\mathbf{s}_i,\mathbf{s}_i}(i);\boldsymbol{W})])$. Nevertheless, as we will soon justify, we follow Murphy et al. (2019) and use this estimate in our optimization.

Definition 2.2: [π-SGD for RP] Let $\mathcal{B}_t = \{(G(1), \boldsymbol{y}(1)), \ldots, (G(B), \boldsymbol{y}(B))\}$ be a mini-batch i.i.d. sampled uniformly from the training data $\mathcal{D}$ at step t. To train RP with π-SGD, we follow the stochastic gradient descent update

$$\boldsymbol{W}_t = \boldsymbol{W}_{t-1} - \eta_t \mathbf{Z}_t, \tag{8}$$

where $\mathbf{Z}_t = \frac{1}{B}\sum_{i=1}^{B} \nabla_{\boldsymbol{W}} L\left(\boldsymbol{y}(i), \hat{\overline{\vec{f}}}(G(i);\boldsymbol{W}_{t-1})\right)$ is the random gradient with the random permutations $\{\mathbf{s}_i\}_{i=1}^{B}$, (sampled independently $\mathbf{s}_i \sim \text{Unif}(\Pi_{|V(i)|})$ for all graphs $G(i)$ in batch $\mathcal{B}_t$), and the learning rate is $\eta_t \in (0,1)$ s.t. $\lim_{t\to\infty}\eta_t = 0$, $\sum_{t=1}^{\infty}\eta_t = \infty$, and $\sum_{t=1}^{\infty}\eta_t^2 < \infty$. $\lozenge$

Effectively, this is a Robbins-Monro stochastic approximation algorithm of gradient descent (Robbins & Monro, 1951; Bottou, 2012) and optimizes the modified objective

$$\begin{aligned}\overline{\vec{J}}(\mathcal{D};\boldsymbol{W}) &= \frac{1}{N}\sum_{i=1}^{N} E_{\mathbf{s}_i}\left[L\left(\boldsymbol{y}(i), \hat{\overline{\vec{f}}}(G_{\mathbf{s}_i,\mathbf{s}_i}(i);\boldsymbol{W})\right)\right] \\ &= \frac{1}{N}\sum_{i=1}^{N}\frac{1}{|V(i)|!}\sum_{\pi\in\Pi_{|V(i)|}} L\left(\boldsymbol{y}(i), \hat{\overline{\vec{f}}}(G_{\pi,\pi}(i);\boldsymbol{W})\right).\end{aligned} \tag{9}$$

Observe that the expectation over permutations is now outside the loss function (recall $\overline{\vec{f}}(G(i);\boldsymbol{W})$ in in Equation 6 is an expectation). The loss in Equation 9 is also permutation-invariant, but π-SGD yields a result sensitive to the random input permutations presented to the algorithm. Further, unless the function $\vec{f}$ itself is permutation-invariant ($\overline{\vec{f}} = \vec{f}$), the optima of $\overline{\vec{J}}$ are different from those of the original objective function $\overline{\vec{L}}$. Instead, if L is convex in $\overline{\vec{f}}(\cdot;\cdot)$, $\overline{\vec{J}}$ is an upper bound to $\overline{\vec{L}}$ via Jensen's inequality, and minimizing this bound forms a tractable surrogate to the original objective in Equation 6.

The following convergence result follows from the π-SGD formulation of Murphy et al. (2019).

Proposition 2.1. *π-SGD stochastic optimization enjoys properties of almost sure convergence to optimal $\boldsymbol{W}$ under conditions similar to SGD (listed in Supplementary).*

Remark 2.1. *Given fixed point $\boldsymbol{W}^\star$ of the π-SGD optimization and a new graph G at test time, we may exactly compute $E_{\mathbf{s}}[\hat{\overline{\vec{f}}}(G_{\mathbf{s},\mathbf{s}};\boldsymbol{W}^\star)] = \overline{\vec{f}}(G;\boldsymbol{W}^\star)$ or estimate it with $\frac{1}{m}\sum_{j=1}^{m}\vec{f}(G_{\mathbf{s}_j,\mathbf{s}_j};\boldsymbol{W}^\star)$, where $\mathbf{s}_1\ldots,\mathbf{s}_m \overset{\text{i.i.d.}}{\sim} \text{Unif}(\Pi_{|V|})$.*

2.3.3. Tractability via k-ary dependencies

Murphy et al. (2019) propose k-ary pooling whereby the computational complexity of summing over all permutations of an input sequence is reduced by considering only permutations of subsequences of size k. Inspired by this, we propose k-ary Relational Pooling which operates on k-node induced subgraphs of G, which corresponds to patches of size $k\times k\times(d_e+1)$ of $\mathbf{A}$ and k rows of $\boldsymbol{X}^{(v)}$. Formally, we define k-ary RP in joint RP by

$$\overline{\vec{f}}^{(k)}(G;\boldsymbol{W}) = \frac{1}{|V|!}\sum_{\pi\in\Pi_{|V|}}\vec{f}\Big(\mathbf{A}_{\pi,\pi}[1{:}k, 1{:}k, :], \boldsymbol{X}_{\pi}^{(v)}[1{:}k, :]; \boldsymbol{W}\Big), \tag{10}$$

where $\mathbf{A}[\cdot,\cdot,\cdot]$ denotes access to elements in the first, second, and third modes of $\mathbf{A}$; $a:b$ denotes selecting elements corresponding to indices from a to b inclusive; and ":" by itself denotes all elements along a mode. Thus, we permute the adjacency tensor and select fibers along the third mode from the upper left $k\times k\times(d_e+1)$ subtensor of $\mathbf{A}$ as well as the vertex attributes from the first k rows of $\boldsymbol{X}_{\pi}^{(v)}$. An illustration is shown in Figure 2. The graph on the right is numbered by its 'original' node indices and we assume that it has no vertex features and one-dimensional edge features. This 'original' graph would be represented by a $5\times5\times2$ tensor $\mathbf{A}$ where, for all pairs of vertices, the front slice holds adjacency matrix information and the back slice holds edge feature information (not shown). Given the permutation function $\pi^\dagger \in \Pi_{|V|}$ defined as $\pi^\dagger(1,2,3,4,5) = (3,4,1,2,5)$, the permuted $\mathbf{A}_{\pi^\dagger,\pi^\dagger}$ is shown on the left. Its entries show elements from $\mathbf{A}$ shuffled appropriately by $\pi^\dagger$. For $k=3$ RP, we select the upper-left 3×3 region from $\mathbf{A}_{\pi^\dagger,\pi^\dagger}$, shaded in red, and pass this to $\vec{f}$. This is repeated for all permutations of the vertices. For separate RP, the formulation is similar but we can select k_1 and k_2 nodes from $V^{(\text{r})}$ and $V^{(\text{c})}$, respectively.

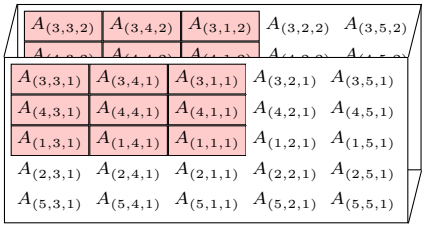
In practice, the relevant k-node induced subgraphs can be selected without first permuting the entire tensor $\mathbf{A}$ and matrix $\boldsymbol{X}^{(v)}$. Instead, we enumerate all subsets of size k from index set V and use those to index $\mathbf{A}$ and $\boldsymbol{X}^{(v)}$.

More generally, we have the following conclusion:

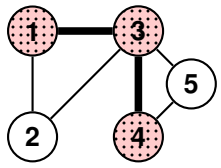
Proposition 2.2. *The RP in Equation 10 requires summing over all k-node induced subgraphs of G, thus saving computation when $k < |V|$, reducing the number of terms in the sum from $|V|!$ to $\frac{|V|!}{(|V|-k)!}$.*

Fewer computations are needed if $\vec{f}$ is made permutation-invariant over its input k-node induced subgraph. We now show that the expressiveness of k-ary RP increases with k.

Proposition 2.3. *$\overline{\vec{f}}^{(k)}$ becomes strictly more expressive as k increases. That is, for any $k\in\mathbb{N}$, define $\mathcal{F}_k$ as the set of all permutation-invariant graph functions that can be represented by RP with k-ary dependencies. Then, $\mathcal{F}_{k-1}\subset\mathcal{F}_k$.*



Adjacency tensor $\mathbf{A}_{\pi^\dagger,\pi^\dagger}$ where $\pi^\dagger(1,2,3,4,5) = (3,4,1,2,5)$ (elementwise): the top-left $3 \times 3 \times 2$ subtensor is passed to $\vec{f}^{(3)}$.



An example five-node graph encoded by $\mathbf{A}$. We select a 3-node induced subgraph, corresponding to the top-left of $\mathbf{A}_{\pi^\dagger,\pi^\dagger}$ indicated by shaded nodes and thickened edges.

Figure 2: Illustration of a k-ary ($k = 3$) RP on a 5-node graph with one-dimensional edge attributes ($d_e = 1$) and no vertex attributes. The graph is encoded as a $5 \times 5 \times 2$ tensor $\mathbf{A}$. k-ary RP selects the top-left $k \times k$ corner of a permuted tensor $\mathbf{A}_{\pi,\pi}$.

Further computational savings. The number of k-node induced subgraphs can be very large for even moderate-sized graphs. The following yield additional savings.

Ignoring some subgraphs: We can encode task- and model-specific knowledge by ignoring certain k-sized induced subgraphs, which amounts to fixing $\vec{f}$ to 0 for these graphs. For example, in most applications the graph structure – and not the node features alone – is important so we may ignore subgraphs of k isolated vertices. Such decisions can yield substantial computational savings in sparse graphs.

Use of π-SGD: We can combine the k-ary approximation with other strategies like π-SGD and poly-canonical orientations. For instance, a forward pass can consist of sampling a random starting vertex and running a BFS until a k-node induced subgraph is selected. Combining π-SGD and k-ary RP can speed up GNNs but will not provide unbiased estimates of the loss calculated with the entire graph. Future work could explore using the MCMC finite-sample unbiased estimator of Teixeira et al. (2018) with RP.

3. Related Work

Our Relational Pooling framework leverages insights from Janossy Pooling (Murphy et al., 2019), which learns expressive permutation-invariant functions over *sequences* by approximating an average over permutation-sensitive functions with tractability strategies. The present work raises novel applications – like RP-GNN – that arise when pooling over permutation-sensitive functions of *graphs*.

Graph Neural Networks (GNNs) and Graph Convolutional Networks (GCNs) form an increasingly popular class of methods (Scarselli et al., 2009; Bruna et al., 2014; Duvenaud et al., 2015; Niepert et al., 2016; Atwood & Towsley, 2016; Kipf & Welling, 2017; Gilmer et al., 2017; Monti et al., 2017; Defferrard et al., 2016; Hamilton et al., 2017a; Velickovic et al., 2018; Lee et al., 2018; Xu et al., 2019). Applications include chemistry, where molecules are represented as graphs and we seek to predict chemical properties like toxicity (Duvenaud et al., 2015; Gilmer et al., 2017; Lee et al., 2018; Wu et al., 2018; Sanchez-Lengeling & Aspuru-Guzik, 2018) and document classification on a citations network (Hamilton et al., 2017b); and many others (cf. Battaglia et al. (2018)).

Recently, Xu et al. (2019) and Morris et al. (2019) show that such GNNs are at most as powerful as the standard Weisfeiler-Lehman algorithm (also known as *color refinement* or *naive vertex classification* (Weisfeiler & Lehman, 1968; Arvind et al., 2017; Fürer, 2017)) for graph isomorphism testing, and can fail to distinguish between certain classes of graphs (Cai et al., 1992; Arvind et al., 2017; Fürer, 2017). In Section 4, we demonstrate this phenomenon and provide empirical evidence that RP can correct some of these shortcomings. Higher-order (k-th order) versions of the WL test (WL[k]) exist and operate on tuples of size k from V rather than on one vertex at a time (Fürer, 2017). Increasing k increases the capacity of WL[k] to distinguish nonisomorphic graphs, which can be exploited to build more powerful GNNs (Morris et al., 2019). Meng et al. (2018), introduce a WL[k]-type representation to predict high-order dynamics in temporal graphs. Using GNNs based on WL[k] may be able to give better $\vec{f}$ functions for RP but we focused on providing a representation for more expressive than WL[1] procedures.

In another direction, WL is used to construct graph kernels (Shervashidze et al., 2009; 2011). CNNs have also been used with graph kernels (Nikolentzos et al., 2018) and some GCNs can be seen as CNNs applied to single canonical orderings (Niepert et al., 2016; Defferrard et al., 2016). RP provides a framework for stochastic optimization over all or poly-canonical orderings. Another line of work derives bases for permutation-invariant functions of graphs and propose learning the coefficients of basis elements from data (Maron et al., 2018; Hartford et al., 2018).

In parallel, Bloem-Reddy & Teh (2019) generalized permutation-invariant functions to group-action invariant functions and discuss connections to exchangeable prob-

ability distributions (De Finetti, 1937; Diaconis & Janson, 2008; Aldous, 1981).Their theory uses a checkerboard function (Orbanz & Roy, 2015) and the left-order canonical orientation of Ghahramani & Griffiths (2006) to orient graphs but it will fail in some cases unless graph isomorphism can be solved in polynomial time. Also, as discussed, there is no guarantee that a hand-picked canonical orientation will perform well on all tasks. On the tractability side, Niepert & Van den Broeck (2014) shows that exchangeabilty assumptions in probabilistic graphical models provide a form of k-ary tractability and Cohen & Welling (2016); Ravanbakhsh et al. (2017) use symmetries to reduce sample complexity and save on computation. Another development explores the universality properties of invariance-preserving neural networks and concludes some architectures are computationally intractable (Maron et al., 2019). Closer to RP, Montavon et al. (2012) discusses random permutations but RP provides a more comprehensive framework with theoretical analysis.

4. Experiments

Our first experiment shows that RP-GNN is more expressive than WL-GNN. The second evaluates RP and its approximations on molecular data. Our code is on GitHub[2].

4.1. Testing RP-GNN vs WL-GNN

Here we perform experiments over the CSL graphs from Figure 1. We demonstrate empirically that WL-GNNs are limited in their power to represent them and that RP can be used to overcome this limitation. Our experiments compare the RP-GNN of Equation 5 using the Graph Isomorphism Network (GIN) architecture (Xu et al., 2019) as $\vec{f}$ against the original GIN architecture. We choose GIN as it is arguably the most powerful WL-GNN architecture.

For the CSL graphs, the "skip length" R effectively defines an isomorphism class in the sense that predicting R is tantamount to classifying a graph into its isomorphism class for a fixed number of vertices M. We are interested in predicting R as an assessment of RP's ability to exploit graph structure. We do not claim to tackle the graph isomorphism problem as we use approximate learning (π-SGD for RP).

RP-GIN. GIN follows the recursion of Equation 4, replacing JP with summation and defining $\phi^{(l)}$ as a function that sums its arguments and feeds them through an MLP:

$$\mathbf{h}_u^{(l)} = \text{MLP}^{(l)}\Big((1+\epsilon^{(l)})\mathbf{h}_u^{(l-1)} + \sum_{v\in\mathcal{N}(u)} \mathbf{h}_v^{(l-1)}\Big),$$

for $l = 1, \ldots, L$, where $\{\epsilon^{(l)}\}_{l=1}^{L}$ can be treated as hyperparameters or learned parameters (we train ϵ). This recursion yields vertex-level representations that can be mapped to a graph-level representation by summing across $\mathbf{h}_u^{(l)}$ at each given l, then concatenating the results, as proposed by Xu et al. (2019). When applying GIN directly on our CSL graphs, we assign a constant vertex attribute to all vertices in keeping with the traditional WL algorithm, as the graph is unattributed. Recall that RP-GIN assigns one-hot node IDs and passes the augmented graph to GIN ($\vec{f}$) (Equation 5). We cannot assign IDs with standard GIN as doing so renders it permutation-sensitive. Further implementation and training details are in the Supplementary Material.

[2] https://github.com/PurdueMINDS/RelationalPooling

Table 1: RP-GNN outperforms WL-GNN in 10-class classification task. Summary of validation-set accuracy (%).

model	mean	median	max	min	sd
RP-GIN	37.6	43.3	53.3	10.0	12.9
GIN	10.0	10.0	10.0	10.0	0.0

Classifying skip lengths. We create a dataset of graphs from $\{\mathcal{G}_{\text{skip}}(41, R)\}_R$ where $R \in \{2, 3, 4, 5, 6, 9, 11, 12, 13, 16\}$ and predict R as a discrete response. Note $M = 41$ is the smallest such that 10 nonisomorphic $\mathcal{G}_{\text{skip}}(M, R)$ can be formed; $\exists R_1 \neq R_2$ such that $\mathcal{G}_{\text{skip}}(M, R_1)$ and $\mathcal{G}_{\text{skip}}(M, R_2)$ are isomorphic. For all 10 classes, we form 15 adjacency matrices by first constructing $A^{(R)}$ according to Definition 2.1 and then 14 more as $A^{(R)}_{\pi,\pi}$ for 14 distinct permutations π. This gives a dataset of 150 graphs. We evaluate GIN and RP-GIN with five-fold cross validation – with balanced classes on both training and validation – on this task.

The validation-set accuracies for both models are shown in Table 1 and Figure 3 in the Supplementary Material. Since GIN learns the same representation for all graphs, it predicts the same class for all graphs in the validation fold, and therefore achieves random-guessing performance of 10% accuracy. In comparison, RP-GIN yields substantially stronger performance on all folds, demonstrating that RP-GNNs are more powerful than their WL-GNN and serving as empirical validation of Theorem 2.2.

4.2. Predicting Molecular Properties

Deep learning for chemical applications learns functions on graph representations of molecules and has a rich literature (Duvenaud et al., 2015; Kearnes et al., 2016; Gilmer et al., 2017). This domain provides challenging tasks on which to evaluate RP, while in other applications, different GNN models of varying sophistication often achieve similar performance (Shchur et al., 2018; Murphy et al., 2019; Xu et al., 2019). We chose datasets from the MoleculeNet project (Wu et al., 2018) – which collects chemical datasets and collates the performance of various models – that yield classification tasks and on which graph-

based methods achieved superior performance[3]. In particular, we chose MUV (Rohrer & Baumann, 2009), HIV, and Tox21 (Mayr et al., 2016; Huang et al., 2016), which contain measurements on a molecule's biological activity, ability to inhibit HIV, and qualitative toxicity, respectively.

We processed datasets with DeepChem (Ramsundar et al., 2019) and evaluated models with ROC-AUC per the MoleculeNet project. Molecules are encoded as graphs with 75- and 14-dimensional node and edge features. Table 3 (in Supplementary) provides more detail.

We use the best-performing graph model reported by MoleculeNet as $\vec{f}$ to evaluate k-ary RP and to explore whether RP-GNN can make it more powerful. This is a model inspired by the GNN in Duvenaud et al. (2015), implemented in DeepChem by Altae-Tran et al. (2017), which we refer to as the 'Duvenaud et al.' model. This model is specialized for molecules; it trains a distinct weight matrix for each possible vertex degree at each layer, which would be infeasible in other domains. One might ask whether RP-GNN can add any power to this state-of-the-art model, which we will explore here. We evaluated GIN (Xu et al., 2019) but it was unable to outperform 'Duvenaud et al'. Model architectures, hyperparameters, and training procedures are detailed in the Supplementary Material.

RP-GNN We compare the performance of the 'Duvenaud et al.' baseline to RP-Duvenaud, wherein the 'Duvenaud et al.' GNN is used as $\vec{f}$ in Equation 5. We evaluate $\vec{f}$ on the entire graph but make RP-Duvenaud tractable by training with π-SGD. At inference time, we sample 20 permutations (see Remark 2.1). Additionally, we assign just enough one-hot IDs to make atoms of the same type have unique IDs (as discussed in Section 2.2). To quantify variability, we train over 20 random data splits.

The results shown in Table 2 suggest that RP-Duvenaud is more powerful than the baseline on the HIV task and similar in performance on the others. While we bear in mind the over-confidence in the variability estimates (Bengio & Grandvalet, 2004), this provides support of our theory.

k-ary RP experiments Next we empirically assess the tradeoffs involved in the k-ary dependency models – evaluating $\vec{f}$ on k-node induced subgraphs – discussed in Section 2.3.3. Propositions 2.3 and 2.2 show that expressive power and computation decrease with k. Here, $\vec{f}$ is a 'Duvenaud et al. model' that operates on induced subgraphs of size $k = 10, 20, 30, 40, 50$ (the percentages of molecules with more than k atoms in each dataset are shown in the Supplementary Material). We train using π-SGD (20 inference-time samples) and evaluate using five random train/val/test splits.

[3] moleculenet.ai/latest-results (Dec. 2018)

Table 2: Evaluation of RP-GNN and k-ary RP where $\vec{f}$ is the 'Duvenaud et al.' GNN or a neural-network. We show mean (standard deviation) ROC-AUC across multiple random train/val/test splits. DFS indicates Depth-First Search poly-canonical orientation.

model	**HIV**	**MUV**	**Tox21**
RP-Duvenaud et al.	0.832 (0.013)	0.794 (0.025)	0.799 (0.006)
Duvenaud et al.	0.812 (0.014)	0.798 (0.025)	0.794 (0.010)
$k=50$ Duvenaud et al.	0.818 (0.022)	0.768 (0.014)	0.778 (0.007)
$k=40$ Duvenaud et al.	0.807 (0.025)	0.776 (0.032)	0.783 (0.007)
$k=30$ Duvenaud et al.	0.829 (0.024)	0.776 (0.030)	0.775 (0.011)
$k=20$ Duvenaud et al.	0.813 (0.017)	0.777 (0.041)	0.755 (0.003)
$k=10$ Duvenaud et al.	0.812 (0.035)	0.773 (0.045)	0.687 (0.005)
CNN-DFS	0.542 (0.004)	0.601 (0.042)	0.597 (0.006)
RNN-DFS	0.627 (0.007)	0.648 (0.014)	0.748 (0.055)

Results are shown in Table 2 and Figures 4, 5, and 6 in the Supplementary Material. With the Tox21 dataset, we see a steady increase in predictive performance and computation as k increases. For instance, k-ary with $k = 10$ is 25% faster than the baseline with mean AUC 0.687 (0.005 sd) and with $k = 20$ being 10% faster with AUC 0.755 (0.003 sd), where (sd) indicates the standard deviation over 5 bootstrapped runs. Results level off around $k = 30$. For the other datasets, neither predictive performance nor computation vary significantly with k. Overall, the molecules are quite small and we do not expect dramatic speed-ups with smaller k, but this enables comparing between using the entire graph and its k-sized induced subgraphs.

RP with CNNs and RNNs. RP permits using neural networks for $\vec{f}$. We explored RNNs and CNNs and report the results in Table 2. Specific details are discussed in the Supplementary Material. The RNN achieves reasonable performance on Tox21 and underperforms on the other tasks. The CNN underperforms on all tasks. Future work is needed to determine tasks where these approaches are better suited.

5. Conclusions

In this work, we proposed the Relational Pooling (RP) framework for graph classification and regression. RP gives ideal most-powerful, though intractable, graph representations. We proposed several approaches to tractably approximate this ideal and showed theoretically and empirically that RP can make WL-GNNs more expressive than the WL test. RP permits neural networks like RNNs and CNNs to be brought to such problems. Our experiments evaluate RP on a number of datasets and show how our framework can be used to improve properties of state-of-the-art methods. Future directions for theoretical study include improving our understanding of the tradeoff between representation power and computational cost of our tractability strategies.

Acknowledgments

This work was sponsored in part by the ARO, under the U.S. Army Research Laboratory contract number W911NF-09-2-0053, the Purdue Integrative Data Science Initiative and the Purdue Research foundation, the DOD through SERC under contract number HQ0034-13-D-0004 RT #206, and the National Science Foundation under contract numbers IIS-1816499 and DMS-1812197.

References

Aldous, D. J. Representations for partially exchangeable arrays of random variables. *J. Multivar. Anal.*, 11 (4):581–598, 1981. ISSN 0047259X. doi: 10.1016/0047-259X(81)90099-3.

Altae-Tran, H., Ramsundar, B., Pappu, A. S., and Pande, V. Low data drug discovery with one-shot learning. *ACS central science*, 3(4):283–293, 2017.

Arvind, V., Köbler, J., Rattan, G., and Verbitsky, O. Graph isomorphism, color refinement, and compactness. *computational complexity*, 26(3):627–685, 2017.

Atwood, J. and Towsley, D. Diffusion-convolutional neural networks. In *Advances in Neural Information Processing Systems*, pp. 1993–2001, 2016.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Bengio, Y. and Grandvalet, Y. No unbiased estimator of the variance of k-fold cross-validation. *Journal of machine learning research*, 5(Sep):1089–1105, 2004.

Bloem-Reddy, B. and Teh, Y. W. Probabilistic symmetry and invariant neural networks. *arXiv preprint arXiv:1901.06082*, 2019.

Bottou, L. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pp. 421–436. Springer, 2012.

Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., and Vandergheynst, P. Geometric Deep Learning: Going beyond Euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, jul 2017. ISSN 1053-5888. doi: 10.1109/MSP.2017.2693418. URL http://ieeexplore.ieee.org/document/7974879/.

Bruna, J., Zaremba, W., Szlam, A., and LeCun, Y. Spectral networks and locally connected networks on graphs. In *International Conference on Learning Representations*, 2014.

Cai, J.-Y., Fürer, M., and Immerman, N. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, 1992.

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1179. URL https://www.aclweb.org/anthology/D14-1179.

Cohen, T. and Welling, M. Group equivariant convolutional networks. In *International conference on machine learning*, pp. 2990–2999, 2016.

De Finetti, B. La prévision: ses lois logiques, ses sources subjectives. In *Annales de l'institut Henri Poincaré*, volume 7, pp. 1–68, 1937. [Translated into Enlish: H. E. Kyburg and H.E. Smokler, eds. Studies in Subjective Probability. *Krieger* 53-118, 1980].

Defferrard, M., Bresson, X., and Vandergheynst, P. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pp. 3844–3852, 2016.

Diaconis, P. and Janson, S. Graph limits and exchangeable random graphs. *Rend. di Mat. e delle sue Appl. Ser. VII*, 28:33–61, 2008. ISSN 1542-7951. doi: 10.1080/15427951.2008.10129166. URL http://arxiv.org/abs/0712.2749.

Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

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. 2224–2232, 2015.

Fürer, M. On the combinatorial power of the Weisfeiler-Lehman algorithm. In *International Conference on Algorithms and Complexity*, pp. 260–271. Springer, 2017.

Ghahramani, Z. and Griffiths, T. L. Infinite latent feature models and the Indian buffet process. In *Advances in neural information processing systems*, pp. 475–482, 2006.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In Precup, D. and Teh, Y. W. (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1263–1272, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL http://proceedings.mlr.press/v70/gilmer17a.html.

Hamilton, W., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pp. 1024–1034, 2017a.

Hamilton, W. L., Ying, R., and Leskovec, J. Representation learning on graphs: Methods and applications. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 40(3):52–74, 2017b.

Hartford, J., Graham, D. R., Leyton-Brown, K., and Ravanbakhsh, S. Deep models of interactions across sets. *arXiv preprint arXiv:1803.02879*, 2018.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Hornik, K., Stinchcombe, M., and White, H. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Huang, R., Xia, M., Nguyen, D.-T., Zhao, T., Sakamuru, S., Zhao, J., Shahane, S. A., Rossoshek, A., and Simeonov, A. Tox21challenge to build predictive models of nuclear receptor and stress response pathways as mediated by exposure to environmental chemicals and drugs. *Frontiers in Environmental Science*, 3:85, 2016.

Ji, S., Xu, W., Yang, M., and Yu, K. 3D convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 35 (1):221–231, 2013.

Kearnes, S., McCloskey, K., Berndl, M., Pande, V., and Riley, P. Molecular graph convolutions: moving beyond fingerprints. *Journal of computer-aided molecular design*, 30(8):595–608, 2016.

Kingma, D. P. and Ba, J. L. ADAM: A Method for Stochastic Optimization. *International Conference on Learning Representations, ICLR*, 2015.

Kipf, T. and Welling, M. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

Lee, J. B., Rossi, R., and Kong, X. Graph classification using structural attention. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1666–1674. ACM, 2018.

Maron, H., Ben-Hamu, H., Shamir, N., and Lipman, Y. Invariant and equivariant graph networks. *arXiv preprint arXiv:1812.09902*, 2018.

Maron, H., Fetaya, E., Segol, N., and Lipman, Y. On the universality of invariant networks. *arXiv preprint arXiv:1901.09342*, 2019.

Mayr, A., Klambauer, G., Unterthiner, T., and Hochreiter, S. Deeptox: toxicity prediction using deep learning. *Frontiers in Environmental Science*, 3:80, 2016.

Meng, C., Mouli, S. C., Ribeiro, B., and Neville, J. Subgraph pattern neural networks for high-order graph evolution prediction. In *AAAI*, 2018.

Montavon, G., Hansen, K., Fazli, S., Rupp, M., Biegler, F., Ziehe, A., Tkatchenko, A., Lilienfeld, A. V., and Müller, K.-R. Learning invariant representations of molecules for atomization energy prediction. In *Advances in Neural Information Processing Systems*, pp. 440–448, 2012.

Monti, F., Boscaini, D., Masci, J., Rodola, E., Svoboda, J., and Bronstein, M. M. Geometric deep learning on graphs and manifolds using mixture model cnns. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5115–5124, 2017.

Morris, C., Ritzert, M., Fey, M., Hamilton, W. L., Lenssen, J. E., Rattan, G., and Grohe, M. Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks. *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, 2019.

Murphy, R. L., Srinivasan, B., Rao, V., and Ribeiro, B. Janossy pooling: Learning deep permutation-invariant functions for variable-size inputs. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=BJluy2RcFm.

Niepert, M. and Van den Broeck, G. Tractability through exchangeability: A new perspective on efficient probabilistic inference. In *AAAI*, pp. 2467–2475, 2014.

Niepert, M., Ahmed, M., and Kutzkov, K. Learning convolutional neural networks for graphs. In *International conference on machine learning*, pp. 2014–2023, 2016.

Nikolentzos, G., Meladianos, P., Tixier, A. J.-P., Skianis, K., and Vazirgiannis, M. Kernel graph convolutional neural networks. In *International Conference on Artificial Neural Networks*, pp. 22–32. Springer, 2018.

Orbanz, P. and Roy, D. M. Bayesian models of graphs, arrays and other exchangeable random structures. *IEEE transactions on pattern analysis and machine intelligence*, 37(2):437–461, 2015.

Qi, C. R., Su, H., Nießner, M., Dai, A., Yan, M., and Guibas, L. J. Volumetric and multi-view CNNs for object classification on 3D data. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 5648–5656, 2016.

Ramsundar, B., Eastman, P., Leswing, K., Walters, P., and Pande, V. *Deep Learning for the Life Sciences*. O'Reilly Media, 2019. https://www.amazon.com/Deep-Learning-Life-Sciences-Microscopy/dp/1492039837.

Ravanbakhsh, S., Schneider, J., and Poczos, B. Equivariance through parameter-sharing. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 2892–2901. JMLR. org, 2017.

Robbins, H. and Monro, S. A stochastic approximation method. *The annals of mathematical statistics*, pp. 400–407, 1951.

Rohrer, S. G. and Baumann, K. Maximum unbiased validation (muv) data sets for virtual screening based on pubchem bioactivity data. *Journal of chemical information and modeling*, 49(2):169–184, 2009.

Sanchez-Lengeling, B. and Aspuru-Guzik, A. Inverse molecular design using machine learning: Generative models for matter engineering. *Science*, 361(6400):360–365, 2018.

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

Shchur, O., Mumme, M., Bojchevski, A., and Günnemann, S. Pitfalls of graph neural network evaluation. *Relational Representation Learning Workshop (R2L 2018), NeurIPS*, 2018.

Shervashidze, N., Vishwanathan, S., Petri, T., Mehlhorn, K., and Borgwardt, K. Efficient graphlet kernels for large graph comparison. In *Artificial Intelligence and Statistics*, pp. 488–495, 2009.

Shervashidze, N., Schweitzer, P., Leeuwen, E. J. v., Mehlhorn, K., and Borgwardt, K. M. Wisfeiler-Lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep):2539–2561, 2011.

Teixeira, C. H., Cotta, L., Ribeiro, B., and Meira, W. Graph pattern mining and learning through user-defined relations. In *2018 IEEE International Conference on Data Mining (ICDM)*, pp. 1266–1271. IEEE, 2018.

Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. Graph attention networks. *ICLR*, 2018.

Vilfred, V. On circulant graphs. In Balakrishnan, R., Sethuraman, G., and Wilson, R. J. (eds.), *Graph Theory and its Applications*, pp. 34–36. Narosa Publishing House, 2004.

Weisfeiler, B. and Lehman, A. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9):12–16, 1968.

Wu, Z., Ramsundar, B., Feinberg, E. N., Gomes, J., Geniesse, C., Pappu, A. S., Leswing, K., and Pande, V. Moleculenet: a benchmark for molecular machine learning. *Chemical science*, 9(2):513–530, 2018.

Xu, K., Li, C., Tian, Y., Sonobe, T., Kawarabayashi, K.-i., and Jegelka, S. Representation Learning on Graphs with Jumping Knowledge Networks. In *ICML*, 2018. URL http://arxiv.org/abs/1806.03536.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=ryGs6iA5Km.

Ying, Z., You, J., Morris, C., Ren, X., Hamilton, W., and Leskovec, J. Hierarchical graph representation learning with differentiable pooling. In *Advances in Neural Information Processing Systems*, pp. 4800–4810, 2018.

Younes, L. On the convergence of markovian stochastic algorithms with rapidly decreasing ergodicity rates. *Stochastics: An International Journal of Probability and Stochastic Processes*, 65(3-4):177–228, 1999.

Yuille, A. L. The convergence of contrastive divergences. In *Advances in neural information processing systems*, pp. 1593–1600, 2005.

Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R. R., and Smola, A. J. Deep sets. In *Advances in neural information processing systems*, pp. 3391–3401, 2017.

Diffusion Improves Graph Learning

Johannes Gasteiger, Stefan Weißenberger, Stephan Günnemann
Technical University of Munich
{j.gasteiger,stefan.weissenberger,guennemann}@in.tum.de

Abstract

Graph convolution is the core of most Graph Neural Networks (GNNs) and usually approximated by message passing between direct (one-hop) neighbors. In this work, we remove the restriction of using only the direct neighbors by introducing a powerful, yet spatially localized graph convolution: Graph diffusion convolution (GDC). GDC leverages generalized graph diffusion, examples of which are the heat kernel and personalized PageRank. It alleviates the problem of noisy and often arbitrarily defined edges in real graphs. We show that GDC is closely related to spectral-based models and thus combines the strengths of both spatial (message passing) and spectral methods. We demonstrate that replacing message passing with graph diffusion convolution consistently leads to significant performance improvements across a wide range of models on both supervised and unsupervised tasks and a variety of datasets. Furthermore, GDC is not limited to GNNs but can trivially be combined with any graph-based model or algorithm (e.g. spectral clustering) without requiring any changes to the latter or affecting its computational complexity. Our implementation is available online. [1]

1 Introduction

When people started using graphs for evaluating chess tournaments in the middle of the 19th century they only considered each player's direct opponents, i.e. their first-hop neighbors. Only later was the analysis extended to recursively consider higher-order relationships via $\boldsymbol{A}^2$, $\boldsymbol{A}^3$, etc. and finally generalized to consider all exponents at once, using the adjacency matrix's dominant eigenvector [38, 75]. The field of Graph Neural Networks (GNNs) is currently in a similar state. Graph Convolutional Networks (GCNs) [33], also referred to as Message Passing Neural Networks (MPNNs) [24] are the prevalent approach in this field but they only pass messages between neighboring nodes in each layer. These messages are then aggregated at each node to form the embedding for the next layer. While MPNNs do leverage higher-order neighborhoods in deeper layers, limiting each layer's messages to one-hop neighbors seems arbitrary. Edges in real graphs are often noisy or defined using an arbitrary threshold [70], so we can clearly improve upon this approach.

Since MPNNs only use the immediate neigborhod information, they are often referred to as spatial methods. On the other hand, spectral-based models do not just rely on first-hop neighbors and capture more complex graph properties [16]. However, while being theoretically more elegant, these methods are routinely outperformed by MPNNs on graph-related tasks [33, 74, 81] and do not generalize to previously unseen graphs. This shows that message passing is a powerful framework worth extending upon. To reconcile these two separate approaches and combine their strengths we propose a novel technique of performing message passing inspired by spectral methods: Graph diffusion convolution (GDC). Instead of aggregating information only from the first-hop neighbors, GDC aggregates information from a larger neighborhood. This neighborhood is constructed via a new graph generated by sparsifying a generalized form of graph diffusion. We show how graph diffusion

[1] https://www.daml.in.tum.de/gdc

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

is expressed as an equivalent polynomial filter and how GDC is closely related to spectral-based models while addressing their shortcomings. GDC is spatially localized, scalable, can be combined with message passing, and generalizes to unseen graphs. Furthermore, since GDC generates a new sparse graph it is not limited to MPNNs and can trivially be combined with *any* existing graph-based model or algorithm in a plug-and-play manner, i.e. without requiring changing the model or affecting its computational complexity. We show that GDC consistently improves performance across a wide range of models on both supervised and unsupervised tasks and various homophilic datasets. In summary, this paper's core contributions are:

1. Proposing graph diffusion convolution (GDC), a more powerful and general, yet spatially localized alternative to message passing that uses a sparsified generalized form of graph diffusion. GDC is not limited to GNNs and can be combined with any graph-based model or algorithm.
2. Analyzing the spectral properties of GDC and graph diffusion. We show how graph diffusion is expressed as an equivalent polynomial filter and analyze GDC's effect on the graph spectrum.
3. Comparing and evaluating several specific variants of GDC and demonstrating its wide applicability to supervised and unsupervised learning on graphs.

2 Generalized graph diffusion

We consider an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with node set $\mathcal{V}$ and edge set $\mathcal{E}$. We denote with $N = |\mathcal{V}|$ the number of nodes and $\boldsymbol{A} \in \mathbb{R}^{N \times N}$ the adjacency matrix. We define generalized graph diffusion via the diffusion matrix

$$\boldsymbol{S} = \sum_{k=0}^{\infty} \theta_k \boldsymbol{T}^k, \tag{1}$$

with the weighting coefficients θ_k, and the generalized transition matrix $\boldsymbol{T}$. The choice of θ_k and $\boldsymbol{T}^k$ must at least ensure that Eq. 1 converges. In this work we will consider somewhat stricter conditions and require that $\sum_{k=0}^{\infty} \theta_k = 1$, $\theta_k \in [0, 1]$, and that the eigenvalues of $\boldsymbol{T}$ are bounded by $\lambda_i \in [0, 1]$, which together are sufficient to guarantee convergence. Note that regular graph diffusion commonly requires $\boldsymbol{T}$ to be column- or row-stochastic.

Transition matrix. Examples for $\boldsymbol{T}$ in an undirected graph include the random walk transition matrix $\boldsymbol{T}_{\text{rw}} = \boldsymbol{A}\boldsymbol{D}^{-1}$ and the symmetric transition matrix $\boldsymbol{T}_{\text{sym}} = \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$, where the degree matrix $\boldsymbol{D}$ is the diagonal matrix of node degrees, i.e. $D_{ii} = \sum_{j=1}^{N} A_{ij}$. Note that in our definition $\boldsymbol{T}_{\text{rw}}$ is column-stochastic. We furthermore adjust the random walk by adding (weighted) self-loops to the original adjacency matrix, i.e. use $\tilde{\boldsymbol{T}}_{\text{sym}} = (w_{\text{loop}}\boldsymbol{I}_N + \boldsymbol{D})^{-1/2}(w_{\text{loop}}\boldsymbol{I}_N + \boldsymbol{A})(w_{\text{loop}}\boldsymbol{I}_N + \boldsymbol{D})^{-1/2}$, with the self-loop weight $w_{\text{loop}} \in \mathbb{R}^+$. This is equivalent to performing a lazy random walk with a probability of staying at node i of $p_{\text{stay},i} = w_{\text{loop}}/D_i$.

Special cases. Two popular examples of graph diffusion are personalized PageRank (PPR) [57] and the heat kernel [36]. PPR corresponds to choosing $\boldsymbol{T} = \boldsymbol{T}_{\text{rw}}$ and $\theta_k^{\text{PPR}} = \alpha(1-\alpha)^k$, with teleport probability $\alpha \in (0, 1)$ [14]. The heat kernel uses $\boldsymbol{T} = \boldsymbol{T}_{\text{rw}}$ and $\theta_k^{\text{HK}} = e^{-t}\frac{t^k}{k!}$, with the diffusion time t [14]. Another special case of generalized graph diffusion is the approximated graph convolution introduced by Kipf & Welling [33], which translates to $\theta_1 = 1$ and $\theta_k = 0$ for $k \neq 1$ and uses $\boldsymbol{T} = \tilde{\boldsymbol{T}}_{\text{sym}}$ with $w_{\text{loop}} = 1$.

Weighting coefficients. We compute the series defined by Eq. 1 either in closed-form, if possible, or by restricting the sum to a finite number K. Both the coefficients defined by PPR and the heat kernel give a closed-form solution for this series that we found to perform well for the tasks considered. Note that we are not restricted to using $\boldsymbol{T}_{\text{rw}}$ and can use any generalized transition matrix along with the coefficients θ_k^{PPR} or θ_k^{HK} and the series still converges. We can furthermore choose θ_k by repurposing the graph-specific coefficients obtained by methods that optimize coefficients analogous to θ_k as part of their training process. We investigated this approach using label propagation [8, 13] and node embedding models [1]. However, we found that the simple coefficients defined by PPR or the heat kernel perform better than those learned by these models (see Fig. 7 in Sec. 6).

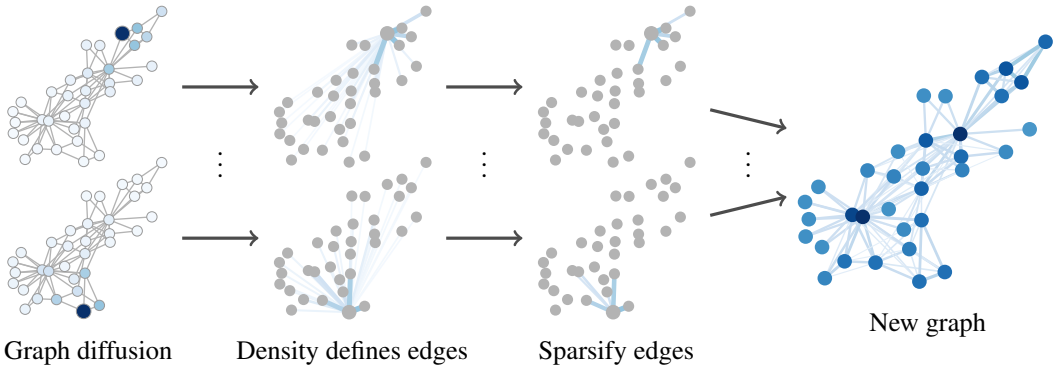


Figure 1: Illustration of graph diffusion convolution (GDC). We transform a graph $\boldsymbol{A}$ via graph diffusion and sparsification into a new graph $\tilde{\boldsymbol{S}}$ and run the given model on this graph instead.

3 Graph diffusion convolution

Essentially, graph diffusion convolution (GDC) exchanges the normal adjacency matrix $\boldsymbol{A}$ with a sparsified version $\tilde{\boldsymbol{S}}$ of the generalized graph diffusion matrix $\boldsymbol{S}$, as illustrated by Fig. 1. This matrix defines a weighted and directed graph, and the model we aim to augment is applied to this graph instead. We found that the calculated edge weights are beneficial for the tasks considered. However, we even found that GDC works when ignoring the weights after sparsification. This enables us to use GDC with models that only support unweighted edges such as the degree-corrected stochastic block model (DCSBM). If required, we make the graph undirected by using $(\tilde{\boldsymbol{S}} + \tilde{\boldsymbol{S}}^T)/2$, e.g. for spectral clustering. With these adjustments GDC is applicable to *any* graph-based model or algorithm.

Intuition. The general intuition behind GDC is that graph diffusion smooths out the neighborhood over the graph, acting as a kind of denoising filter similar to Gaussian filters on images. This helps with graph learning since both features and edges in real graphs are often noisy. Previous works also highlighted the effectiveness of graph denoising. Berberidis & Giannakis [7] showed that PPR is able to reconstruct the underlying probability matrix of a sampled stochastic block model (SBM) graph. Kloumann et al. [35] and Ragain [64] showed that PPR is optimal in recovering the SBM and DCSBM clusters in the space of landing probabilities under the mean field assumption. Li et al. [40] generalized this result by analyzing the convergence of landing probabilities to their mean field values. These results confirm the intuition that graph diffusion-based smoothing indeed recovers meaningful neighborhoods from noisy graphs.

Sparsification. Most graph diffusions result in a dense matrix $\boldsymbol{S}$. This happens even if we do not sum to $k = \infty$ in Eq. 1 due to the "four/six degrees of separation" in real-world graphs [5]. However, the values in $\boldsymbol{S}$ represent the influence between all pairs of nodes, which typically are highly localized [54]. This is a major advantage over spectral-based models since the spectral domain does not provide any notion of locality. Spatial localization allows us to simply truncate small values of $\boldsymbol{S}$ and recover sparsity, resulting in the matrix $\tilde{\boldsymbol{S}}$. In this work we consider two options for sparsification: 1. top-k: Use the k entries with the highest mass per column, 2. Threshold ϵ: Set entries below ϵ to zero. Sparsification would still require calculating a dense matrix $\boldsymbol{S}$ during preprocessing. However, many popular graph diffusions can be approximated efficiently and accurately in linear time and space. Most importantly, there are fast approximations for both PPR [3, 77] and the heat kernel [34], with which GDC achieves a linear runtime $\mathcal{O}(N)$. Furthermore, top-k truncation generates a regular graph, which is amenable to batching methods and solves problems related to widely varying node degrees [15]. Empirically, we even found that sparsification slightly *improves* prediction accuracy (see Fig. 5 in Sec. 6). After sparsification we calculate the (symmetric or random walk) transition matrix on the resulting graph via $\boldsymbol{T}^{\tilde{\boldsymbol{S}}}_{\text{sym}} = \boldsymbol{D}_{\tilde{\boldsymbol{S}}}^{-1/2} \tilde{\boldsymbol{S}} \boldsymbol{D}_{\tilde{\boldsymbol{S}}}^{-1/2}$.

Limitations. GDC is based on the assumption of homophily, i.e. "birds of a feather flock together" [49]. Many methods share this assumption and most common datasets adhere to this principle. However, this is an often overlooked limitation and it seems non-straightforward to overcome. One way of extending GDC to heterophily, i.e. "opposites attract", might be negative edge weights

[17, 44]. Furthermore, we suspect that GDC does not perform well in settings with more complex edges (e.g. knowledge graphs) or graph reconstruction tasks such as link prediction. Preliminary experiments showed that GDC indeed does not improve link prediction performance.

4 Spectral analysis of GDC

Even though GDC is a spatial-based method it can also be interpreted as a graph convolution and analyzed in the graph spectral domain. In this section we show how generalized graph diffusion is expressed as an equivalent polynomial filter and vice versa. Additionally, we perform a spectral analysis of GDC, which highlights the tight connection between GDC and spectral-based models.

Spectral graph theory. To employ the tools of spectral theory to graphs we exchange the regular Laplace operator with either the unnormalized Laplacian $\boldsymbol{L}_{\text{un}} = \boldsymbol{D} - \boldsymbol{A}$, the random-walk normalized $\boldsymbol{L}_{\text{rw}} = \boldsymbol{I}_N - \boldsymbol{T}_{\text{rw}}$, or the symmetric normalized graph Laplacian $\boldsymbol{L}_{\text{sym}} = \boldsymbol{I}_N - \boldsymbol{T}_{\text{sym}}$ [76]. The Laplacian's eigendecomposition is $\boldsymbol{L} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T$, where both $\boldsymbol{U}$ and $\boldsymbol{\Lambda}$ are real-valued. The graph Fourier transform of a vector $\boldsymbol{x}$ is then defined via $\hat{\boldsymbol{x}} = \boldsymbol{U}^T\boldsymbol{x}$ and its inverse as $\boldsymbol{x} = \boldsymbol{U}\hat{\boldsymbol{x}}$. Using this we define a graph convolution on $\mathcal{G}$ as $\boldsymbol{x} *_{\mathcal{G}} \boldsymbol{y} = \boldsymbol{U}((\boldsymbol{U}^T\boldsymbol{x}) \odot (\boldsymbol{U}^T\boldsymbol{y}))$, where $\odot$ denotes the Hadamard product. Hence, a filter g_ξ with parameters ξ acts on $\boldsymbol{x}$ as $g_\xi(\boldsymbol{L})\boldsymbol{x} = \boldsymbol{U}\hat{\boldsymbol{G}}_\xi(\boldsymbol{\Lambda})\boldsymbol{U}^T\boldsymbol{x}$, where $\hat{\boldsymbol{G}}_\xi(\boldsymbol{\Lambda}) = \text{diag}(\hat{g}_{\xi,1}(\boldsymbol{\Lambda}), \dots, \hat{g}_{\xi,N}(\boldsymbol{\Lambda}))$. A common choice for g_ξ in the literature is a polynomial filter of order J, since it is localized and has a limited number of parameters [16, 28]:

$$g_\xi(\boldsymbol{L}) = \sum_{j=0}^{J} \xi_j \boldsymbol{L}^j = \boldsymbol{U} \left(\sum_{j=0}^{J} \xi_j \boldsymbol{\Lambda}^j \right) \boldsymbol{U}^T. \tag{2}$$

Graph diffusion as a polynomial filter. Comparing Eq. 1 with Eq. 2 shows the close relationship between polynomial filters and generalized graph diffusion since we only need to exchange $\boldsymbol{L}$ by $\boldsymbol{T}$ to go from one to the other. To make this relationship more specific and find a direct correspondence between GDC with θ_k and a polynomial filter with parameters ξ_j we need to find parameters that solve

$$\sum_{j=0}^{J} \xi_j \boldsymbol{L}^j \stackrel{!}{=} \sum_{k=0}^{K} \theta_k \boldsymbol{T}^k. \tag{3}$$

To find these parameters we choose the Laplacian corresponding to $\boldsymbol{L} = \boldsymbol{I}_n - \boldsymbol{T}$, resulting in (see App. A)

$$\xi_j = \sum_{k=j}^{K} \binom{k}{j} (-1)^j \theta_k, \qquad \theta_k = \sum_{j=k}^{J} \binom{j}{k} (-1)^k \xi_j, \tag{4}$$

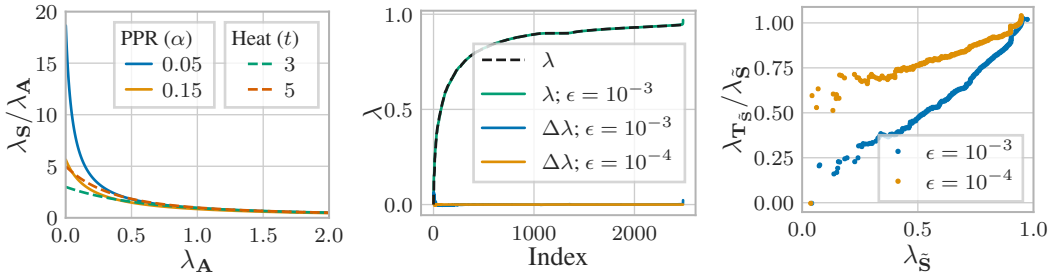
which shows the direct correspondence between graph diffusion and spectral methods. Note that we need to set $J = K$. Solving Eq. 4 for the coefficients corresponding to the heat kernel θ_k^{HK} and PPR θ_k^{PPR} leads to

$$\xi_j^{\text{HK}} = \frac{(-t)^j}{j!}, \qquad \xi_j^{\text{PPR}} = \left(1 - \frac{1}{\alpha}\right)^j, \tag{5}$$

showing how the heat kernel and PPR are expressed as polynomial filters. Note that PPR's corresponding polynomial filter converges only if $\alpha > 0.5$. This is caused by changing the order of summation when deriving ξ_j^{PPR}, which results in an alternating series. However, if the series does converge it gives the exact same transformation as the equivalent graph diffusion.

Spectral properties of GDC. We will now extend the discussion to all parts of GDC and analyze how they transform the graph Laplacian's eigenvalues. GDC consists of four steps: 1. Calculate the transition matrix $\boldsymbol{T}$, 2. take the sum in Eq. 1 to obtain $\boldsymbol{S}$, 3. sparsify the resulting matrix by truncating small values, resulting in $\tilde{\boldsymbol{S}}$, and 4. calculate the transition matrix $\boldsymbol{T}_{\tilde{\boldsymbol{S}}}$.

1. Transition matrix. Calculating the transition matrix $\boldsymbol{T}$ only changes which Laplacian matrix we use for analyzing the graph's spectrum, i.e. we use $\boldsymbol{L}_{\text{sym}}$ or $\boldsymbol{L}_{\text{rw}}$ instead of $\boldsymbol{L}_{\text{un}}$. Adding self-loops to obtain $\tilde{\boldsymbol{T}}$ does not preserve the eigenvectors and its effect therefore cannot be calculated precisely. Wu et al. [78] empirically found that adding self-loops shrinks the graph's eigenvalues.



(a) Graph diffusion as a filter, PPR with α and heat kernel with t. Both act as low-pass filters.

(b) Sparsification with threshold ϵ of PPR ($\alpha = 0.1$) on CORA. Eigenvalues are almost unchanged.

(c) Transition matrix on CORA's sparsified graph $\tilde{\boldsymbol{S}}$. This acts as a weak high-pass filter.

Figure 2: Influence of different parts of GDC on the Laplacian's eigenvalues λ.

2. Sum over $\boldsymbol{T}^k$. Summation does not affect the eigenvectors of the original matrix, since $\boldsymbol{T}^k\boldsymbol{v}_i = \lambda_i\boldsymbol{T}^{k-1}\boldsymbol{v}_i = \lambda_i^k\boldsymbol{v}_i$, for the eigenvector $\boldsymbol{v}_i$ of $\boldsymbol{T}$ with associated eigenvalue λ_i. This also shows that the eigenvalues are transformed as

$$\tilde{\lambda}_i = \sum_{k=0}^{\infty}\theta_k\lambda_i^k. \tag{6}$$

Since the eigenvalues of $\boldsymbol{T}$ are bounded by 1 we can use the geometric series to derive a closed-form expression for PPR, i.e. $\tilde{\lambda}_i = \alpha\sum_{k=0}^{\infty}(1-\alpha)^k\lambda_i^k = \frac{\alpha}{1-(1-\alpha)\lambda_i}$. For the heat kernel we use the exponential series, resulting in $\tilde{\lambda}_i = e^{-t}\sum_{k=0}^{\infty}\frac{t^k}{k!}\lambda_i^k = e^{t(\lambda_i-1)}$. How this transformation affects the corresponding Laplacian's eigenvalues is illustrated in Fig. 2a. Both PPR and the heat kernel act as low-pass filters. Low eigenvalues corresponding to large-scale structure in the graph (e.g. clusters [55]) are amplified, while high eigenvalues corresponding to fine details but also noise are suppressed.

3. Sparsification. Sparsification changes both the eigenvalues and the eigenvectors, which means that there is no direct correspondence between the eigenvalues of $\boldsymbol{S}$ and $\tilde{\boldsymbol{S}}$ and we cannot analyze its effect analytically. However, we can use eigenvalue perturbation theory (Stewart & Sun [69], Corollary 4.13) to derive the upper bound

$$\sqrt{\sum_{i=1}^{N}(\tilde{\lambda}_i - \lambda_i)^2} \leq ||\boldsymbol{E}||_{\mathrm{F}} \leq N||\boldsymbol{E}||_{\max} \leq N\epsilon, \tag{7}$$

with the perturbation matrix $\boldsymbol{E} = \tilde{\boldsymbol{S}} - \boldsymbol{S}$ and the threshold ϵ. This bound significantly overestimates the perturbation since PPR and the heat kernel both exhibit strong localization on real-world graphs and hence the change in eigenvalues empirically does not scale with N (or, rather, $\sqrt{N}$). By ordering the eigenvalues we see that, empirically, the typical thresholds for sparsification have almost no effect on the eigenvalues, as shown in Fig. 2b and in the close-up in Fig. 11 in App. B.2. We find that the small changes caused by sparsification mostly affect the highest and lowest eigenvalues. The former correspond to very large clusters and long-range interactions, which are undesired for local graph smoothing. The latter correspond to spurious oscillations, which are not helpful for graph learning either and most likely affected because of the abrupt cutoff at ϵ.

4. Transition matrix on $\tilde{\boldsymbol{S}}$. As a final step we calculate the transition matrix on the resulting graph $\tilde{\boldsymbol{S}}$. This step does not just change which Laplacian we consider since we have already switched to using the transition matrix in step 1. It furthermore does not preserve the eigenvectors and is thus again best investigated empirically by ordering the eigenvalues. Fig. 2c shows that, empirically, this step slightly dampens low eigenvalues. This may seem counterproductive. However, the main purpose of using the transition matrix is ensuring that sparsification does not cause nodes to be treated differently by losing a different number of adjacent edges. The filtering is only a side-effect.

Limitations of spectral-based models. While there are tight connections between GDC and spectral-based models, GDC is actually spatial-based and therefore does not share their limitations. Similar to polynomial filters, GDC does not compute an expensive eigenvalue decomposition, preserves locality on the graph and is not limited to a single graph after training, i.e. typically the same coefficients θ_k

can be used across graphs. The choice of coefficients θ_k depends on the type of graph at hand and does not change significantly between similar graphs. Moreover, the hyperparameters α of PPR and t of the heat kernel usually fall within a narrow range that is rather insensitive to both the graph and model (see Fig. 8 in Sec. 6).

5 Related work

Graph diffusion and random walks have been extensively studied in classical graph learning [13, 14, 36, 37], especially for clustering [34], semi-supervised classification [12, 22], and recommendation systems [44]. For an overview of existing methods see Masuda et al. [46] and Fouss et al. [22].

The first models similar in structure to current Graph Neural Networks (GNNs) were proposed by Sperduti & Starita [68] and Baskin et al. [6], and the name GNN first appeared in [25, 65]. However, they only became widely adopted in recent years, when they started to outperform classical models in many graph-related tasks [19, 23, 42, 82]. In general, GNNs are classified into spectral-based models [11, 16, 29, 33, 41], which are based on the eigendecomposition of the graph Laplacian, and spatial-based methods [24, 27, 43, 52, 56, 62, 74], which use the graph directly and form new representations by aggregating the representations of a node and its neighbors. However, this distinction is often rather blurry and many models can not be clearly attributed to one type or the other. Deep learning also inspired a variety of unsupervised node embedding methods. Most models use random walks to learn node embeddings in a similar fashion as word2vec [51] [26, 61] and have been shown to implicitly perform a matrix factorization [63]. Other unsupervised models learn Gaussian distributions instead of vectors [10], use an auto-encoder [32], or train an encoder by maximizing the mutual information between local and global embeddings [73].

There have been some isolated efforts of using extended neighborhoods for aggregation in GNNs and graph diffusion for node embeddings. PPNP [23] propagates the node predictions generated by a neural network using personalized PageRank, DCNN [4] extends node features by concatenating features aggregated using the transition matrices of k-hop random walks, GraphHeat [79] uses the heat kernel and PAN [45] the transition matrix of maximal entropy random walks to aggregate over nodes in each layer, PinSage [82] uses random walks for neighborhood aggregation, and MixHop [2] concatenates embeddings aggregated using the transition matrices of k-hop random walks before each layer. VERSE [71] learns node embeddings by minimizing KL-divergence from the PPR matrix to a low-rank approximation. Attention walk [1] uses a similar loss to jointly optimize the node embeddings and diffusion coefficients θ_k. None of these works considered sparsification, generalized graph diffusion, spectral properties, or using preprocessing to generalize across models.

6 Experimental results

Experimental setup. We take extensive measures to prevent any kind of bias in our results. We optimize the hyperparameters of *all* models on *all* datasets with both the unmodified graph and all GDC variants *separately* using a combination of grid and random search on the validation set. Each result is averaged across 100 data splits and random initializations for supervised tasks and 20 random initializations for unsupervised tasks, as suggested by Gasteiger et al. [23] and Shchur et al. [67]. We report performance on a test set that was used exactly *once*. We report all results as averages with 95 % confidence intervals calculated via bootstrapping.

We use the symmetric transition matrix with self-loops $\tilde{\boldsymbol{T}}_{\text{sym}} = (\boldsymbol{I}_N + \boldsymbol{D})^{-1/2}(\boldsymbol{I}_N + \boldsymbol{A})(\boldsymbol{I}_N + \boldsymbol{D})^{-1/2}$ for GDC and the column-stochastic transition matrix $\boldsymbol{T}_{\text{rw}}^{\tilde{\boldsymbol{S}}} = \tilde{\boldsymbol{S}}\boldsymbol{D}_{\tilde{\boldsymbol{S}}}^{-1}$ on $\tilde{\boldsymbol{S}}$. We present two simple and effective choices for the coefficients θ_k: The heat kernel and PPR. The diffusion matrix $\boldsymbol{S}$ is sparsified using either an ϵ-threshold or top-k.

Datasets and models. We evaluate GDC on six datasets: The citation graphs CITESEER [66], CORA [48], and PUBMED [53], the co-author graph COAUTHOR CS [67], and the co-purchase graphs AMAZON COMPUTERS and AMAZON PHOTO [47, 67]. We only use their largest connected components. We show how GDC impacts the performance of 9 models: Graph Convolutional Network (GCN) [33], Graph Attention Network (GAT) [74], jumping knowledge network (JK) [80], Graph Isomorphism Network (GIN) [81], and ARMA [9] are supervised models. The degree-corrected stochastic block model (DCSBM) [31], spectral clustering (using $\boldsymbol{L}_{\text{sym}}$) [55], DeepWalk

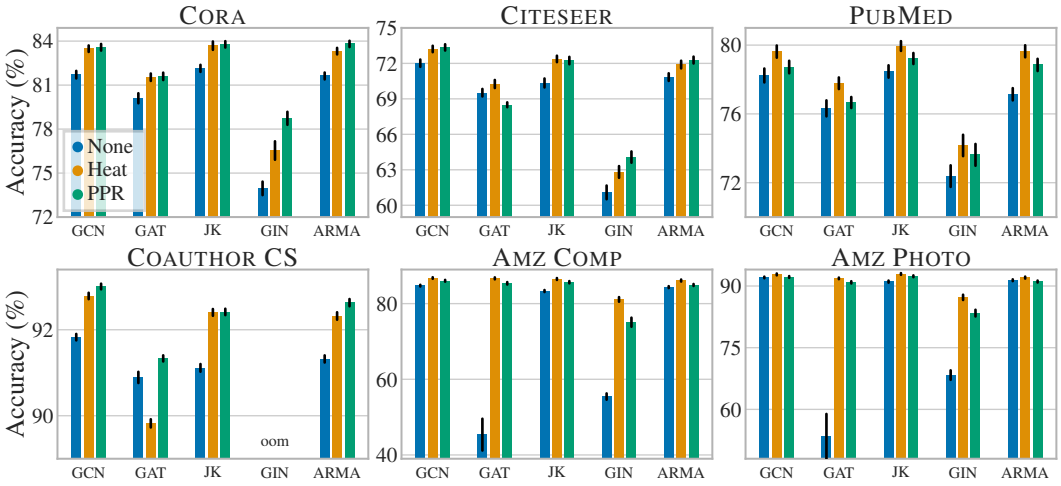


Figure 3: Node classification accuracy of GNNs with and without GDC. GDC consistently improves accuracy across models and datasets. It is able to fix models whose accuracy otherwise breaks down.

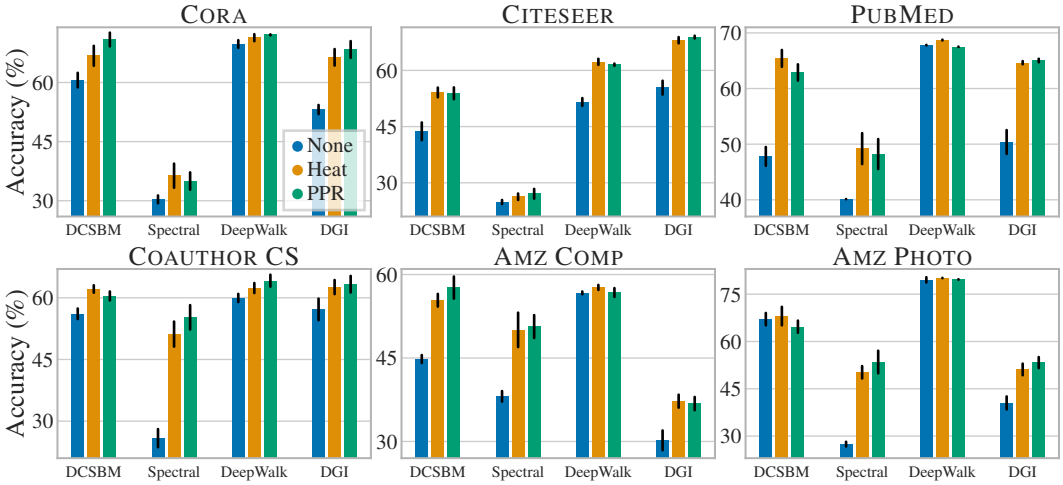


Figure 4: Clustering accuracy with and without GDC. GDC consistently improves the accuracy across a diverse set of models and datasets.

[61], and Deep Graph Infomax (DGI) [73] are unsupervised models. Note that DGI uses node features while other unsupervised models do not. We use k-means clustering to generate clusters from node embeddings. Dataset statistics and hyperparameters are reported in App. B.

Semi-supervised node classification. In this task the goal is to label nodes based on the graph, node features $\boldsymbol{X} \in \mathbb{R}^{N \times F}$ and a subset of labeled nodes $\boldsymbol{y}$. The main goal of GDC is improving the performance of MPNN models. Fig. 3 shows that GDC consistently and significantly improves the accuracy of a wide variety of state-of-the-art models across multiple diverse datasets. Note how GDC is able to fix the performance of GNNs that otherwise break down on some datasets (e.g. GAT). We also surpass or match the previous state of the art on all datasets investigated (see App. B.2).

Clustering. We highlight GDC's ability to be combined with any graph-based model by reporting the performance of a diverse set of models that use a wide range of paradigms. Fig. 4 shows the unsupervised accuracy obtained by matching clusters to ground-truth classes using the Hungarian algorithm. Accuracy consistently and significantly improves for all models and datasets. Note that spectral clustering uses the graph's eigenvectors, which are not affected by the diffusion step itself. Still, its performance improves by up to 30 percentage points. Results in tabular form are presented in App. B.2.

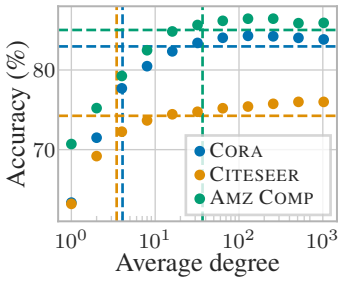


Figure 5: GCN+GDC accuracy (using PPR and top-k). Lines indicate original accuracy and degree. GDC surpasses original accuracy at around the same degree independent of dataset. Sparsification often improves accuracy.

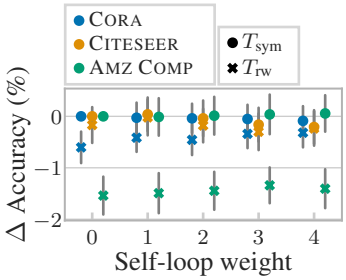


Figure 6: Difference in GCN+GDC accuracy (using PPR and top-k, percentage points) compared to the symmetric $\boldsymbol{T}_{\text{sym}}$ without self-loops. $\boldsymbol{T}_{\text{rw}}$ performs worse and self-loops have no significant effect.

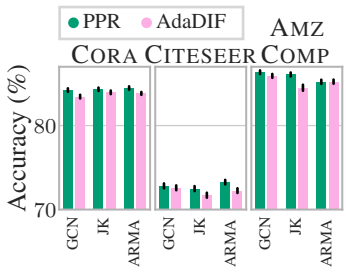


Figure 7: Accuracy of GDC with coefficients θ_k defined by PPR and learned by AdaDIF. Simple PPR coefficients consistently perform better than those obtained by AdaDIF, even with optimized regularization.

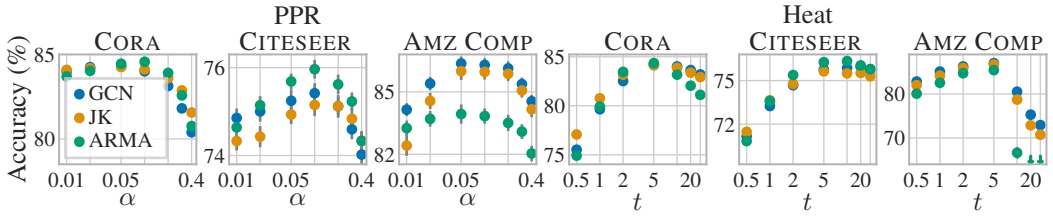


Figure 8: Accuracy achieved by using GDC with varying hyperparameters of PPR (α) and the heat kernel (t). Optimal values fall within a narrow range that is consistent across datasets and models.

In this work we concentrate on node-level prediction tasks in a transductive setting. However, GDC can just as easily be applied to inductive problems or different tasks like graph classification. In our experiments we found promising, yet not as consistent results for graph classification (e.g. 2.5 percentage points with GCN on the DD dataset [18]). We found no improvement for the inductive setting on PPI [50], which is rather unsurprising since the underlying data used for graph construction already includes graph diffusion-like mechanisms (e.g. regulatory interactions, protein complexes, and metabolic enzyme-coupled interactions). We furthermore conducted experiments to answer five important questions:

Does GDC increase graph density? When sparsifying the generalized graph diffusion matrix $\boldsymbol{S}$ we are free to choose the resulting level of sparsity in $\tilde{\boldsymbol{S}}$. Fig. 5 indicates that, surprisingly, GDC requires roughly the same average degree to surpass the performance of the original graph independent of the dataset and its average degree (ϵ-threshold in App. B.2, Fig. 12). This suggests that the sparsification hyperparameter can be obtained from a fixed average degree. Note that CORA and CITESEER are both small graphs with low average degree. Graphs become denser with size [39] and in practice we expect GDC to typically *reduce* the average degree at constant accuracy. Fig. 5 furthermore shows that there is an optimal degree of sparsity above which the accuracy decreases. This indicates that sparsification is not only computationally beneficial but also improves prediction performance.

How to choose the transition matrix $\boldsymbol{T}$? We found $\boldsymbol{T}_{\text{sym}}$ to perform best across datasets. More specifically, Fig. 6 shows that the symmetric version on average outperforms the random walk transition matrix $\boldsymbol{T}_{\text{rw}}$. This figure also shows that GCN accuracy is largely insensitive to self-loops when using $\boldsymbol{T}_{\text{sym}}$ – all changes lie within the estimated uncertainty. However, we did find that other models, e.g. GAT, perform better with self-loops (not shown).

How to choose the coefficients θ_k? We found the coefficients defined by PPR and the heat kernel to be effective choices for θ_k. Fig. 8 shows that their optimal hyperparameters typically fall within a narrow range of $\alpha \in [0.05, 0.2]$ and $t \in [1, 10]$. We also tried obtaining θ_k from models that learn analogous coefficients [1, 8, 13]. However, we found that θ_k obtained by these models tend to converge to a minimal neighborhood, i.e. they converge to $\theta_0 \approx 1$ or $\theta_1 \approx 1$ and all other $\theta_k \approx 0$.

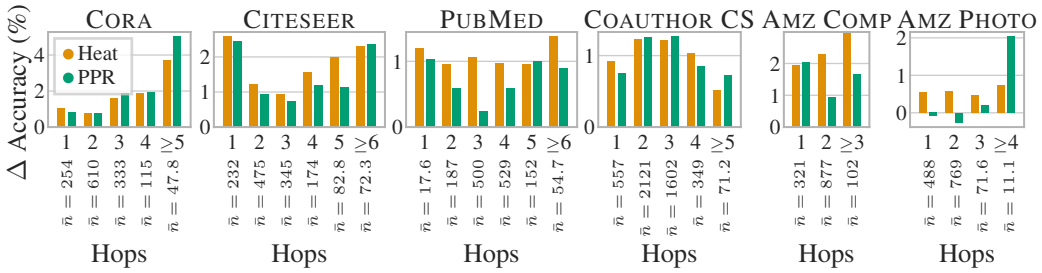


Figure 10: Improvement (percentage points) in GCN accuracy by adding GDC depending on distance (number of hops) from the training set. Nodes further away tend to benefit more from GDC.

This is caused by their training losses almost always decreasing when the considered neighborhood shrinks. We were able to control this overfitting to some degree using strong regularization (specifically, we found L_2 regularization on the difference of neighboring coefficients $\theta_{k+1} - \theta_k$ to perform best). However, this requires hand-tuning the regularization for every dataset, which defeats the purpose of *learning* the coefficients from the graph. Moreover, we found that even with hand-tuned regularization the coefficients defined by PPR and the heat kernel perform better than trained θ_k, as shown in Fig. 7.

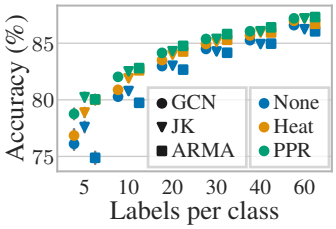


Figure 9: Accuracy on Cora with different label rates. Improvement from GDC increases for sparser label rates.

How does the label rate affect GDC? When varying the label rate from 5 up to 60 labels per class we found that the improvement achieved by GDC increases the sparser the labels are. Still, GDC improves performance even for 60 labels per class, i.e. 17 % label rate (see Fig. 9). This trend is most likely due to larger neighborhood leveraged by GDC.

Which nodes benefit from GDC? Our experiments showed no correlation of improvement with most common node properties, except for the distance from the training set. Nodes further away from the training set tend to benefit more from GDC, as demonstrated by Fig. 10. Besides smoothing out the neighborhood, GDC also has the effect of increasing the model's range, since it is no longer restricted to only using first-hop neighbors. Hence, nodes further away from the training set influence the training and later benefit from the improved model weights.

7 Conclusion

We propose graph diffusion convolution (GDC), a method based on sparsified generalized graph diffusion. GDC is a more powerful, yet spatially localized extension of message passing in GNNs, but able to enhance any graph-based model. We show the tight connection between GDC and spectral-based models and analyzed GDC's spectral properties. GDC shares many of the strengths of spectral methods and none of their weaknesses. We conduct extensive and rigorous experiments that show that GDC consistently improves the accuracy of a wide range of models on both supervised and unsupervised tasks across various homophilic datasets and requires very little hyperparameter tuning. There are many extensions and applications of GDC that remain to be explored. We expect many graph-based models and tasks to benefit from GDC, e.g. graph classification and regression. Promising extensions include other diffusion coefficients θ_k such as those given by the methods presented in Fouss et al. [22] and more advanced random walks and operators that are not defined by powers of a transition matrix.

Acknowledgments

This research was supported by the German Federal Ministry of Education and Research (BMBF), grant no. 01IS18036B, and by the Deutsche Forschungsgemeinschaft (DFG) through the Emmy Noether grant GU 1409/2-1 and the TUM International Graduate School of Science and Engineering (IGSSE), GSC 81. The authors of this work take full responsibilities for its content.

References

[1] Sami Abu-El-Haija, Bryan Perozzi, Rami Al-Rfou, and Alex Alemi. Watch Your Step: Learning Node Embeddings via Graph Attention. In *NeurIPS*, 2018.

[2] Sami Abu-El-Haija, Bryan Perozzi, Amol Kapoor, Nazanin Alipourfard, Kristina Lerman, Hrayr Harutyunyan, Greg Ver Steeg, and Aram Galstyan. MixHop: Higher-Order Graph Convolutional Architectures via Sparsified Neighborhood Mixing. In *ICML*, 2019.

[3] R. Andersen, F. Chung, and K. Lang. Local Graph Partitioning using PageRank Vectors. In *FOCS*, 2006.

[4] James Atwood and Don Towsley. Diffusion-Convolutional Neural Networks. In *NIPS*, 2016.

[5] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. Four degrees of separation. In *ACM Web Science Conference*, 2012.

[6] Igor I. Baskin, Vladimir A. Palyulin, and Nikolai S. Zefirov. A Neural Device for Searching Direct Correlations between Structures and Properties of Chemical Compounds. *Journal of Chemical Information and Computer Sciences*, 37(4):715–721, 1997.

[7] Dimitris Berberidis and Georgios B. Giannakis. Node Embedding with Adaptive Similarities for Scalable Learning over Graphs. *CoRR*, 1811.10797, 2018.

[8] Dimitris Berberidis, Athanasios N. Nikolakopoulos, and Georgios B. Giannakis. Adaptive diffusions for scalable learning over graphs. *IEEE Transactions on Signal Processing*, 67(5):1307–1321, 2019.

[9] Filippo Maria Bianchi, Daniele Grattarola, Lorenzo Livi, and Cesare Alippi. Graph Neural Networks with convolutional ARMA filters. *CoRR*, 1901.01343, 2019.

[10] Aleksandar Bojchevski and Stephan Günnemann. Deep Gaussian Embedding of Graphs: Unsupervised Inductive Learning via Ranking. *ICLR*, 2018.

[11] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral Networks and Deep Locally Connected Networks on Graphs. In *ICLR*, 2014.

[12] Eliav Buchnik and Edith Cohen. Bootstrapped Graph Diffusions: Exposing the Power of Nonlinearity. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, 2(1):1–19, 2018.

[13] Siheng Chen, Aliaksei Sandryhaila, Jose M. F. Moura, and Jelena Kovacevic. Adaptive graph filtering: Multiresolution classification on graphs. In *IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, 2013.

[14] F. Chung. The heat kernel as the pagerank of a graph. *Proceedings of the National Academy of Sciences*, 104(50):19735–19740, 2007.

[15] Aurelien Decelle, Florent Krzakala, Cristopher Moore, and Lenka Zdeborová. Inference and phase transitions in the detection of modules in sparse networks. *Physical Review Letters*, 107(6):065701, 2011.

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

[17] Tyler Derr, Yao Ma, and Jiliang Tang. Signed Graph Convolutional Networks. In *ICDM*, 2018.

[18] Paul D. Dobson and Andrew J. Doig. Distinguishing enzyme structures from non-enzymes without alignments. *Journal of Molecular Biology*, 330(4):771–783, 2003.

[19] David K. Duvenaud, Dougal Maclaurin, Jorge Aguilera-Iparraguirre, Rafael Gómez-Bombarelli, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P. Adams. Convolutional Networks on Graphs for Learning Molecular Fingerprints. In *NIPS*, 2015.

[20] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *LREC 2010 Workshop on New Challenges for NLP Frameworks*, 2010.

[21] Matthias Fey and Jan E. Lenssen. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR workshop*, 2019.

[22] François Fouss, Kevin Francoisse, Luh Yen, Alain Pirotte, and Marco Saerens. An experimental investigation of kernels on graphs for collaborative recommendation and semisupervised classification. *Neural Networks*, 31:53–72, 2012.

[23] Johannes Gasteiger, Aleksandar Bojchevski, and Stephan Günnemann. Predict then Propagate: Graph Neural Networks Meet Personalized PageRank. In *ICLR*, 2019.

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

[25] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *IEEE International Joint Conference on Neural Networks*, 2005.

[26] Aditya Grover and Jure Leskovec. node2vec: Scalable Feature Learning for Networks. In *KDD*, 2016.

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

[28] David K. Hammond, Pierre Vandergheynst, and Rémi Gribonval. Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis*, 30(2):129–150, 2011.

[29] Mikael Henaff, Joan Bruna, and Yann LeCun. Deep Convolutional Networks on Graph-Structured Data. *CoRR*, 1506.05163, 2015.

[30] Eric Jones, Travis Oliphant, Pearu Peterson, and others. *SciPy: Open source scientific tools for Python*. 2001.

[31] Brian Karrer and Mark EJ Newman. Stochastic blockmodels and community structure in networks. *Physical review E*, 83(1):016107, 2011.

[32] Thomas N. Kipf and Max Welling. Variational Graph Auto-Encoders. In *NIPS workshop*, 2016.

[33] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*, 2017.

[34] Kyle Kloster and David F Gleich. Heat kernel based community detection. In *KDD*, 2014.

[35] Isabel M. Kloumann, Johan Ugander, and Jon Kleinberg. Block models and personalized PageRank. *Proceedings of the National Academy of Sciences*, 114(1):33–38, 2017.

[36] Risi Imre Kondor and John Lafferty. Diffusion kernels on graphs and other discrete structures. In *ICML*, 2002.

[37] Stéphane Lafon and Ann B. Lee. Diffusion Maps and Coarse-Graining: A Unified Framework for Dimensionality Reduction, Graph Partitioning, and Data Set Parameterization. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(9):1393–1403, 2006.

[38] Edmund Landau. Zur relativen Wertbemessung der Turnierresultate. *Deutsches Wochenschach*, 11: 366–369, 1895.

[39] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *KDD*, 2005.

[40] Pan Li, Eli Chien, and Olgica Milenkovic. Optimizing generalized pagerank methods for seed-expansion community detection. In *NeurIPS*, 2019.

[41] Ruoyu Li, Sheng Wang, Feiyun Zhu, and Junzhou Huang. Adaptive Graph Convolutional Neural Networks. In *AAAI*, 2018.

[42] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. Diffusion Convolutional Recurrent Neural Network: Data-Driven Traffic Forecasting. In *ICLR*, 2018.

[43] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated Graph Sequence Neural Networks. In *ICLR*, 2016.

[44] Jeremy Ma, Weiyu Huang, Santiago Segarra, and Alejandro Ribeiro. Diffusion filtering of graph signals and its use in recommendation systems. In *ICASSP*, 2016.

[45] Zheng Ma, Ming Li, and Yuguang Wang. PAN: Path Integral Based Convolution for Deep Graph Neural Networks. In *ICML workshop*, 2019.

[46] Naoki Masuda, Mason A Porter, and Renaud Lambiotte. Random walks and diffusion on networks. *Physics reports*, 716:1–58, 2017.

[47] Julian J. McAuley, Christopher Targett, Qinfeng Shi, and Anton van den Hengel. Image-Based Recommendations on Styles and Substitutes. In *SIGIR*, 2015.

[48] Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval*, 3(2):127–163, 2000.

[49] Miller McPherson, Lynn Smith-Lovin, and James M Cook. Birds of a feather: Homophily in social networks. *Annual review of sociology*, 27(1):415–444, 2001.

[50] Jörg Menche, Amitabh Sharma, Maksim Kitsak, Susan Ghiassian, Marc Vidal, Joseph Loscalzo, and Albert-László Barabási. Uncovering disease-disease relationships through the incomplete human interactome. *Science*, 347(6224):1257601, 2015.

[51] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. In *NIPS*, 2013.

[52] Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodola, Jan Svoboda, and Michael M. Bronstein. Geometric Deep Learning on Graphs and Manifolds Using Mixture Model CNNs. In *CVPR*, 2017.

[53] Galileo Namata, Ben London, Lise Getoor, and Bert Huang. Query-driven Active Surveying for Collective Classification. In *International Workshop on Mining and Learning with Graphs (MLG), KDD*, 2012.

[54] Huda Nassar, Kyle Kloster, and David F. Gleich. Strong Localization in Personalized PageRank Vectors. In *International Workshop on Algorithms and Models for the Web Graph (WAW)*, 2015.

[55] Andrew Y Ng, Michael I Jordan, and Yair Weiss. On Spectral Clustering: Analysis and an algorithm. In *NIPS*, 2002.

[56] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning Convolutional Neural Networks for Graphs. In *ICML*, 2016.

[57] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Report, Stanford InfoLab, 1998.

[58] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS workshop*, 2017.

[59] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[60] Tiago P. Peixoto. The graph-tool python library. *figshare*, 2014.

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

[62] Trang Pham, Truyen Tran, Dinh Q. Phung, and Svetha Venkatesh. Column Networks for Collective Classification. In *AAAI*, 2017.

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

[64] Stephen Ragain. Community Detection via Discriminant functions for Random Walks in the degree-corrected Stochastic Block Model. Report, Stanford University, 2017.

[65] F. Scarselli, M. Gori, Ah Chung Tsoi, M. Hagenbuchner, and G. Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[66] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective Classification in Network Data. *AI Magazine*, 29(3):93–106, 2008.

[67] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of Graph Neural Network Evaluation. In *NIPS workshop*, 2018.

[68] A. Sperduti and A. Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, 1997.

[69] Gilbert Wright Stewart and Ji-guang Sun. *Matrix Perturbation Theory*. Computer Science and Scientific Computing. 1990.

[70] Yu-Hang Tang, Dongkun Zhang, and George Em Karniadakis. An atomistic fingerprint algorithm for learning ab initio molecular force fields. *The Journal of Chemical Physics*, 148(3):034101, 2018.

[71] Anton Tsitsulin, Davide Mottin, Panagiotis Karras, and Emmanuel Müller. VERSE: Versatile Graph Embeddings from Similarity Measures. In *WWW*, 2018.

[72] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.

[73] Petar Velickovic, William Fedus, William L. Hamilton, Pietro Liò, Yoshua Bengio, and R. Devon Hjelm. Deep Graph Infomax. In *ICLR*, 2019.

[74] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. In *ICLR*, 2018.

[75] Sebastiano Vigna. Spectral ranking. *Network Science, CoRR (updated, 0912.0238v15)*, 4(4):433–445, 2016.

[76] Ulrike von Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, 2007.

[77] Zhewei Wei, Xiaodong He, Xiaokui Xiao, Sibo Wang, Shuo Shang, and Ji-Rong Wen. TopPPR: Top-k Personalized PageRank Queries with Precision Guarantees on Large Graphs. In *SIGMOD*, 2018.

[78] Felix Wu, Tianyi Zhang, Amauri Holanda de Souza Jr., Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. Simplifying Graph Convolutional Networks. In *ICML*, 2019.

[79] Bingbing Xu, Huawei Shen, Qi Cao, Keting Cen, and Xueqi Cheng. Graph Convolutional Networks using Heat Kernel for Semi-supervised Learning. In *IJCAI*, 2019.

[80] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation Learning on Graphs with Jumping Knowledge Networks. In *ICML*, 2018.

[81] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How Powerful are Graph Neural Networks? In *ICLR*, 2019.

[82] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. *KDD*, 2018.

Representation Learning on Graphs with Jumping Knowledge Networks

Keyulu Xu [1] **Chengtao Li** [1] **Yonglong Tian** [1] **Tomohiro Sonobe** [2]
Ken-ichi Kawarabayashi [2] **Stefanie Jegelka** [1]

Abstract

Recent deep learning approaches for representation learning on graphs follow a neighborhood aggregation procedure. We analyze some important properties of these models, and propose a strategy to overcome those. In particular, the range of "neighboring" nodes that a node's representation draws from strongly depends on the graph structure, analogous to the spread of a random walk. To adapt to local neighborhood properties and tasks, we explore an architecture – jumping knowledge (JK) networks – that flexibly leverages, for each node, different neighborhood ranges to enable better structure-aware representation. In a number of experiments on social, bioinformatics and citation networks, we demonstrate that our model achieves state-of-the-art performance. Furthermore, combining the JK framework with models like Graph Convolutional Networks, GraphSAGE and Graph Attention Networks consistently improves those models' performance.

1. Introduction

Graphs are a ubiquitous structure that widely occurs in data analysis problems. Real-world graphs such as social networks, financial networks, biological networks and citation networks represent important rich information which is not seen from the individual entities alone, for example, the communities a person is in, the functional role of a molecule, and the sensitivity of the assets of an enterprise to external shocks. Therefore, representation learning of nodes in graphs aims to extract high-level features from a node as well as its neighborhood, and has proved extremely useful for many applications, such as node classification, clustering, and link prediction (Perozzi et al., 2014; Monti et al., 2017; Grover & Leskovec, 2016; Tang et al., 2015).

Recent works focus on deep learning approaches to node representation. Many of these approaches broadly follow a neighborhood aggregation (or "message passing" scheme), and those have been very promising (Kipf & Welling, 2017; Hamilton et al., 2017; Gilmer et al., 2017; Veličković et al., 2018; Kearnes et al., 2016). These models learn to iteratively aggregate the hidden features of every node in the graph with its adjacent nodes' as its new hidden features, where an iteration is parametrized by a layer of the neural network. Theoretically, an aggregation process of k iterations makes use of the subtree structures of height k rooted at every node. Such schemes have been shown to generalize the Weisfeiler-Lehman graph isomorphism test (Weisfeiler & Lehman, 1968) enabling to simultaneously learn the topology as well as the distribution of node features in the neighborhood (Shervashidze et al., 2011; Kipf & Welling, 2017; Hamilton et al., 2017).

Yet, such aggregation schemes sometimes lead to surprises. For example, it has been observed that the best performance with one of the state-of-the-art models, Graph Convolutional Networks (GCN), is achieved with a 2-layer model. Deeper versions of the model that, in principle, have access to more information, perform worse (Kipf & Welling, 2017). A similar degradation of learning for computer vision problems is resolved by residual connections (He et al., 2016a) that greatly aid the training of deep models. But, even with residual connections, GCNs with more layers do not perform as well as the 2-layer GCN on many datasets, e.g. citation networks.

Motivated by observations like the above, in this paper, we address two questions. First, we study properties and resulting limitations of neighborhood aggregation schemes. Second, based on this analysis, we propose an architecture that, as opposed to existing models, enables adaptive, *structure-aware* representations. Such representations are particularly interesting for representation learning on large complex graphs with diverse subgraph structures.

Model analysis. To better understand the behavior of different neighborhood aggregation schemes, we analyze the effective range of nodes that any given node's representation draws from. We summarize this sensitivity analysis by what

[1]Massachusetts Institute of Technology (MIT) [2]National Institute of Informatics, Tokyo. Correspondence to: Keyulu Xu <keyulu@mit.edu>, Stefanie Jegelka <stefje@mit.edu>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

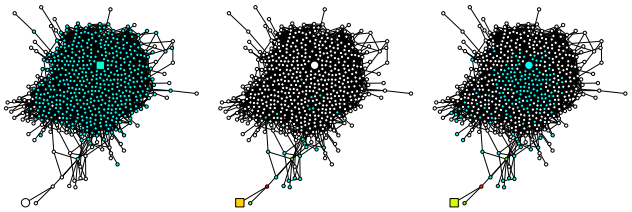
we name the *influence distribution* of a node. This effective range implicitly encodes prior assumptions on what are the "nearest neighbors" that a node should draw information from. In particular, we will see that this influence is heavily affected by the graph structure, raising the question whether "one size fits all", in particular in graphs whose subgraphs have varying properties (such as more tree-like or more expander-like).

In particular, our more formal analysis connects influence distributions with the spread of a random walk at a given node, a well-understood phenomenon as a function of the graph structure and eigenvalues (Lovász, 1993). For instance, in some cases and applications, a 2-step random walk influence that focuses on local neighborhoods can be more informative than higher-order features where some of the information may be "washed out" via averaging.

Changing locality. To illustrate the effect and importance of graph structure, recall that many real-world graphs possess locally strongly varying structure. In biological and citation networks, the majority of the nodes have few connections, whereas some nodes (hubs) are connected to many other nodes. Social and web networks usually consist of an expander-like core part and an almost-tree (bounded treewidth) part, which represent well-connected entities and the small communities respectively (Leskovec et al., 2009; Maehara et al., 2014; Tsonis et al., 2006).

Besides node features, this subgraph structure has great impact on the result of neighborhood aggregation. The speed of expansion or, equivalently, growth of the influence radius, is characterized by the random walk's mixing time, which changes dramatically on subgraphs with different structures (Lovász, 1993). Thus, the same number of iterations (layers) can lead to influence distributions of very different locality. As an example, consider the social network in Figure 1 from GooglePlus (Leskovec & Mcauley, 2012). The figure illustrates the expansions of a random walk starting at the square node. The walk (a) from a node within the core rapidly includes almost the entire graph. In contrast, the walk (b) starting at a node in the tree part includes only a very small fraction of all nodes. After 5 steps, the same walk has reached the core and, suddenly, spreads quickly. Translated to graph representation models, these spreads become the influence distributions or, in other words, the averaged features yield the new feature of the walk's starting node. This shows that in the same graph, the same number of steps can lead to very different effects. Depending on the application, wide-range or small-range feature combinations may be more desirable. A too rapid expansion may average too broadly and thereby lose information, while in other parts of the graph, a sufficient neighborhood may be needed for stabilizing predictions.

JK networks. The above observations raise the question whether it is possible to adaptively *adjust* (i.e., learn) the influence radii for each node and task. To achieve this, we explore an architecture that learns to selectively exploit information from neighborhoods of differing locality. This architecture selectively combines different aggregations at the last layer, i.e., the representations "jump" to the last layer. Hence, we name the resulting networks *Jumping Knowledge Networks (JK-Nets)*. We will see that empirically, when adaptation is an option, the networks indeed learn representations of different orders for different graph substructures. Moreover, in Section 6, we show that applying our framework to various state-of-the-art neighborhood-aggregation models consistently improves their performance.



(a) 4 steps at core (b) 4 steps at tree (c) 5 steps at tree

Figure 1. Expansion of a random walk (and hence influence distribution) starting at (square) nodes in subgraphs with different structures. Different subgraph structures result in very different neighborhood sizes.

2. Background and Neighborhood aggregation schemes

We begin by summarizing some of the most common neighborhood aggregation schemes and, along the way, introduce our notation. Let $G = (V, E)$ be a simple graph with node features $X_v \in \mathbb{R}^{d_i}$ for $v \in V$. Let $\widetilde{G}$ be the graph obtained by adding a self-loop to every $v \in V$. The hidden feature of node v learned by the l-th layer of the model is denoted by $h_v^{(l)} \in \mathbb{R}^{d_h}$. Here, d_i is the dimension of the input features and d_h is the dimension of the hidden features, which, for simplicity of exposition, we assume to be the same across layers. We also use $h_v^{(0)} = X_v$ for the node feature. The neighborhood $N(v) = \{u \in V | (v, u) \in E\}$ of node v is the set of adjacent nodes of v. The analogous neighborhood $\widetilde{N}(v) = \{v\} \cup \{u \in V | (v, u) \in E\}$ on $\widetilde{G}$ includes v.

A typical neighborhood aggregation scheme can generically be written as follows: for a k-layer model, the l-th layer ($l = 1..k$) updates $h_v^{(l)}$ for every $v \in V$ simultaneously as

$$h_v^{(l)} = \sigma\left(W_l \cdot \text{AGGREGATE}\left(\left\{h_u^{(l-1)}, \forall u \in \widetilde{N}(v)\right\}\right)\right) \tag{1}$$

where AGGREGATE is an aggregation function defined by the specific model, W_l is a trainable weight matrix on the l-th layer shared by all nodes, and σ is a non-linear activation function, e.g. a ReLU.

Graph Convolutional Networks (GCN). Graph Convolutional Networks (GCN) (Kipf & Welling, 2017), initially motivated by spectral graph convolutions (Hammond et al., 2011; Defferrard et al., 2016), are a specific instantiation of this framework (Gilmer et al., 2017), of the form

$$h_v^{(l)} = \text{ReLU}\Big(W_l \cdot \sum_{u \in \widetilde{N}(v)} (\deg(v)\deg(u))^{-1/2} h_u^{(l-1)}\Big) \tag{2}$$

where $\deg(v)$ is the degree of node v in G. Hamilton et al. (2017) derived a variant of GCN that also works in inductive settings (previously unseen nodes), by using a different normalization to average:

$$h_v^{(l)} = \text{ReLU}\Big(W_l \cdot \frac{1}{\widetilde{\deg}(v)} \sum_{u \in \widetilde{N}(v)} h_u^{(l-1)}\Big) \tag{3}$$

where $\widetilde{\deg}(v)$ is the degree of node v in $\widetilde{G}$.

Neighborhood Aggregation with Skip Connections. Instead of aggregating a node and its neighbors at the same time as in Eqn. (1), a number of recent approaches aggregate the neighbors first and then combine the resulting neighborhood representation with the node's representation from the last iteration. More formally, each node is updated as

$$h_{N(v)}^{(l)} = \sigma\left(W_l \cdot \text{AGGREGATE}_N\left(\{h_u^{(l-1)}, \forall u \in N(v)\}\right)\right)$$
$$h_v^{(l)} = \text{COMBINE}\left(h_v^{(l-1)}, h_{N(v)}^{(l)}\right)$$

where AGGREGATE_N and COMBINE are defined by the specific model. The COMBINE step is key to this paradigm and can be viewed as a form of a "skip connection" between different layers.For COMBINE, GraphSAGE (Hamilton et al., 2017) uses concatenation after a feature transform. Column Networks (Pham et al., 2017) interpolate the neighborhood representation and the node's previous representation, and Gated GNN (Li et al., 2016) uses the Gated Recurrent Unit (GRU) (Cho et al., 2014). Another well-known variant of skip connections, residual connections, use the identity mapping to help signals propagate (He et al., 2016a;b).

These skip connections are input- but not output-unit specific: If we "skip" a layer for $h_v^{(l)}$ (do not aggregate) or use a certain COMBINE, all subsequent units using this representation will be using this skip implicitly. It is impossible that a certain higher-up representation $h_u^{(l+j)}$ uses the skip and another one does not. As a result, skip connections cannot adaptively adjust the neighborhood sizes of the final-layer representations independently.

Neighborhood Aggregation with Directional Biases. Some recent models, rather than treating the features of adjacent nodes equally, weigh "important" neighbors more. This paradigm can be viewed as neighborhood-aggregation with directional biases because a node will be influenced by some directions of expansion more than the others.

Graph Attention Networks (GAT) (Veličković et al., 2018) and VAIN (Hoshen, 2017) learn to select the important neighbors via an attention mechanism. The max-pooling operation in GraphSAGE (Hamilton et al., 2017) implicitly selects the important nodes. This line of work is orthogonal to ours, because it modifies the direction of expansion whereas our model operates on the locality of expansion. Our model can be combined with these models to add representational power. In Section 6, we demonstrate that our framework works with not only simple neighborhood-aggregation models (GCN), but also with skip connections (GraphSAGE) and directional biases (GAT).

3. Influence Distribution and Random Walks

Next, we explore some important properties of the above aggregation schemes. Related to ideas of sensitivity analysis and influence functions in statistics (Koh & Liang, 2017) that measure the influence of a training point on parameters, we study the range of nodes whose features affect a given node's representation. This range gives insight into how large a neighborhood a node is drawing information from.

We measure the sensitivity of node x to node y, or the influence of y on x, by measuring how much a change in the input feature of y affects the representation of x in the last layer. For any node x, the *influence distribution* captures the relative influences of all other nodes.

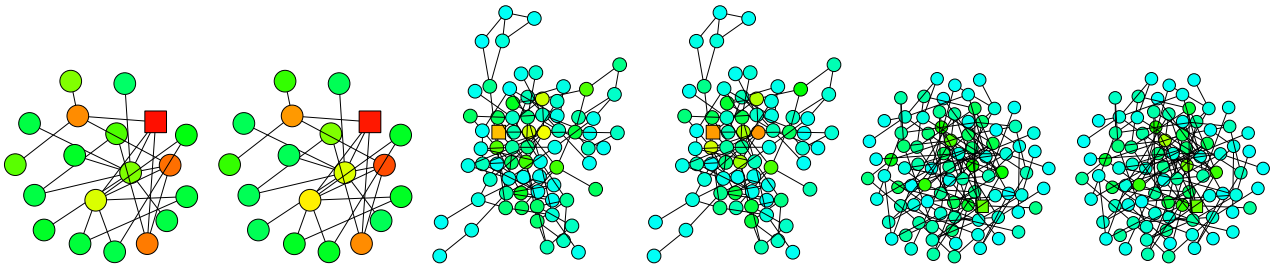
Definition 3.1 (Influence score and distribution)**.** *For a simple graph $G = (V, E)$, let $h_x^{(0)}$ be the input feature and $h_x^{(k)}$ be the learned hidden feature of node $x \in V$ at the k-th (last) layer of the model. The* influence score $I(x, y)$ *of node x by any node $y \in V$ is the sum of the absolute values of the entries of the Jacobian matrix $\left[\frac{\partial h_x^{(k)}}{\partial h_y^{(0)}}\right]$. We define the* influence distribution I_x *of $x \in V$ by normalizing the influence scores: $I_x(y) = I(x, y)/\sum_z I(x, z)$, or*

$$I_x(y) = e^T \left[\frac{\partial h_x^{(k)}}{\partial h_y^{(0)}}\right] e \Big/ \left(\sum_{z \in V} e^T \left[\frac{\partial h_x^{(k)}}{\partial h_z^{(0)}}\right] e\right)$$

where e is the all-ones vector.

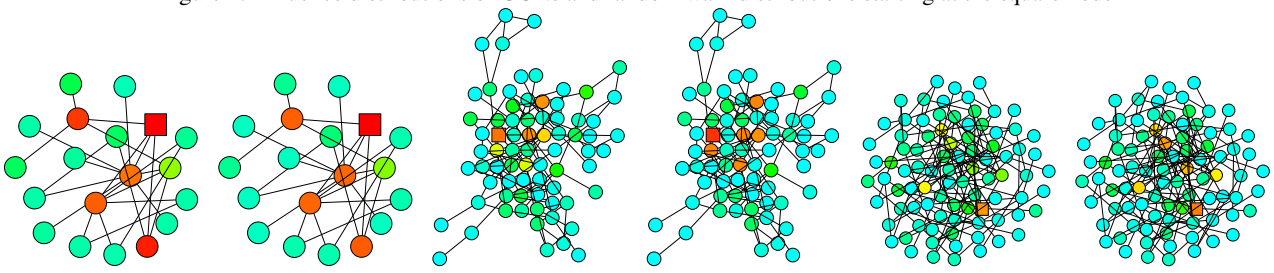
Later, we will see connections of influence distributions with random walks. For completeness, we also define random walk distributions.

Definition 3.2. *Consider a random walk on $\widetilde{G}$ starting at a node v_0; if at the t-th step we are at a node v_t, we move to any neighbor of v_t (including v_t) with equal probability.*



(a) 2 layer GCN (b) 2 step r.w. (c) 4 layer GCN (d) 4 step r.w. (e) 6 layer GCN (f) 6 step r.w.

Figure 2. Influence distributions of GCNs and random walk distributions starting at the square node



(a) 2 layer Res (b) 2 step lazy r.w. (c) 4 layer Res (d) 4 step lazy r.w. (e) 6 layer Res (f) 6 step lazy r.w.

Figure 3. Influence distributions of GCNs with residual connections and random walk distributions with lazy factor 0.4

The t-step random walk distribution P_t *of* v_0 *is*

$$P_t(i) = \textit{Prob}(v_t = i). \tag{4}$$

Analogous definitions apply for random walks with non-uniform transition probabilities.

An important property of the random walk distribution is that it becomes more spread out as t increases and converges to the limit distribution if the graph is non-bipartite. The rate of convergence depends on the structure of the subgraph and can be bounded by the spectral gap (or the conductance) of the random walk's transition matrix (Lovász, 1993).

3.1. Model Analysis

The influence distribution for different aggregation models and nodes can give insights into the information captured by the respective representations. The following results show that the influence distributions of common aggregation schemes are closely connected to random walk distributions. This observation hints at specific implications – strengths and weaknesses – that we will discuss.

With a randomization assumption of the ReLU activations similar to that in (Kawaguchi, 2016; Choromanska et al., 2015), we can draw connections between GCNs and random walks:

Theorem 1. *Given a* k*-layer GCN with averaging as in Equation* (3)*, assume that all paths in the computation graph of the model are activated with the same probability of success* ρ*. Then the influence distribution* I_x *for any node* $x \in V$ *is equivalent, in expectation, to the* k*-step random walk distribution on* $\widetilde{G}$ *starting at node* x*.*

We prove Theorem 1 in the appendix.

It is straightforward to modify the proof of Theorem 1 to show a nearly equivalent result for the version of GCN in Equation (2). The only difference is that each random walk path $v_p^0, v_p^1, ..., v_p^k$ from node x (v_p^0) to y (v_p^k), instead of probability $\rho \prod_{l=1}^{k} \frac{1}{\widetilde{\deg}(v_p^l)}$, now has probability $\frac{\rho}{Q} \prod_{l=1}^{k-1} \frac{1}{\widetilde{\deg}(v_p^l)} \cdot (\widetilde{\deg}(x)\widetilde{\deg}(y))^{-1/2}$, where Q is a normalizing factor. Thus, the difference in probability is small, especially when the degree of x and y are close.

Similarly, we can show that neighborhood aggregation schemes with directional biases resemble biased random walk distributions. This follows by substituting the corresponding probabilities into the proof of Theorem 1.

Empirically, we observe that, despite somewhat simplifying assumptions, our theory is close to what happens in practice. We visualize the heat maps of the influence distributions for a node (labeled square) for trained GCNs, and compare with the random walk distributions starting at the same node. Figure 2 shows example results. Darker colors correspond to higher influence probabilities. To show the effect of skip connections, Figure 3 visualizes the analogous heat maps for one example—GCN with residual connections. Indeed, we observe that the influence distributions of networks with residual connections approximately correspond to lazy random walks: each step has a higher probability of staying at

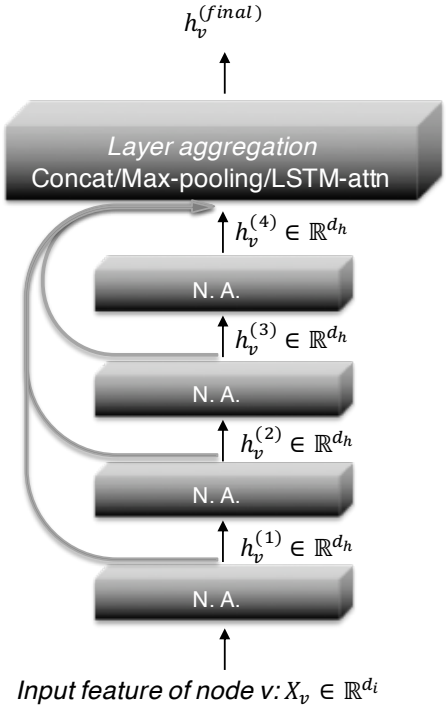


Figure 4. A 4-layer Jumping Knowledge Network (JK-Net). N.A. stands for neighborhood aggregation.

the current node. Local information is retained with similar probabilities for all nodes in each iteration; this cannot adapt to diverse needs of specific upper-layer nodes. Further visualizations may be found in the appendix.

Fast Collapse on Expanders. To better understand the implication of Theorem 1 and the limitations of the corresponding neighborhood aggregation algorithms, we revisit the scenario of learning on a social network shown in Figure 1. Random walks starting inside an expander converge rapidly in $O(\log |V|)$ steps to an almost-uniform distribution (Hoory et al., 2006). After $O(\log |V|)$ iterations of neighborhood aggregation, by Theorem 1 the representation of every node is influenced almost equally by any other node in the expander. Thus, the node representations will be representative of the global graph and carry limited information about individual nodes. In contrast, random walks starting at the bounded tree-width (almost-tree) part converge slowly, i.e., the features retain more local information. Models that impose a fixed random walk distribution inherit these discrepancies in the speed of expansion and influence neighborhoods, which may not lead to the best representations for all nodes.

4. Jumping Knowledge Networks

The above observations raise the question whether the fixed but structure-dependent influence radius size induced by common aggregation schemes really achieves the best representations for all nodes and tasks. Large radii may lead to too much averaging, while small radii may lead to instabilities or insufficient information aggregation. Hence, we propose two simple yet powerful architectural changes – jump connections and a subsequent selective but adaptive aggregation mechanism.

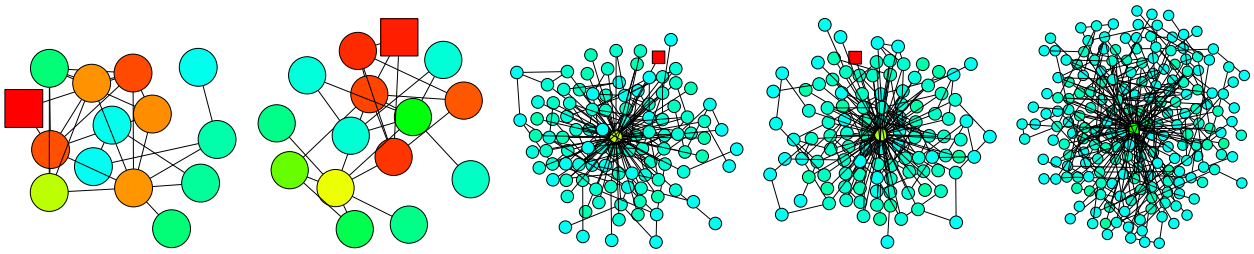
Figure 4 illustrates the main idea: as in common neighborhood aggregation networks, each layer increases the size of the influence distribution by aggregating neighborhoods from the previous layer. At the last layer, for each node, we carefully select from all of those itermediate representations (which "jump" to the last layer), potentially combining a few. If this is done independently for each node, then the model can adapt the effective neighborhood size for each node as needed, resulting in exactly the desired adaptivity.

Our model permits general layer-aggregation mechanisms. We explore three approaches; others are possible too. Let $h_v^{(1)}, ..., h_v^{(k)}$ be the jumping representations of node v (from k layers) that are to be aggregated.

Concatenation. A concatenation $\left[h_v^{(1)}, ..., h_v^{(k)}\right]$ is the most straightforward way to combine the layers, after which we may perform a linear transformation. If the transformation weights are shared across graph nodes, this approach is not node-adaptive. Instead, it optimizes the weights to combine the subgraph features in a way that works best for the dataset overall. One may expect concatenation to be suitable for small graphs and graphs with regular structure that require less adaptivity; also because weight-sharing helps reduce overfitting.

Max-pooling. An element-wise $\max\left(h_v^{(1)}, ..., h_v^{(k)}\right)$ selects the most informative layer *for each feature coordinate*. For example, feature coordinates that represent more local properties can use the feature coordinates learned from the close neighbors and those representing global status would favor features from the higher-up layers. Max-pooling is adaptive and has the advantage that it does not introduce any additional parameters to learn.

LSTM-attention. An attention mechanism identifies the most useful neighborhood ranges for each node v by computing an attention score $s_v^{(l)}$ for each layer l $\left(\sum_l s_v^{(l)} = 1\right)$, which represents the importance of the feature learned on the l-th layer for node v. The aggregated representation for node v is a weighted average of the layer features $\sum_l s_v^{(l)} \cdot h_v^{(l)}$. For LSTM attention, we input $h_v^{(1)}, ..., h_v^{(k)}$ into a bi-directional LSTM (Hochreiter & Schmidhuber, 1997) and generate the forward-LSTM and backward-LSTM hidden features $f_v^{(l)}$ and $b_v^{(l)}$ for each layer l. A linear mapping of the concatenated features $[f_v^{(l)}||b_v^{(l)}]$ yields the scalar importance score $s_v^{(l)}$. A Softmax layer applied to $\{s_v^{(l)}\}_{l=1}^k$



(a) tree-like (b) tree-like (c) affiliate to the hub (d) affiliate to the hub (e) hub

Figure 5. A 6-layer JK-Net learns to adapt to different subgraph structures

yields the attention of node v on its neighborhood in different ranges. Finally we take the sum of $[f_v^{(l)}||b_v^{(l)}]$ weighted by $\texttt{SoftMax}(\{s_v^{(l)}\}_{l=1}^k)$ to get the final layer representation. Another possible implementation combines LSTM with max-pooling. LSTM-attention is node adaptive because the attention scores are different for each node. We shall see that the this approach shines on large complex graphs, although it may overfit on small graphs (fewer training nodes) due to its relatively higher complexity.

4.1. JK-Net Learns to Adapt

The key idea for the design of layer-aggregation functions is to determine the importance of a node's subgraph features at different ranges after looking at the learned features on all layers, rather than to optimize and fix the same weights for all nodes. Under the same assumption on the ReLU activation distribution as in Theorem 1, we show below that layer-wise max-pooling implicitly learns the influence locality adaptively for different nodes. The proof for layer-wise attention follows similarly.

Proposition 1. *Assume that paths of the same length in the computation graph are activated with the same probability. The influence score $I(x, y)$ for any $x, y \in V$ under a k-layer JK-Net with layer-wise max-pooling is equivalent in expectation to a mixture of $0, .., k$-step random walk distributions on $\widetilde{G}$ at y starting at x, the coefficients of which depend on the values of the layer features $h_x^{(l)}$.*

We prove Proposition 1 in the appendix. Contrasting this result with the influence distributions of other aggregation mechanisms, we see that JK-networks indeed differ in their node-wise adaptivity of neighborhood ranges.

Figure 5 illustrates how a 6-layer JK-Net with max-pooling aggregation learns to adapt to different subgraph structures on a citation network. Within a tree-like structure, the influence stays in the "small community" the node belongs to. In contrast, 6-layer models whose influence distributions follow random walks, e.g. GCNs, would reach out too far into irrelevant parts of the graph, and models with few layers may not be able to cover the entire "community", as illustrated in Figure 1, and Figures 7, 8 in the appendix. For a node affiliated to a "hub", which presumably plays the role of connecting different types of nodes, JK-Net learns to put most influence on the node itself and otherwise spreads out the influence. GCNs, however, would not capture the importance of the node's own features in such a structure because the probability at an affiliate node is small after a few random walk steps. For hubs, JK-Net spreads out the influence across the neighboring nodes in a reasonable range, which makes sense because the nodes connected to the hubs are presumably as informative as the hubs' own features. For comparison, Table 6 in the appendix includes more visualizations of how models with random walk priors behave.

4.2. Intermediate Layer Aggregation and Structures

Looking at Figure 4, one may wonder whether the same inter-layer connections could be drawn between all layers. The resulting architecture is approximately a graph correspondent of DenseNets, which were introduced for computer vision problems (Huang et al., 2017), if the layer-wise concatenation aggregation is applied. This version, however, would require many more features to learn. Viewing the DenseNet setting (images) from a graph-theoretic perspective, images correspond to regular, in fact, near-planar graphs. Such graphs are far from being expanders, and do not pose the challenges of graphs with varying subgraph structures. Indeed, as we shall see, models with concatenation aggregation perform well on graphs with more regular structures such as images and well-structured communities. As a more general framework, JK-Net admits general layer-wise aggregation models and enables better structure-aware representations on graphs with complex structures.

5. Other Related Work

Spectral graph convolutional neural networks apply convolution on graphs by using the graph Laplacian eigenvectors as the Fourier atoms (Bruna et al., 2014; Shuman et al., 2013; Defferrard et al., 2016). A major drawback of the spectral methods, compared to spatial approaches like neighborhood-aggregation, is that the graph Laplacian needs to be known in advance. Hence, they cannot generalize to unseen graphs.

Dataset	Nodes	Edges	Classes	Features
Citeseer	3,327	4,732	6	3,703
Cora	2,708	5,429	7	1,433
Reddit	232,965	avg deg 492	50	300
PPI	56,944	818,716	121	50

Table 1. Dataset statistics

Model	Citeseer	Model	Cora
GCN (2)	77.3 (1.3)	GCN (2)	88.2 (0.7)
GAT (2)	76.2 (0.8)	GAT (3)	87.7 (0.3)
JK-MaxPool (1)	77.7 (0.5)	JK-Maxpool (6)	**89.6** (0.5)
JK-Concat (1)	**78.3** (0.8)	JK-Concat (6)	89.1 (1.1)
JK-LSTM (2)	74.7 (0.9)	JK-LSTM (1)	85.8 (1.0)

Table 2. Results of GCN-based JK-Nets on Citeseer and Cora. The baselines are GCN and GAT. The number in parentheses next to the model name indicates the best-performing number of layers among 1 to 6. Accuracy and standard deviation are computed from 3 random data splits.

6. Experiments

We evaluate JK-Nets on four benchmark datasets. (I) The task on citation networks (Citeseer, Cora) (Sen et al., 2008) is to classify academic papers into different subjects. The dataset contains bag-of-words features for each document (node) and citation links (edges) between documents. (II) On Reddit (Hamilton et al., 2017), the task is to predict the community to which different Reddit posts belong. Reddit is an online discussion forum where users comment in different topical communities. Two posts (nodes) are connected if some user commented on both posts. The dataset contains word vectors as node features. (III) For protein-protein interaction networks (PPI) (Hamilton et al., 2017), the task is to classify protein functions. PPI consists of 24 graphs, each corresponds to a human tissue. Each node has positional gene sets, motif gene sets and immunological signatures as features and gene ontology sets as labels. 20 graphs are used for training, 2 graphs are used for validation and the rest for testing. Statistics of the datasets are summarized in Table 1.

Settings. In the *transductive setting*, we are only allowed to access a subset of nodes in one graph as training data, and validate/test on others. Our experiments on Citeseer, Cora and Reddit are transductive. In the *inductive setting*, we use a number of full graphs as training data and use other completely unseen graphs as validation/testing data. Our experiments on PPI are inductive.

We compare against three baselines: Graph Convolutional Networks (GCN) (Kipf & Welling, 2017), GraphSAGE (Hamilton et al., 2017) and Graph Attention Networks (GAT) (Veličković et al., 2018).

6.1. Citeseer & Cora

For experiments on Citeseer and Cora, we choose GCN as the base model since on our data split, it is outperforming GAT. We construct JK-Nets by choosing MaxPooling (JK-MaxPool), Concatenation (JK-Concat), or LSTM-attention (JK-LSTM) as final aggregation layer. When taking the final aggregation, besides normal graph convolutional layers, we also take the first linear-transformed representation into account. The final prediction is done via a fully connected layer on top of the final aggregated representation. We split nodes in each graph into 60%, 20% and 20% for training, validation and testing. We vary the number of layers from 1 to 6 for each model and choose the best performing model with respect to the validation set. Throughout the experiments, we use the Adam optimizer (Kingma & Ba, 2014) with learning rate 0.005. We fix the dropout rate to be 0.5, the dimension of hidden features to be within $\{16, 32\}$, and add an $L2$ regularization of 0.0005 on model parameters. The results are shown in Table 2.

Results. We observe in Table 2 that JK-Nets outperform both GCN and GAT baselines in terms of prediction accuracy. Though JK-Nets perform well in general, there is no consistent winner and performance varies slightly across datasets.

Taking a closer look at results on Cora, both GCN and GAT achieve their best accuracies with only 2 or 3 layers, suggesting that local information is a stronger signal for classification than global ones. However, the fact that JK-Nets achieve the best performance with 6 layers indicates that global together with local information will help boost performance. This is where models like JK-Nets can be particularly beneficial. LSTM-attention may not be suitable for such small graphs because of its relatively high complexity.

6.2. Reddit

The Reddit data is too large to be handled well by current implementations of GCN or GAT. Hence, we use the more scalable GraphSAGE as the base model for JK-Net. It has skip connections and different modes of node aggregation. We experiment with Mean and MaxPool node aggregators, which take mean and max-pooling of a *linear transformation* of representations of the sampled neighbors. Combining each of GraphSAGE modes with MaxPooling, Concatenation or LSTM-attention as the last aggregation layer gives 6 JK-Net variants. We follow exactly the same setting of GraphSAGE as in the original paper (Hamilton et al., 2017), where the model consists of 2 hidden layers, each with 128 hidden units and is trained with Adam with learning rate of 0.01 and no weight decay. Results are shown in Table 3.

Results. With MaxPool as node aggregator and Concat as layer aggregator, JK-Net achieves the best Micro-F1 score

Node \ JK	GraphSAGE	Maxpool	Concat	LSTM
Mean	0.950	0.953	0.955	0.950
MaxPool	0.948	0.924	**0.965**	0.877

Table 3. Results of GraphSAGE-based JK-Nets on Reddit. The baseline is GraphSAGE. Model performance is measured in Micro-F1 score. Each column shows the results of a JK-Net variant. For all models, the number of layers is fixed to 2.

among GarphSAGE and JK-Net variants. Note that the original GraphSAGE already performs fairly well with a Micro-F1 of 0.95. JK-Net reduces the error by 30%. The communities in the Reddit dataset were explicitly chosen from the well-behaved middle-sized communities to avoid the noisy cores and tree-like small communities (Hamilton et al., 2017). As a result, this graph is more regular than the original Reddit data, and hence not exhibit the problems of varying subgraph structures. In such a case, the added flexibility of the node-specific neighborhood choices may not be as relevant, and the stabilizing properties of concatenation instead come into play.

6.3. PPI

We demonstrate the power of adaptive JK-Nets, e.g., JK-LSTM, with experiments on the PPI data, where the subgraphs have more diverse and complex structures than those in the Reddit community detection dataset. We use both GraphSAGE and GAT as base models for JK-Net. The implementation of GraphSAGE and GAT are quite different: GraphSAGE is sample-based, where neighbors of a node are sampled to be a fixed number, while GAT considers all neighbors. Such differences cause large gaps in terms of both scalability and performances. Given that GraphSAGE scales to much larger graphs, it appears particularly valuable to evaluate how much JK-Net can improve upon GraphSAGE.

For GraphSAGE we follow the setup as in the Reddit experiments, except that we use 3 layers when possible, and compare the performance after 10 and 30 epochs of training. The results are shown in Table 4. For GAT and its JK-Net variants we stack two hidden layers with 4 attention heads computing 256 features (for a total of 1024 features), and a final prediction layer with 6 attention heads computing 121 features each. They are further averaged and input into sigmoid activations. We employ skip connections across intermediate attentional layers. These models are trained with Batch-size 2 and Adam optimizer with learning rate of 0.005. The results are shown in Table 5.

Results. JK-Nets with the LSTM-attention aggregators outperform the non-adaptive models GraphSAGE, GAT and JK-Nets with concatenation aggregators. In particular, JK-LSTM outperforms GraphSAGE by 0.128 in terms of micro-F1 score after 30 epochs of training. Structure-aware node adaptive models are especially beneficial on such complex graphs with diverse structures.

Node \ JK	SAGE	MaxPool	Concat	LSTM
Mean (10 epochs)	0.644	0.658	0.667	**0.721**
Mean (30 epochs)	0.690	0.713	0.694	**0.818**
MaxPool (10 epochs)	0.668	0.671	0.687	0.621*

Table 4. Results of GraphSAGE-based JK-Net on the PPI data. The baseline is GraphSAGE (SAGE). Each column, excluding SAGE, represents a JK-Net with different layer aggregation. All models use 3 layers, except for those with "*", whose number of layers is set to 2 due to GPU memory constraints. 0.6 is the corresponding 2-layer GraphSAGE performance.

Model	PPI
MLP	0.422
GAT	0.968 (0.002)
JK-Concat (2)	0.959 (0.003)
JK-LSTM (3)	0.969 (0.006)
JK-Dense-Concat (2)*	0.956 (0.004)
JK-Dense-LSTM (2)*	**0.976** (0.007)

Table 5. Micro-F1 scores of GAT-based JK-Nets on the PPI data. The baselines are GAT and MLP (Multilayer Perceptron). While the number of layers for JK-Concat and JK-LSTM are chosen from $\{2, 3\}$, the ones for JK-Dense-Concat and JK-Dense-LSTM are directly set to 2 due to GPU memory constraints.

7. Conclusion

Motivated by observations that reveal great differences in neighborhood information ranges for graph node embeddings, we propose a new aggregation scheme for node representation learning that can adapt neigborhood ranges to nodes individually. This JK-network can improve representations in particular for graphs that have subgraphs of diverse local structure, and may hence not be well captured by fixed numbers of neighborhood aggregations. Interesting directions for future work include exploring other layer aggregators and studying the effect of the combination of various layer-wise and node-wise aggregators on different types of graph structures.

Acknowledgements

This research was supported by NSF CAREER award 1553284, and JST ERATO Kawarabayashi Large Graph Project, Grant Number JPMJER1201, Japan.

References

Bruna, J., Zaremba, W., Szlam, A., and LeCun, Y. Spectral networks and locally connected networks on graphs.

International Conference on Learning Representations (ICLR), 2014.

Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. On the properties of neural machine translation: Encoder-decoder approaches. In *Workshop on Syntax, Semantics and Structure in Statistical Translation*, pp. 103–111, 2014.

Choromanska, A., LeCun, Y., and Arous, G. B. Open problem: The landscape of the loss surfaces of multilayer networks. In *Conference on Learning Theory (COLT)*, pp. 1756–1760, 2015.

Defferrard, M., Bresson, X., and Vandergheynst, P. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 3844–3852, 2016.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *International Conference on Machine Learning (ICML)*, pp. 1273–1272, 2017.

Grover, A. and Leskovec, J. node2vec: Scalable feature learning for networks. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 855–864, 2016.

Hamilton, W. L., Ying, R., and Leskovec, J. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 1025–1035, 2017.

Hammond, D. K., Vandergheynst, P., and Gribonval, R. Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis*, 30(2):129–150, 2011.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016a.

He, K., Zhang, X., Ren, S., and Sun, J. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pp. 630–645, 2016b.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Hoory, S., Linial, N., and Wigderson, A. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, 2006.

Hoshen, Y. Vain: Attentional multi-agent predictive modeling. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 2698–2708, 2017.

Huang, G., Liu, Z., Weinberger, K. Q., and van der Maaten, L. Densely connected convolutional networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2261–2269, 2017.

Kawaguchi, K. Deep learning without poor local minima. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 586–594, 2016.

Kearnes, S., McCloskey, K., Berndl, M., Pande, V., and Riley, P. Molecular graph convolutions: moving beyond fingerprints. *Journal of computer-aided molecular design*, 30(8):595–608, 2016.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations (ICLR)*, 2017.

Koh, P. W. and Liang, P. Understanding black-box predictions via influence functions. In *International Conference on Machine Learning (ICML)*, pp. 1885–1894, 2017.

Leskovec, J. and Mcauley, J. J. Learning to discover social circles in ego networks. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 539–547, 2012.

Leskovec, J., Lang, K. J., Dasgupta, A., and Mahoney, M. W. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. Gated graph sequence neural networks. *International Conference on Learning Representations (ICLR)*, 2016.

Lovász, L. Random walks on graphs. *Combinatorics, Paul erdos is eighty*, 2:1–46, 1993.

Maehara, T., Akiba, T., Iwata, Y., and Kawarabayashi, K.-i. Computing personalized pagerank quickly by exploiting graph structures. *Proceedings of the VLDB Endowment*, 7(12):1023–1034, 2014.

Monti, F., Boscaini, D., Masci, J., Rodolà, E., Svoboda, J., and Bronstein, M. M. Geometric deep learning on graphs and manifolds using mixture model cnns. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5425–5434, 2017.

Perozzi, B., Al-Rfou, R., and Skiena, S. Deepwalk: Online learning of social representations. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 701–710, 2014.

Pham, T., Tran, T., Phung, D. Q., and Venkatesh, S. Column networks for collective classification. In *AAAI Conference on Artificial Intelligence*, pp. 2485–2491, 2017.

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.

Shervashidze, N., Schweitzer, P., Leeuwen, E. J. v., Mehlhorn, K., and Borgwardt, K. M. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep):2539–2561, 2011.

Shuman, D. I., Narang, S. K., Frossard, P., Ortega, A., and Vandergheynst, P. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE Signal Processing Magazine*, 30(3):83–98, 2013.

Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., and Mei, Q. Line: Large-scale information network embedding. In *Proceedings of the International World Wide Web Conference (WWW)*, pp. 1067–1077, 2015.

Tsonis, A. A., Swanson, K. L., and Roebber, P. J. What do networks have to do with climate? *Bulletin of the American Meteorological Society*, 87(5):585–595, 2006.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph attention networks. *International Conference on Learning Representations (ICLR)*, 2018.

Weisfeiler, B. and Lehman, A. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9): 12–16, 1968.

GRAPH ATTENTION NETWORKS

Petar Veličković*
Department of Computer Science and Technology
University of Cambridge
petar.velickovic@cst.cam.ac.uk

Guillem Cucurull*
Centre de Visió per Computador, UAB
gcucurull@gmail.com

Arantxa Casanova*
Centre de Visió per Computador, UAB
ar.casanova.8@gmail.com

Adriana Romero
Montréal Institute for Learning Algorithms
Facebook AI Research
adrianars@fb.com

Pietro Liò
Department of Computer Science and Technology
University of Cambridge
pietro.lio@cst.cam.ac.uk

Yoshua Bengio
Montréal Institute for Learning Algorithms
yoshua.umontreal@gmail.com

ABSTRACT

We present graph attention networks (GATs), novel neural network architectures that operate on graph-structured data, leveraging masked self-attentional layers to address the shortcomings of prior methods based on graph convolutions or their approximations. By stacking layers in which nodes are able to attend over their neighborhoods' features, we enable (implicitly) specifying different weights to different nodes in a neighborhood, without requiring any kind of computationally intensive matrix operation (such as inversion) or depending on knowing the graph structure upfront. In this way, we address several key challenges of spectral-based graph neural networks simultaneously, and make our model readily applicable to inductive as well as transductive problems. Our GAT models have achieved or matched state-of-the-art results across four established transductive and inductive graph benchmarks: the *Cora*, *Citeseer* and *Pubmed* citation network datasets, as well as a *protein-protein interaction* dataset (wherein test graphs remain unseen during training).

1 INTRODUCTION

Convolutional Neural Networks (CNNs) have been successfully applied to tackle problems such as image classification (He et al., 2016), semantic segmentation (Jégou et al., 2017) or machine translation (Gehring et al., 2016), where the underlying data representation has a grid-like structure. These architectures efficiently reuse their local filters, with learnable parameters, by applying them to all the input positions.

However, many interesting tasks involve data that can not be represented in a grid-like structure and that instead lies in an irregular domain. This is the case of 3D meshes, social networks, telecommunication networks, biological networks or brain connectomes. Such data can usually be represented in the form of graphs.

There have been several attempts in the literature to extend neural networks to deal with arbitrarily structured graphs. Early work used recursive neural networks to process data represented in graph domains as directed acyclic graphs (Frasconi et al., 1998; Sperduti & Starita, 1997). Graph Neural Networks (GNNs) were introduced in Gori et al. (2005) and Scarselli et al. (2009) as a generalization of recursive neural networks that can directly deal with a more general class of graphs, e.g. cyclic, directed and undirected graphs. GNNs consist of an iterative process, which propagates the node

*Work performed while the author was at the Montréal Institute of Learning Algorithms.

states until equilibrium; followed by a neural network, which produces an output for each node based on its state. This idea was adopted and improved by Li et al. (2016), which propose to use gated recurrent units (Cho et al., 2014) in the propagation step.

Nevertheless, there is an increasing interest in generalizing convolutions to the graph domain. Advances in this direction are often categorized as spectral approaches and non-spectral approaches.

On one hand, spectral approaches work with a spectral representation of the graphs and have been successfully applied in the context of node classification. In Bruna et al. (2014), the convolution operation is defined in the Fourier domain by computing the eigendecomposition of the graph Laplacian, resulting in potentially intense computations and non-spatially localized filters. These issues were addressed by subsequent works. Henaff et al. (2015) introduced a parameterization of the spectral filters with smooth coefficients in order to make them spatially localized. Later, Defferrard et al. (2016) proposed to approximate the filters by means of a Chebyshev expansion of the graph Laplacian, removing the need to compute the eigenvectors of the Laplacian and yielding spatially localized filters. Finally, Kipf & Welling (2017) simplified the previous method by restricting the filters to operate in a 1-step neighborhood around each node. However, in all of the aforementioned spectral approaches, the learned filters depend on the Laplacian eigenbasis, which depends on the graph structure. Thus, a model trained on a specific structure can not be directly applied to a graph with a different structure.

On the other hand, we have non-spectral approaches (Duvenaud et al., 2015; Atwood & Towsley, 2016; Hamilton et al., 2017), which define convolutions directly on the graph, operating on groups of spatially close neighbors. One of the challenges of these approaches is to define an operator which works with different sized neighborhoods and maintains the weight sharing property of CNNs. In some cases, this requires learning a specific weight matrix for each node degree (Duvenaud et al., 2015), using the powers of a transition matrix to define the neighborhood while learning weights for each input channel and neighborhood degree (Atwood & Towsley, 2016), or extracting and normalizing neighborhoods containing a fixed number of nodes (Niepert et al., 2016). Monti et al. (2016) presented mixture model CNNs (MoNet), a spatial approach which provides a unified generalization of CNN architectures to graphs. More recently, Hamilton et al. (2017) introduced GraphSAGE, a method for computing node representations in an *inductive* manner. This technique operates by sampling a fixed-size neighborhood of each node, and then performing a specific aggregator over it (such as the mean over all the sampled neighbors' feature vectors, or the result of feeding them through a recurrent neural network). This approach has yielded impressive performance across several large-scale inductive benchmarks.

Attention mechanisms have become almost a *de facto* standard in many sequence-based tasks (Bahdanau et al., 2015; Gehring et al., 2016). One of the benefits of attention mechanisms is that they allow for dealing with variable sized inputs, focusing on the most relevant parts of the input to make decisions. When an attention mechanism is used to compute a representation of a single sequence, it is commonly referred to as *self-attention* or *intra-attention*. Together with Recurrent Neural Networks (RNNs) or convolutions, self-attention has proven to be useful for tasks such as machine reading (Cheng et al., 2016) and learning sentence representations (Lin et al., 2017). However, Vaswani et al. (2017) showed that not only *self-attention* can improve a method based on RNNs or convolutions, but also that it is sufficient for constructing a powerful model obtaining *state-of-the-art* performance on the machine translation task.

Inspired by this recent work, we introduce an attention-based architecture to perform node classification of graph-structured data. The idea is to compute the hidden representations of each node in the graph, by attending over its neighbors, following a *self-attention* strategy. The attention architecture has several interesting properties: (1) the operation is efficient, since it is parallelizable across node-neighbor pairs; (2) it can be applied to graph nodes having different degrees by specifying arbitrary weights to the neighbors; and (3) the model is directly applicable to *inductive* learning problems, including tasks where the model has to generalize to completely unseen graphs. We validate the proposed approach on four challenging benchmarks: *Cora*, *Citeseer* and *Pubmed* citation networks as well as an inductive *protein-protein interaction* dataset, achieving or matching state-of-the-art results that highlight the potential of attention-based models when dealing with arbitrarily structured graphs.

It is worth noting that, as Kipf & Welling (2017) and Atwood & Towsley (2016), our work can also be reformulated as a particular instance of MoNet (Monti et al., 2016). Moreover, our approach of sharing a neural network computation across edges is reminiscent of the formulation of relational networks (Santoro et al., 2017) and VAIN (Hoshen, 2017), wherein relations between objects or agents are aggregated pair-wise, by employing a shared mechanism. Similarly, our proposed attention model can be connected to the works by Duan et al. (2017) and Denil et al. (2017), which use a neighborhood attention operation to compute attention coefficients between different objects in an environment. Other related approaches include locally linear embedding (LLE) (Roweis & Saul, 2000) and memory networks (Weston et al., 2014). LLE selects a fixed number of neighbors around each data point, and learns a weight coefficient for each neighbor to reconstruct each point as a weighted sum of its neighbors. A second optimization step extracts the point's feature embedding. Memory networks also share some connections with our work, in particular, if we interpret the neighborhood of a node as the memory, which is used to compute the node features by attending over its values, and then is updated by storing the new features in the same position.

2 GAT ARCHITECTURE

In this section, we will present the building block layer used to construct arbitrary graph attention networks (through stacking this layer), and directly outline its theoretical and practical benefits and limitations compared to prior work in the domain of neural graph processing.

2.1 GRAPH ATTENTIONAL LAYER

We will start by describing a single *graph attentional layer*, as the sole layer utilized throughout all of the GAT architectures used in our experiments. The particular attentional setup utilized by us closely follows the work of Bahdanau et al. (2015)—but the framework is agnostic to the particular choice of attention mechanism.

The input to our layer is a set of node features, $\mathbf{h} = \{\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_N\}, \vec{h}_i \in \mathbb{R}^F$, where N is the number of nodes, and F is the number of features in each node. The layer produces a new set of node features (of potentially different cardinality F'), $\mathbf{h}' = \{\vec{h}'_1, \vec{h}'_2, \ldots, \vec{h}'_N\}, \vec{h}'_i \in \mathbb{R}^{F'}$, as its output.

In order to obtain sufficient expressive power to transform the input features into higher-level features, at least one learnable linear transformation is required. To that end, as an initial step, a shared linear transformation, parametrized by a *weight matrix*, $\mathbf{W} \in \mathbb{R}^{F' \times F}$, is applied to every node. We then perform *self-attention* on the nodes—a shared attentional mechanism $a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \to \mathbb{R}$ computes *attention coefficients*

$$e_{ij} = a(\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j) \tag{1}$$

that indicate the *importance* of node j's features to node i. In its most general formulation, the model allows every node to attend on every other node, *dropping all structural information*. We inject the graph structure into the mechanism by performing *masked attention*—we only compute e_{ij} for nodes $j \in \mathcal{N}_i$, where $\mathcal{N}_i$ is some *neighborhood* of node i in the graph. In all our experiments, these will be exactly the first-order neighbors of i (including i). To make coefficients easily comparable across different nodes, we normalize them across all choices of j using the softmax function:

$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})}. \tag{2}$$

In our experiments, the attention mechanism a is a single-layer feedforward neural network, parametrized by a weight vector $\vec{\mathbf{a}} \in \mathbb{R}^{2F'}$, and applying the LeakyReLU nonlinearity (with negative input slope $\alpha = 0.2$). Fully expanded out, the coefficients computed by the attention mechanism (illustrated by Figure 1 (left)) may then be expressed as:

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_j]\right)\right)}{\sum_{k \in \mathcal{N}_i} \exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_k]\right)\right)} \tag{3}$$

where $\cdot^T$ represents transposition and $\|$ is the concatenation operation.

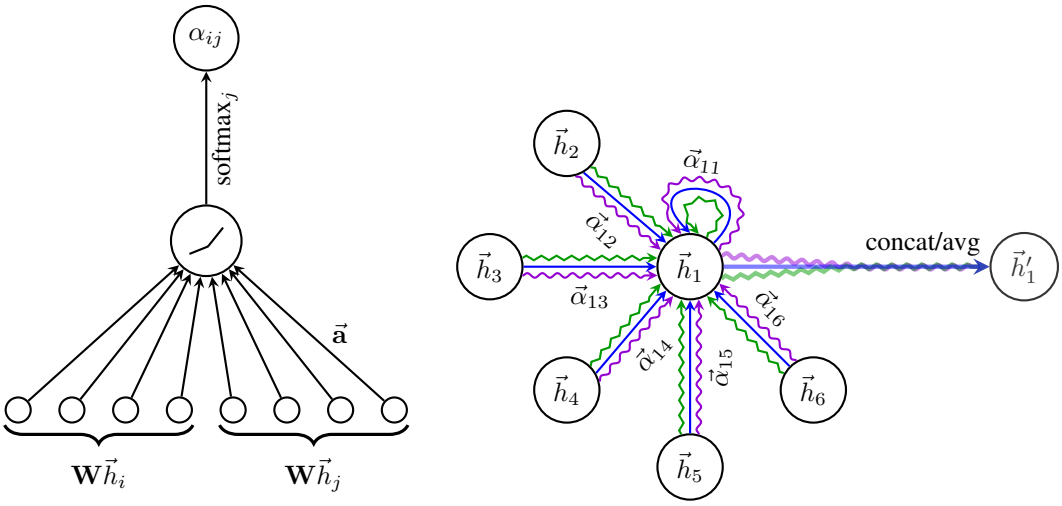


Figure 1: **Left:** The attention mechanism $a(\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j)$ employed by our model, parametrized by a weight vector $\vec{\mathbf{a}} \in \mathbb{R}^{2F'}$, applying a LeakyReLU activation. **Right:** An illustration of multi-head attention (with $K = 3$ heads) by node 1 on its neighborhood. Different arrow styles and colors denote independent attention computations. The aggregated features from each head are concatenated or averaged to obtain $\vec{h}'_1$.

Once obtained, the normalized attention coefficients are used to compute a linear combination of the features corresponding to them, to serve as the final output features for every node (after potentially applying a nonlinearity, σ):

$$\vec{h}'_i = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W}\vec{h}_j\right). \tag{4}$$

To stabilize the learning process of self-attention, we have found extending our mechanism to employ *multi-head attention* to be beneficial, similarly to Vaswani et al. (2017). Specifically, K independent attention mechanisms execute the transformation of Equation 4, and then their features are concatenated, resulting in the following output feature representation:

$$\vec{h}'_i = \Big\Vert_{k=1}^{K} \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k \vec{h}_j\right) \tag{5}$$

where $\|$ represents concatenation, α_{ij}^k are normalized attention coefficients computed by the k-th attention mechanism (a^k), and $\mathbf{W}^k$ is the corresponding input linear transformation's weight matrix. Note that, in this setting, the final returned output, $\mathbf{h}'$, will consist of KF' features (rather than F') for each node.

Specially, if we perform multi-head attention on the final (prediction) layer of the network, concatenation is no longer sensible—instead, we employ *averaging*, and delay applying the final nonlinearity (usually a softmax or logistic sigmoid for classification problems) until then:

$$\vec{h}'_i = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k \vec{h}_j\right) \tag{6}$$

The aggregation process of a multi-head graph attentional layer is illustrated by Figure 1 (right).

2.2 Comparisons to related work

The graph attentional layer described in subsection 2.1 directly addresses several issues that were present in prior approaches to modelling graph-structured data with neural networks:

- Computationally, it is highly efficient: the operation of the self-attentional layer can be parallelized across all edges, and the computation of output features can be parallelized across all nodes. No eigendecompositions or similar computationally intensive matrix operations are required. The time complexity of a single GAT attention head computing F' features may be expressed as $O(|V|FF' + |E|F')$, where F is the number of input features, and $|V|$ and $|E|$ are the numbers of nodes and edges in the graph, respectively. This complexity is on par with the baseline methods such as Graph Convolutional Networks (GCNs) (Kipf & Welling, 2017). Applying multi-head attention multiplies the storage and parameter requirements by a factor of K, while the individual heads' computations are fully independent and can be parallelized.

- As opposed to GCNs, our model allows for (implicitly) assigning *different importances* to nodes of a same neighborhood, enabling a leap in model capacity. Furthermore, analyzing the learned attentional weights may lead to benefits in interpretability, as was the case in the machine translation domain (e.g. the qualitative analysis of Bahdanau et al. (2015)).

- The attention mechanism is applied in a shared manner to all edges in the graph, and therefore it does not depend on upfront access to the global graph structure or (features of) all of its nodes (a limitation of many prior techniques). This has several desirable implications:

  - The graph is not required to be undirected (we may simply leave out computing α_{ij} if edge $j \to i$ is not present).
  - It makes our technique directly applicable to *inductive* learning—including tasks where the model is evaluated on graphs that are *completely unseen* during training.

- The recently published inductive method of Hamilton et al. (2017) samples a *fixed-size neighborhood* of each node, in order to keep its computational footprint consistent; this does not allow it access to the entirety of the neighborhood while performing inference. Moreover, this technique achieved some of its strongest results when an LSTM (Hochreiter & Schmidhuber, 1997)-based neighborhood aggregator is used. This assumes the existence of a consistent sequential node ordering across neighborhoods, and the authors have rectified it by consistently feeding randomly-ordered sequences to the LSTM. Our technique does not suffer from either of these issues—it works with the entirety of the neighborhood (at the expense of a variable computational footprint, which is still on-par with methods like the GCN), and does not assume any ordering within it.

- As mentioned in Section 1, GAT can be reformulated as a particular instance of MoNet (Monti et al., 2016). More specifically, setting the pseudo-coordinate function to be $u(x, y) = f(x) \| f(y)$, where $f(x)$ represent (potentially MLP-transformed) features of node x and $\|$ is concatenation; and the weight function to be $w_j(u) = \mathrm{softmax}(\mathrm{MLP}(u))$ (with the softmax performed over the entire neighborhood of a node) would make MoNet's patch operator similar to ours. Nevertheless, one should note that, in comparison to previously considered MoNet instances, our model uses node features for similarity computations, rather than the node's structural properties (which would assume knowing the graph structure upfront).

We were able to produce a version of the GAT layer that leverages *sparse* matrix operations, reducing the storage complexity to linear in the number of nodes and edges and enabling the execution of GAT models on larger graph datasets. However, the tensor manipulation framework we used only supports sparse matrix multiplication for rank-2 tensors, which limits the batching capabilities of the layer as it is currently implemented (especially for datasets with multiple graphs). Appropriately addressing this constraint is an important direction for future work. Depending on the regularity of the graph structure in place, GPUs may not be able to offer major performance benefits compared to CPUs in these sparse scenarios. It should also be noted that the size of the "receptive field" of our model is upper-bounded by the depth of the network (similarly as for GCN and similar models). Techniques such as skip connections (He et al., 2016) could be readily applied for appropriately extending the depth, however. Lastly, parallelization across all the graph edges, especially in a distributed manner, may involve a lot of redundant computation, as the neighborhoods will often highly overlap in graphs of interest.

Table 1: Summary of the datasets used in our experiments.

	Cora	**Citeseer**	**Pubmed**	**PPI**
Task	Transductive	Transductive	Transductive	Inductive
# Nodes	2708 (1 graph)	3327 (1 graph)	19717 (1 graph)	56944 (24 graphs)
# Edges	5429	4732	44338	818716
# Features/Node	1433	3703	500	50
# Classes	7	6	3	121 (multilabel)
# Training Nodes	140	120	60	44906 (20 graphs)
# Validation Nodes	500	500	500	6514 (2 graphs)
# Test Nodes	1000	1000	1000	5524 (2 graphs)

3 EVALUATION

We have performed comparative evaluation of GAT models against a wide variety of strong baselines and previous approaches, on four established graph-based benchmark tasks (transductive as well as inductive), achieving or matching state-of-the-art performance across all of them. This section summarizes our experimental setup, results, and a brief qualitative analysis of a GAT model's extracted feature representations.

3.1 DATASETS

Transductive learning We utilize three standard citation network benchmark datasets—Cora, Citeseer and Pubmed (Sen et al., 2008)—and closely follow the transductive experimental setup of Yang et al. (2016). In all of these datasets, nodes correspond to documents and edges to (undirected) citations. Node features correspond to elements of a bag-of-words representation of a document. Each node has a class label. We allow for only 20 nodes per class to be used for training—however, honoring the transductive setup, the training algorithm has access to all of the nodes' feature vectors. The predictive power of the trained models is evaluated on 1000 test nodes, and we use 500 additional nodes for validation purposes (the same ones as used by Kipf & Welling (2017)). The Cora dataset contains 2708 nodes, 5429 edges, 7 classes and 1433 features per node. The Citeseer dataset contains 3327 nodes, 4732 edges, 6 classes and 3703 features per node. The Pubmed dataset contains 19717 nodes, 44338 edges, 3 classes and 500 features per node.

Inductive learning We make use of a protein-protein interaction (PPI) dataset that consists of graphs corresponding to different human tissues (Zitnik & Leskovec, 2017). The dataset contains 20 graphs for training, 2 for validation and 2 for testing. Critically, testing graphs remain *completely unobserved* during training. To construct the graphs, we used the preprocessed data provided by Hamilton et al. (2017). The average number of nodes per graph is 2372. Each node has 50 features that are composed of positional gene sets, motif gene sets and immunological signatures. There are 121 labels for each node set from gene ontology, collected from the Molecular Signatures Database (Subramanian et al., 2005), and a node can possess several labels simultaneously.

An overview of the interesting characteristics of the datasets is given in Table 1.

3.2 STATE-OF-THE-ART METHODS

Transductive learning For transductive learning tasks, we compare against the same strong baselines and state-of-the-art approaches as specified in Kipf & Welling (2017). This includes label propagation (LP) (Zhu et al., 2003), semi-supervised embedding (SemiEmb) (Weston et al., 2012), manifold regularization (ManiReg) (Belkin et al., 2006), skip-gram based graph embeddings (DeepWalk) (Perozzi et al., 2014), the iterative classification algorithm (ICA) (Lu & Getoor, 2003) and Planetoid (Yang et al., 2016). We also directly compare our model against GCNs (Kipf & Welling, 2017), as well as graph convolutional models utilising higher-order Chebyshev filters (Defferrard et al., 2016), and the MoNet model presented in Monti et al. (2016).

Inductive learning For the inductive learning task, we compare against the four different supervised GraphSAGE inductive methods presented in Hamilton et al. (2017). These provide a variety of approaches to aggregating features within a sampled neighborhood: GraphSAGE-GCN (which extends a graph convolution-style operation to the inductive setting), GraphSAGE-mean (taking the elementwise mean value of feature vectors), GraphSAGE-LSTM (aggregating by feeding the neighborhood features into an LSTM) and GraphSAGE-pool (taking the elementwise maximization operation of feature vectors transformed by a shared nonlinear multilayer perceptron). The other transductive approaches are either completely inappropriate in an inductive setting or assume that nodes are incrementally added to a single graph, making them unusable for the setup where test graphs are completely unseen during training (such as the PPI dataset).

Additionally, for both tasks we provide the performance of a per-node shared multilayer perceptron (MLP) classifier (that does not incorporate graph structure at all).

3.3 Experimental setup

Transductive learning For the transductive learning tasks, we apply a two-layer GAT model. Its architectural hyperparameters have been optimized on the Cora dataset and are then reused for Citeseer. The first layer consists of $K = 8$ attention heads computing $F' = 8$ features each (for a total of 64 features), followed by an exponential linear unit (ELU) (Clevert et al., 2016) nonlinearity. The second layer is used for classification: a single attention head that computes C features (where C is the number of classes), followed by a softmax activation. For coping with the small training set sizes, regularization is liberally applied within the model. During training, we apply L_2 regularization with $\lambda = 0.0005$. Furthermore, dropout (Srivastava et al., 2014) with $p = 0.6$ is applied to both layers' inputs, as well as *to the normalized attention coefficients* (critically, this means that at each training iteration, each node is exposed to a stochastically sampled neighborhood). Similarly as observed by Monti et al. (2016), we found that Pubmed's training set size (60 examples) required slight changes to the GAT architecture: we have applied $K = 8$ output attention heads (instead of one), and strengthened the L_2 regularization to $\lambda = 0.001$. Otherwise, the architecture matches the one used for Cora and Citeseer.

Inductive learning For the inductive learning task, we apply a three-layer GAT model. Both of the first two layers consist of $K = 4$ attention heads computing $F' = 256$ features (for a total of 1024 features), followed by an ELU nonlinearity. The final layer is used for (multi-label) classification: $K = 6$ attention heads computing 121 features each, that are averaged and followed by a logistic sigmoid activation. The training sets for this task are sufficiently large and we found no need to apply L_2 regularization or dropout—we have, however, successfully employed skip connections (He et al., 2016) across the intermediate attentional layer. We utilize a batch size of 2 graphs during training. To strictly evaluate the benefits of applying an attention mechanism in this setting (i.e. comparing with a near GCN-equivalent model), we also provide the results when a *constant attention mechanism*, $a(x, y) = 1$, is used, with the same architecture—this will assign the same weight to every neighbor.

Both models are initialized using Glorot initialization (Glorot & Bengio, 2010) and trained to minimize cross-entropy on the training nodes using the Adam SGD optimizer (Kingma & Ba, 2014) with an initial learning rate of 0.01 for Pubmed, and 0.005 for all other datasets. In both cases we use an early stopping strategy on both the cross-entropy loss and accuracy (transductive) or micro-F_1 (inductive) score on the validation nodes, with a patience of 100 epochs[1].

3.4 Results

The results of our comparative evaluation experiments are summarized in Tables 2 and 3.

For the transductive tasks, we report the mean classification accuracy (with standard deviation) on the test nodes of our method after 100 runs, and reuse the metrics already reported in Kipf & Welling (2017) and Monti et al. (2016) for state-of-the-art techniques. Specifically, for the Chebyshev filter-based approach (Defferrard et al., 2016), we provide the maximum reported performance for filters of orders $K = 2$ and $K = 3$. In order to fairly assess the benefits of the attention mechanism, we further evaluate a GCN model that computes 64 hidden features, attempting both the ReLU and

[1] Our implementation of the GAT layer may be found at: `https://github.com/PetarV-/GAT`

Table 2: Summary of results in terms of classification accuracies, for Cora, Citeseer and Pubmed. GCN-64* corresponds to the best GCN result computing 64 hidden features (using ReLU or ELU).

Transductive

Method	**Cora**	**Citeseer**	**Pubmed**
MLP	55.1%	46.5%	71.4%
ManiReg (Belkin et al., 2006)	59.5%	60.1%	70.7%
SemiEmb (Weston et al., 2012)	59.0%	59.6%	71.7%
LP (Zhu et al., 2003)	68.0%	45.3%	63.0%
DeepWalk (Perozzi et al., 2014)	67.2%	43.2%	65.3%
ICA (Lu & Getoor, 2003)	75.1%	69.1%	73.9%
Planetoid (Yang et al., 2016)	75.7%	64.7%	77.2%
Chebyshev (Defferrard et al., 2016)	81.2%	69.8%	74.4%
GCN (Kipf & Welling, 2017)	81.5%	70.3%	**79.0%**
MoNet (Monti et al., 2016)	81.7 ± 0.5%	—	78.8 ± 0.3%
GCN-64*	81.4 ± 0.5%	70.9 ± 0.5%	**79.0** ± 0.3%
GAT (ours)	**83.0** ± 0.7%	**72.5** ± 0.7%	**79.0** ± 0.3%

Table 3: Summary of results in terms of micro-averaged F_1 scores, for the PPI dataset. GraphSAGE* corresponds to the best GraphSAGE result we were able to obtain by just modifying its architecture. Const-GAT corresponds to a model with the same architecture as GAT, but with a constant attention mechanism (assigning same importance to each neighbor; GCN-like inductive operator).

Inductive

Method	**PPI**
Random	0.396
MLP	0.422
GraphSAGE-GCN (Hamilton et al., 2017)	0.500
GraphSAGE-mean (Hamilton et al., 2017)	0.598
GraphSAGE-LSTM (Hamilton et al., 2017)	0.612
GraphSAGE-pool (Hamilton et al., 2017)	0.600
GraphSAGE*	0.768
Const-GAT (ours)	0.934 ± 0.006
GAT (ours)	**0.973** ± 0.002

ELU activation, and reporting (as GCN-64*) the better result after 100 runs (which was the ReLU in all three cases).

For the inductive task, we report the micro-averaged F_1 score on the nodes of the two unseen test graphs, averaged after 10 runs, and reuse the metrics already reported in Hamilton et al. (2017) for the other techniques. Specifically, as our setup is supervised, we compare against the supervised GraphSAGE approaches. To evaluate the benefits of aggregating across the entire neighborhood, we further provide (as GraphSAGE*) the best result we were able to achieve with GraphSAGE by just modifying its architecture (this was with a three-layer GraphSAGE-LSTM with [512, 512, 726] features computed in each layer and 128 features used for aggregating neighborhoods). Finally, we report the 10-run result of our constant attention GAT model (as Const-GAT), to fairly evaluate the benefits of the attention mechanism against a GCN-like aggregation scheme (with the same architecture).

Our results successfully demonstrate state-of-the-art performance being achieved or matched across all four datasets—in concordance with our expectations, as per the discussion in Section 2.2. More specifically, we are able to improve upon GCNs by a margin of 1.5% and 1.6% on Cora and Citeseer, respectively, suggesting that assigning different weights to nodes of a same neighborhood may be beneficial. It is worth noting the improvements achieved on the PPI dataset: Our GAT model

improves by 20.5% w.r.t. the best GraphSAGE result we were able to obtain, demonstrating that our model has the potential to be applied in inductive settings, and that larger predictive power can be leveraged by observing the entire neighborhood. Furthermore, it improves by 3.9% w.r.t. Const-GAT (the identical architecture with constant attention mechanism), once again directly demonstrating the significance of being able to assign different weights to different neighbors.

The effectiveness of the learned feature representations may also be investigated qualitatively—and for this purpose we provide a visualization of the t-SNE (Maaten & Hinton, 2008)-transformed feature representations extracted by the first layer of a GAT model pre-trained on the Cora dataset (Figure 2). The representation exhibits discernible clustering in the projected 2D space. Note that these clusters correspond to the seven labels of the dataset, verifying the model's discriminative power across the seven topic classes of Cora. Additionally, we visualize the relative strengths of the normalized attention coefficients (averaged across all eight attention heads). Properly interpreting these coefficients (as performed by e.g. Bahdanau et al. (2015)) will require further domain knowledge about the dataset under study, and is left for future work.

4 Conclusions

We have presented graph attention networks (GATs), novel convolution-style neural networks that operate on graph-structured data, leveraging masked self-attentional layers. The graph attentional layer utilized throughout these networks is computationally efficient (does not require computationally intensive matrix operations, and is parallelizable across all nodes in the graph), allows for (implicitly) assigning different importances to different nodes within a neighborhood while dealing with different sized neighborhoods, and does not depend on knowing the entire graph structure upfront—thus addressing many of the theoretical issues with previous spectral-based approaches. Our models leveraging attention have successfully achieved or matched state-of-the-art performance across four well-established node classification benchmarks, both transductive and inductive (especially, with completely unseen graphs used for testing).

There are several potential improvements and extensions to graph attention networks that could be addressed as future work, such as overcoming the practical problems described in subsection 2.2 to be able to handle larger batch sizes. A particularly interesting research direction would be taking advantage of the attention mechanism to perform a thorough analysis on the model interpretability.

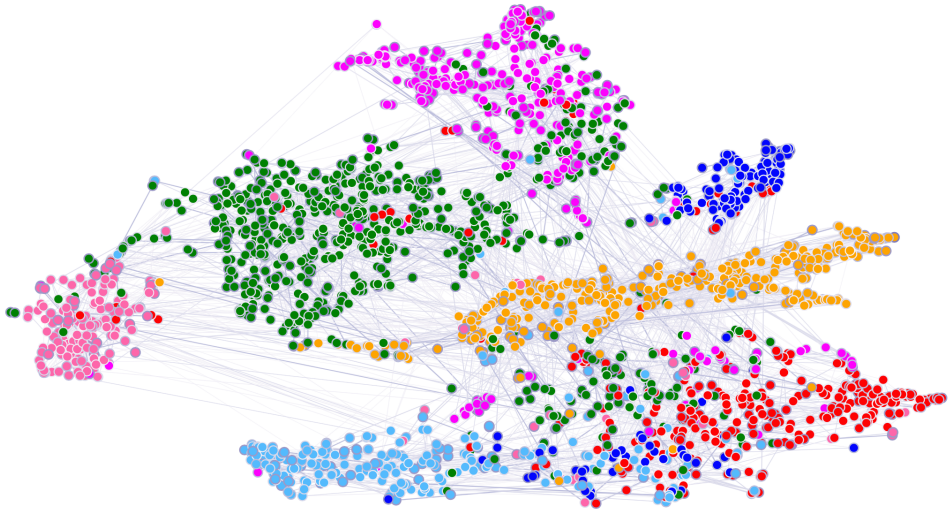


Figure 2: A t-SNE plot of the computed feature representations of a pre-trained GAT model's first hidden layer on the Cora dataset. Node colors denote classes. Edge thickness indicates aggregated normalized attention coefficients between nodes i and j, across all eight attention heads ($\sum_{k=1}^{K} \alpha_{ij}^k + \alpha_{ji}^k$).

Moreover, extending the method to perform graph classification instead of node classification would also be relevant from the application perspective. Finally, extending the model to incorporate edge features (possibly indicating relationship among nodes) would allow us to tackle a larger variety of problems.

ACKNOWLEDGEMENTS

The authors would like to thank the developers of TensorFlow (Abadi et al., 2015). PV and PL have received funding from the European Union's Horizon 2020 research and innovation programme PROPAG-AGEING under grant agreement No 634821. We further acknowledge the support of the following agencies for research funding and computing support: CIFAR, Canada Research Chairs, Compute Canada and Calcul Québec, as well as NVIDIA for the generous GPU support. Special thanks to: Benjamin Day and Fabian Jansen for kindly pointing out issues in a previous iteration of the paper; Michał Drożdżal for useful discussions, feedback and support; and Gaétan Marceau for reviewing the paper prior to submission.

REFERENCES

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. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

James Atwood and Don Towsley. Diffusion-convolutional neural networks. In *Advances in Neural Information Processing Systems*, pp. 1993–2001, 2016.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *International Conference on Learning Representations (ICLR)*, 2015.

Mikhail Belkin, Partha Niyogi, and Vikas Sindhwani. Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. *Journal of machine learning research*, 7 (Nov):2399–2434, 2006.

Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *International Conference on Learning Representations (ICLR)*, 2014.

Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. *arXiv preprint arXiv:1601.06733*, 2016.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *International Conference on Learning Representations (ICLR)*, 2016.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pp. 3844–3852, 2016.

Misha Denil, Sergio Gómez Colmenarejo, Serkan Cabi, David Saxton, and Nando de Freitas. Programmable agents. *arXiv preprint arXiv:1706.06383*, 2017.

Yan Duan, Marcin Andrychowicz, Bradly Stadie, Jonathan Ho, Jonas Schneider, Ilya Sutskever, Pieter Abbeel, and Wojciech Zaremba. One-shot imitation learning. *arXiv preprint arXiv:1703.07326*, 2017.

David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, pp. 2224–2232, 2015.

Paolo Frasconi, Marco Gori, and Alessandro Sperduti. A general framework for adaptive processing of data structures. *IEEE transactions on Neural Networks*, 9(5):768–786, 1998.

Jonas Gehring, Michael Auli, David Grangier, and Yann N. Dauphin. A convolutional encoder model for neural machine translation. *CoRR*, abs/1611.02344, 2016. URL http://arxiv.org/abs/1611.02344.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 249–256, 2010.

Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *IEEE International Joint Conference on Neural Networks*, pp. 729734, 2005.

William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Neural Information Processing Systems (NIPS)*, 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163*, 2015.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Yedid Hoshen. Vain: Attentional multi-agent predictive modeling. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 30*, pp. 2698–2708. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/6863-vain-attentional-multi-agent-predictive-modeling.pdf.

Simon Jégou, Michal Drozdzal, David Vázquez, Adriana Romero, and Yoshua Bengio. The one hundred layers tiramisu: Fully convolutional densenets for semantic segmentation. In *Workshop on Computer Vision in Vehicle Technology CVPRW*, 2017.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations (ICLR)*, 2017.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *International Conference on Learning Representations (ICLR)*, 2016.

Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding. *arXiv preprint arXiv:1703.03130*, 2017.

Qing Lu and Lise Getoor. Link-based classification. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pp. 496–503, 2003.

Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodolà, Jan Svoboda, and Michael M Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. *arXiv preprint arXiv:1611.08402*, 2016.

Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *Proceedings of The 33rd International Conference on Machine Learning*, volume 48, pp. 2014–2023, 2016.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 701–710. ACM, 2014.

Sam T. Roweis and Lawrence K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290:2323–2326, 2000.

Adam Santoro, David Raposo, David GT Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Timothy Lillicrap. A simple neural network module for relational reasoning. *arXiv preprint arXiv:1706.01427*, 2017.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93, 2008.

A. Sperduti and A. Starita. Supervised neural networks for the classification of structures. *Trans. Neur. Netw.*, 8(3):714–735, May 1997. ISSN 1045-9227. doi: 10.1109/72.572108.

Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

Aravind Subramanian, Pablo Tamayo, Vamsi K Mootha, Sayan Mukherjee, Benjamin L Ebert, Michael A Gillette, Amanda Paulovich, Scott L Pomeroy, Todd R Golub, Eric S Lander, et al. Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles. *Proceedings of the National Academy of Sciences*, 102(43):15545–15550, 2005.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

Jason Weston, Frédéric Ratle, Hossein Mobahi, and Ronan Collobert. Deep learning via semi-supervised embedding. In *Neural Networks: Tricks of the Trade*, pp. 639–655. Springer, 2012.

Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *CoRR*, abs/1410.3916, 2014. URL http://arxiv.org/abs/1410.3916.

Zhilin Yang, William Cohen, and Ruslan Salakhudinov. Revisiting semi-supervised learning with graph embeddings. In *International Conference on Machine Learning*, pp. 40–48, 2016.

Xiaojin Zhu, Zoubin Ghahramani, and John D Lafferty. Semi-supervised learning using gaussian fields and harmonic functions. In *Proceedings of the 20th International conference on Machine learning (ICML-03)*, pp. 912–919, 2003.

Marinka Zitnik and Jure Leskovec. Predicting multicellular function through multi-layer tissue networks. *Bioinformatics*, 33(14):i190–i198, 2017.

Simplifying Graph Convolutional Networks

Felix Wu [*1] **Tianyi Zhang** [*1] **Amauri Holanda de Souza Jr.** [*1 2] **Christopher Fifty** [1] **Tao Yu** [1]
Kilian Q. Weinberger [1]

Abstract

Graph Convolutional Networks (GCNs) and their variants have experienced significant attention and have become the de facto methods for learning graph representations. GCNs derive inspiration primarily from recent deep learning approaches, and as a result, may inherit unnecessary complexity and redundant computation. In this paper, we reduce this excess complexity through successively removing nonlinearities and collapsing weight matrices between consecutive layers. We theoretically analyze the resulting linear model and show that it corresponds to a fixed low-pass filter followed by a linear classifier. Notably, our experimental evaluation demonstrates that these simplifications do not negatively impact accuracy in many downstream applications. Moreover, the resulting model scales to larger datasets, is naturally interpretable, and yields up to two orders of magnitude speedup over FastGCN.

1. Introduction

Graph Convolutional Networks (GCNs) (Kipf & Welling, 2017) are an efficient variant of Convolutional Neural Networks (CNNs) on graphs. GCNs stack layers of learned first-order spectral filters followed by a nonlinear activation function to learn graph representations. Recently, GCNs and subsequent variants have achieved state-of-the-art results in various application areas, including but not limited to citation networks (Kipf & Welling, 2017), social networks (Chen et al., 2018), applied chemistry (Liao et al., 2019), natural language processing (Yao et al., 2019; Han et al., 2012; Zhang et al., 2018c), and computer vision (Wang et al., 2018; Kampffmeyer et al., 2018).

Historically, the development of machine learning algorithms has followed a clear trend from initial simplicity to need-driven complexity. For instance, limitations of the linear Perceptron (Rosenblatt, 1958) motivated the development of the more complex but also more expressive neural network (or multi-layer Perceptrons, MLPs) (Rosenblatt, 1961). Similarly, simple pre-defined linear image filters (Sobel & Feldman, 1968; Harris & Stephens, 1988) eventually gave rise to nonlinear CNNs with learned convolutional kernels (Waibel et al., 1989; LeCun et al., 1989). As additional algorithmic complexity tends to complicate theoretical analysis and obfuscates understanding, it is typically only introduced for applications where simpler methods are insufficient. Arguably, most classifiers in real world applications are still linear (typically logistic regression), which are straight-forward to optimize and easy to interpret.

However, possibly because GCNs were proposed after the recent "renaissance" of neural networks, they tend to be a rare exception to this trend. GCNs are built upon multi-layer neural networks, and were never an extension of a simpler (insufficient) linear counterpart.

In this paper, we observe that GCNs inherit considerable complexity from their deep learning lineage, which can be burdensome and unnecessary for less demanding applications. Motivated by the glaring historic omission of a simpler predecessor, we aim to derive the simplest linear model that "could have" preceded the GCN, had a more "traditional" path been taken. We reduce the excess complexity of GCNs by repeatedly removing the nonlinearities between GCN layers and collapsing the resulting function into a single linear transformation. We empirically show that the final linear model exhibits comparable or even superior performance to GCNs on a variety of tasks while being computationally more efficient and fitting significantly fewer parameters. We refer to this simplified linear model as Simple Graph Convolution (SGC).

In contrast to its nonlinear counterparts, the SGC is intuitively interpretable and we provide a theoretical analysis from the graph convolution perspective. Notably, feature extraction in SGC corresponds to a single fixed filter applied to each feature dimension. Kipf & Welling (2017) empirically observe that the "renormalization trick", i.e. adding self-loops to the graph, improves accuracy, and we demon-

[*]Equal contribution [1]Cornell University [2]Federal Institute of Ceara (Brazil). Correspondence to: Felix Wu <fw245@cornell.edu>, Tianyi Zhang <tz58@cornell.edu>.

Proceedings of the 36^{th} International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019.

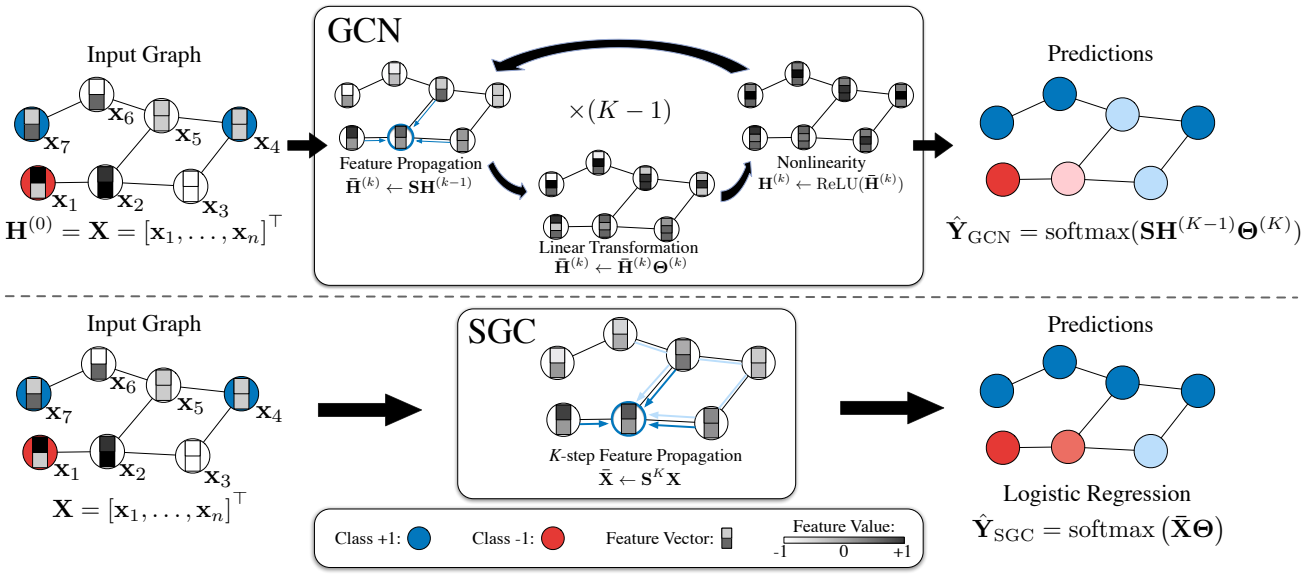


Figure 1. Schematic layout of a GCN v.s. a SGC. *Top row:* The GCN transforms the feature vectors repeatedly throughout K layers and then applies a linear classifier on the final representation. *Bottom row:* the SGC reduces the entire procedure to a simple feature propagation step followed by standard logistic regression.

strate that this method effectively shrinks the graph spectral domain, resulting in a low-pass-type filter when applied to SGC. Crucially, this filtering operation gives rise to locally smooth features across the graph (Bruna et al., 2014).

Through an empirical assessment on node classification benchmark datasets for citation and social networks, we show that the SGC achieves comparable performance to GCN and other state-of-the-art graph neural networks. However, it is significantly faster, and even outperforms FastGCN (Chen et al., 2018) by up to two orders of magnitude on the largest dataset (Reddit) in our evaluation. Finally, we demonstrate that SGC extrapolates its effectiveness to a wide-range of downstream tasks. In particular, SGC rivals, if not surpasses, GCN-based approaches on text classification, user geolocation, relation extraction, and zero-shot image classification tasks. The code is available on Github[1].

2. Simple Graph Convolution

We follow Kipf & Welling (2017) to introduce GCNs (and subsequently SGC) in the context of node classification. Here, GCNs take a graph with some labeled nodes as input and generate label predictions for all graph nodes. Let us formally define such a graph as $\mathcal{G} = (\mathcal{V}, \mathbf{A})$, where $\mathcal{V}$ represents the vertex set consisting of nodes $\{v_1, \ldots, v_n\}$, and $\mathbf{A} \in \mathbb{R}^{n \times n}$ is a symmetric (typically sparse) adjacency matrix where a_{ij} denotes the edge weight between nodes v_i and v_j. A missing edge is represented through $a_{ij} = 0$. We define the degree matrix $\mathbf{D} = \mathrm{diag}(d_1, \ldots, d_n)$ as a diagonal matrix where each entry on the diagonal is equal to the row-sum of the adjacency matrix $d_i = \sum_j a_{ij}$.

Each node v_i in the graph has a corresponding d-dimensional feature vector $\mathbf{x}_i \in \mathbb{R}^d$. The entire feature matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ stacks n feature vectors on top of one another, $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_n]^\top$. Each node belongs to one out of C classes and can be labeled with a C-dimensional one-hot vector $\mathbf{y}_i \in \{0, 1\}^C$. We only know the labels of a subset of the nodes and want to predict the unknown labels.

2.1. Graph Convolutional Networks

Similar to CNNs or MLPs, GCNs learn a new feature representation for the feature $\mathbf{x}_i$ of each node over multiple layers, which is subsequently used as input into a linear classifier. For the k-th graph convolution layer, we denote the input node representations of all nodes by the matrix $\mathbf{H}^{(k-1)}$ and the output node representations $\mathbf{H}^{(k)}$. Naturally, the initial node representations are just the original input features:

$$\mathbf{H}^{(0)} = \mathbf{X}, \tag{1}$$

which serve as input to the first GCN layer.

A K-layer GCN is identical to applying a K-layer MLP to the feature vector $\mathbf{x}_i$ of each node in the graph, except that the hidden representation of each node is averaged with its neighbors at the beginning of each layer. In each graph convolution layer, node representations are updated in three

[1] https://github.com/Tiiiger/SGC

stages: feature propagation, linear transformation, and a pointwise nonlinear activation (see Figure 1). For the sake of clarity, we describe each step in detail.

Feature propagation is what distinguishes a GCN from an MLP. At the beginning of each layer the features $\mathbf{h}_i$ of each node v_i are averaged with the feature vectors in its local neighborhood,

$$\bar{\mathbf{h}}_i^{(k)} \leftarrow \frac{1}{d_i+1}\mathbf{h}_i^{(k-1)} + \sum_{j=1}^{n} \frac{a_{ij}}{\sqrt{(d_i+1)(d_j+1)}}\mathbf{h}_j^{(k-1)}. \tag{2}$$

More compactly, we can express this update over the entire graph as a simple matrix operation. Let $\mathbf{S}$ denote the "normalized" adjacency matrix with added self-loops,

$$\mathbf{S} = \tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}, \tag{3}$$

where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ and $\tilde{\mathbf{D}}$ is the degree matrix of $\tilde{\mathbf{A}}$. The simultaneous update in Equation 2 for all nodes becomes a simple sparse matrix multiplication

$$\bar{\mathbf{H}}^{(k)} \leftarrow \mathbf{S}\mathbf{H}^{(k-1)}. \tag{4}$$

Intuitively, this step smoothes the hidden representations locally along the edges of the graph and ultimately encourages similar predictions among locally connected nodes.

Feature transformation and nonlinear transition. After the local smoothing, a GCN layer is identical to a standard MLP. Each layer is associated with a learned weight matrix $\mathbf{\Theta}^{(k)}$, and the smoothed hidden feature representations are transformed linearly. Finally, a nonlinear activation function such as ReLU is applied pointwise before outputting feature representation $\mathbf{H}^{(k)}$. In summary, the representation updating rule of the k-th layer is:

$$\mathbf{H}^{(k)} \leftarrow \mathrm{ReLU}\left(\bar{\mathbf{H}}^{(k)}\mathbf{\Theta}^{(k)}\right). \tag{5}$$

The pointwise nonlinear transformation of the k-th layer is followed by the feature propagation of the $(k+1)$-th layer.

Classifier. For node classification, and similar to a standard MLP, the last layer of a GCN predicts the labels using a *softmax* classifier. Denote the class predictions for n nodes as $\hat{\mathbf{Y}} \in \mathbb{R}^{n\times C}$ where $\hat{y}_{ic}$ denotes the probability of node i belongs to class c. The class prediction $\hat{\mathbf{Y}}$ of a K-layer GCN can be written as:

$$\hat{\mathbf{Y}}_{\mathrm{GCN}} = \mathrm{softmax}\left(\mathbf{S}\mathbf{H}^{(K-1)}\mathbf{\Theta}^{(K)}\right), \tag{6}$$

where $\mathrm{softmax}(\mathbf{x}) = \exp(\mathbf{x})/\sum_{c=1}^{C}\exp(x_c)$ acts as a normalizer across all classes.

2.2. Simple Graph Convolution

In a traditional MLP, deeper layers increase the expressivity because it allows the creation of feature hierarchies, e.g. features in the second layer build on top of the features of the first layer. In GCNs, the layers have a second important function: in each layer the hidden representations are averaged among neighbors that are one hop away. This implies that after k layers a node obtains feature information from all nodes that are $k-$hops away in the graph. This effect is similar to convolutional neural networks, where depth increases the receptive field of internal features (Hariharan et al., 2015). Although convolutional networks can benefit substantially from increased depth (Huang et al., 2016), typically MLPs obtain little benefit beyond 3 or 4 layers.

Linearization. We hypothesize that the nonlinearity between GCN layers is not critical - but that the majority of the benefit arises from the local averaging. We therefore remove the nonlinear transition functions between each layer and only keep the final softmax (in order to obtain probabilistic outputs). The resulting model is linear, but still has the same increased "receptive field" of a K-layer GCN,

$$\hat{\mathbf{Y}} = \mathrm{softmax}\left(\mathbf{S}\ldots\mathbf{S}\mathbf{S}\mathbf{X}\mathbf{\Theta}^{(1)}\mathbf{\Theta}^{(2)}\ldots\mathbf{\Theta}^{(K)}\right). \tag{7}$$

To simplify notation we can collapse the repeated multiplication with the normalized adjacency matrix $\mathbf{S}$ into a single matrix by raising $\mathbf{S}$ to the K-th power, $\mathbf{S}^K$. Further, we can reparameterize our weights into a single matrix $\mathbf{\Theta} = \mathbf{\Theta}^{(1)}\mathbf{\Theta}^{(2)}\ldots\mathbf{\Theta}^{(K)}$. The resulting classifier becomes

$$\hat{\mathbf{Y}}_{\mathrm{SGC}} = \mathrm{softmax}\left(\mathbf{S}^K\mathbf{X}\mathbf{\Theta}\right), \tag{8}$$

which we refer to as Simple Graph Convolution (SGC).

Logistic regression. Equation 8 gives rise to a natural and intuitive interpretation of SGC: by distinguishing between feature extraction and classifier, SGC consists of a fixed (i.e., parameter-free) feature extraction/smoothing component $\bar{\mathbf{X}} = \mathbf{S}^K\mathbf{X}$ followed by a linear logistic regression classifier $\hat{\mathbf{Y}} = \mathrm{softmax}(\bar{\mathbf{X}}\mathbf{\Theta})$. Since the computation of $\bar{\mathbf{X}}$ requires no weight it is essentially equivalent to a feature pre-processing step and the entire training of the model reduces to straight-forward multi-class logistic regression on the pre-processed features $\bar{\mathbf{X}}$.

Optimization details. The training of logistic regression is a well studied convex optimization problem and can be performed with any efficient second order method or stochastic gradient descent (Bottou, 2010). Provided the graph connectivity pattern is sufficiently sparse, SGD naturally scales to very large graph sizes and the training of SGC is drastically faster than that of GCNs.

3. Spectral Analysis

We now study SGC from a graph convolution perspective. We demonstrate that SGC corresponds to a fixed filter on the graph spectral domain. In addition, we show that adding self-loops to the original graph, i.e. the renormalization trick (Kipf & Welling, 2017), effectively shrinks the underlying graph spectrum. On this scaled domain, SGC acts as a low-pass filter that produces smooth features over the graph. As a result, nearby nodes tend to share similar representations and consequently predictions.

3.1. Preliminaries on Graph Convolutions

Analogous to the Euclidean domain, graph Fourier analysis relies on the spectral decomposition of graph Laplacians. The graph Laplacian $\mathbf{\Delta} = \mathbf{D} - \mathbf{A}$ (as well as its normalized version $\mathbf{\Delta}_{\text{sym}} = \mathbf{D}^{-1/2}\mathbf{\Delta}\mathbf{D}^{-1/2}$) is a symmetric positive semidefinite matrix with eigendecomposition $\mathbf{\Delta} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top$, where $\mathbf{U} \in \mathbb{R}^{n\times n}$ comprises orthonormal eigenvectors and $\mathbf{\Lambda} = \text{diag}(\lambda_1, \dots, \lambda_n)$ is a diagonal matrix of eigenvalues. The eigendecomposition of the Laplacian allows us to define the Fourier transform equivalent on the graph domain, where eigenvectors denote Fourier modes and eigenvalues denote frequencies of the graph. In this regard, let $\mathbf{x} \in \mathbb{R}^n$ be a signal defined on the vertices of the graph. We define the graph Fourier transform of $\mathbf{x}$ as $\hat{\mathbf{x}} = \mathbf{U}^\top\mathbf{x}$, with inverse operation given by $\mathbf{x} = \mathbf{U}\hat{\mathbf{x}}$. Thus, the graph convolution operation between signal $\mathbf{x}$ and filter $\mathbf{g}$ is

$$\mathbf{g} * \mathbf{x} = \mathbf{U}\left((\mathbf{U}^\top\mathbf{g}) \odot (\mathbf{U}^\top\mathbf{x})\right) = \mathbf{U}\hat{\mathbf{G}}\mathbf{U}^\top\mathbf{x}, \quad (9)$$

where $\hat{\mathbf{G}} = \text{diag}\left(\hat{g}_1, \dots, \hat{g}_n\right)$ denotes a diagonal matrix in which the diagonal corresponds to spectral filter coefficients.

Graph convolutions can be approximated by k-th order polynomials of Laplacians

$$\mathbf{U}\hat{\mathbf{G}}\mathbf{U}^\top\mathbf{x} \approx \sum_{i=0}^{k}\theta_i\mathbf{\Delta}^i\mathbf{x} = \mathbf{U}\left(\sum_{i=0}^{k}\theta_i\mathbf{\Lambda}^i\right)\mathbf{U}^\top\mathbf{x}, \quad (10)$$

where θ_i denotes coefficients. In this case, filter coefficients correspond to polynomials of the Laplacian eigenvalues, i.e., $\hat{\mathbf{G}} = \sum_i \theta_i\mathbf{\Lambda}^i$ or equivalently $\hat{g}(\lambda_j) = \sum_i \theta_i\lambda_j^i$.

Graph Convolutional Networks (GCNs) (Kipf & Welling, 2017) employ an affine approximation ($k = 1$) of Equation 10 with coefficients $\theta_0 = 2\theta$ and $\theta_1 = -\theta$ from which we attain the basic GCN convolution operation

$$\mathbf{g} * \mathbf{x} = \theta(\mathbf{I} + \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2})\mathbf{x}. \quad (11)$$

In their final design, Kipf & Welling (2017) replace the matrix $\mathbf{I} + \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ by a normalized version $\tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}$ where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ and consequently $\tilde{\mathbf{D}} = \mathbf{D} + \mathbf{I}$, dubbed the *renormalization trick*. Finally, by generalizing the convolution to work with multiple filters in a d-channel input and layering the model with nonlinear activation functions between each layer, we have the GCN propagation rule as defined in Equation 5.

3.2. SGC and Low-Pass Filtering

The initial first-order Chebyshev filter derived in GCNs corresponds to the propagation matrix $\mathbf{S}_{\text{1-order}} = \mathbf{I} + \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ (see Equation 11). Since the normalized Laplacian is $\mathbf{\Delta}_{\text{sym}} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$, then $\mathbf{S}_{\text{1-order}} = 2\mathbf{I} - \mathbf{\Delta}_{\text{sym}}$. Therefore, feature propagation with $\mathbf{S}_{\text{1-order}}^K$ implies filter coefficients $\hat{g}_i = \hat{g}(\lambda_i) = (2 - \lambda_i)^K$, where λ_i denotes the eigenvalues of $\mathbf{\Delta}_{\text{sym}}$. Figure 2 illustrates the filtering operation related to $\mathbf{S}_{\text{1-order}}$ for a varying number of propagation steps $K \in \{1, \dots, 6\}$. As one may observe, high powers of $\mathbf{S}_{\text{1-order}}$ lead to exploding filter coefficients and undesirably over-amplify signals at frequencies $\lambda_i < 1$.

To tackle potential numerical issues associated with the first-order Chebyshev filter, Kipf & Welling (2017) propose the *renormalization trick*. Basically, it consists of replacing $\mathbf{S}_{\text{1-order}}$ by the normalized adjacency matrix after adding self-loops for all nodes. We call the resulting propagation matrix the augmented normalized adjacency matrix $\tilde{\mathbf{S}}_{\text{adj}} = \tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}$, where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ and $\tilde{\mathbf{D}} = \mathbf{D} + \mathbf{I}$. Correspondingly, we define the augmented normalized Laplacian $\tilde{\mathbf{\Delta}}_{\text{sym}} = \mathbf{I} - \tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}$. Thus, we can describe the spectral filters associated with $\tilde{\mathbf{S}}_{\text{adj}}$ as a polynomial of the eigenvalues of the underlying Laplacian, i.e., $\hat{g}(\tilde{\lambda}_i) = (1 - \tilde{\lambda}_i)^K$, where $\tilde{\lambda}_i$ are eigenvalues of $\tilde{\mathbf{\Delta}}_{\text{sym}}$.

We now analyze the spectrum of $\tilde{\mathbf{\Delta}}_{\text{sym}}$ and show that adding self-loops to graphs shrinks the spectrum (eigenvalues) of the corresponding normalized Laplacian.

Theorem 1. *Let $\mathbf{A}$ be the adjacency matrix of an undirected, weighted, simple graph $\mathcal{G}$ without isolated nodes and with corresponding degree matrix $\mathbf{D}$. Let $\tilde{\mathbf{A}} = \mathbf{A} + \gamma\mathbf{I}$, such that $\gamma > 0$, be the augmented adjacency matrix with corresponding degree matrix $\tilde{\mathbf{D}}$. Also, let λ_1 and λ_n denote the smallest and largest eigenvalues of $\mathbf{\Delta}_{sym} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$; similarly, let $\tilde{\lambda}_1$ and $\tilde{\lambda}_n$ be the smallest and largest eigenvalues of $\tilde{\mathbf{\Delta}}_{sym} = \mathbf{I} - \tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}$. We have that*

$$0 = \lambda_1 = \tilde{\lambda}_1 < \tilde{\lambda}_n < \lambda_n. \quad (12)$$

Theorem 1 shows that the largest eigenvalue of the normalized graph Laplacian becomes smaller after adding self-loops $\gamma > 0$ (see supplementary materials for the proof).

Figure 2 depicts the filtering operations associated with the normalized adjacency $\mathbf{S}_{\text{adj}} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ and its augmented variant $\tilde{\mathbf{S}}_{\text{adj}} = \tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}$ on the Cora dataset (Sen et al., 2008). Feature propagation with $\mathbf{S}_{\text{adj}}$ corresponds to filters $g(\lambda_i) = (1 - \lambda_i)^K$ in the spectral range

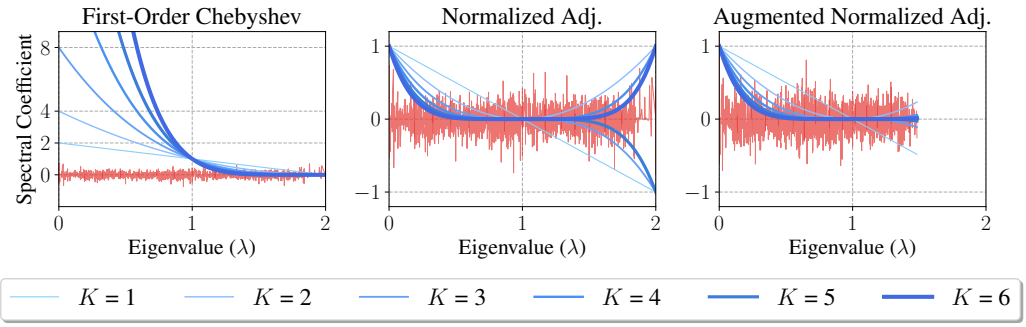


Figure 2. Feature (red) and filters (blue) spectral coefficients for different propagation matrices on Cora dataset (3rd feature).

$[0, 2]$; therefore odd powers of $\mathbf{S}_{\text{adj}}$ yield negative filter coefficients at frequencies $\lambda_i > 1$. By adding self-loops ($\tilde{\mathbf{S}}_{\text{adj}}$), the largest eigenvalue shrinks from 2 to approximately 1.5 and then eliminates the effect of negative coefficients. Moreover, this scaled spectrum allows the filter defined by taking powers $K > 1$ of $\tilde{\mathbf{S}}_{\text{adj}}$ to act as a low-pass-type filters. In supplementary material, we empirically evaluate different choices for the propagation matrix.

4. Related Works

4.1. Graph Neural Networks

Bruna et al. (2014) first propose a spectral graph-based extension of convolutional networks to graphs. In a follow-up work, ChebyNets (Defferrard et al., 2016) define graph convolutions using Chebyshev polynomials to remove the computationally expensive Laplacian eigendecomposition. GCNs (Kipf & Welling, 2017) further simplify graph convolutions by stacking layers of first-order Chebyshev polynomial filters with a redefined propagation matrix $\mathbf{S}$. Chen et al. (2018) propose an efficient variant of GCN based on importance sampling, and Hamilton et al. (2017) propose a framework based on sampling and aggregation. Atwood & Towsley (2016), Abu-El-Haija et al. (2018), and Liao et al. (2019) exploit multi-scale information by raising $\mathbf{S}$ to higher order. Xu et al. (2019) study the expressiveness of graph neural networks in terms of their ability to distinguish any two graphs and introduce Graph Isomorphism Network, which is proved to be as powerful as the Weisfeiler-Lehman test for graph isomorphism. Klicpera et al. (2019) separate the non-linear transformation from propagation by using a neural network followed by a personalized random walk. There are many other graph neural models (Monti et al., 2017; Duran & Niepert, 2017; Li et al., 2018); we refer to Zhou et al. (2018); Battaglia et al. (2018); Wu et al. (2019) for a more comprehensive review.

Previous publications have pointed out that simpler, sometimes linear models can be effective for node/graph classification tasks. Thekumparampil et al. (2018) empirically show that a linear version of GCN can perform competitively and propose an attention-based GCN variant. Cai & Wang (2018) propose an effective linear baseline for graph classification using node degree statistics. Eliav & Cohen (2018) show that models which use linear feature/label propagation steps can benefit from self-training strategies. Li et al. (2019) propose a generalized version of label propagation and provide a similar spectral analysis of the renormalization trick.

Graph Attentional Models learn to assign different edge weights at each layer based on node features and have achieved state-of-the-art results on several graph learning tasks (Velickovic et al., 2018; Thekumparampil et al., 2018; Zhang et al., 2018a; Kampffmeyer et al., 2018). However, the attention mechanism usually adds significant overhead to computation and memory usage. We refer the readers to Lee et al. (2018) for further comparison.

4.2. Other Works on Graphs

Graph methodologies can roughly be categorized into two approaches: graph embedding methods and graph laplacian regularization methods. Graph embedding methods (Weston et al., 2008; Perozzi et al., 2014; Yang et al., 2016; Velikovi et al., 2019) represent nodes as high-dimensional feature vectors. Among them, DeepWalk (Perozzi et al., 2014) and Deep Graph Infomax (DGI) (Velikovi et al., 2019) use unsupervised strategies to learn graph embeddings. DeepWalk relies on truncated random walk and uses a skip-gram model to generate embeddings, whereas DGI trains a graph convolutional encoder through maximizing mutual information. Graph Laplacian regularization (Zhu et al., 2003; Zhou et al., 2004; Belkin & Niyogi, 2004; Belkin et al., 2006) introduce a regularization term based on graph structure which forces nodes to have similar labels to their neighbors. Label Propagation (Zhu et al., 2003) makes predictions by spreading label information from labeled nodes to their neighbors until convergence.

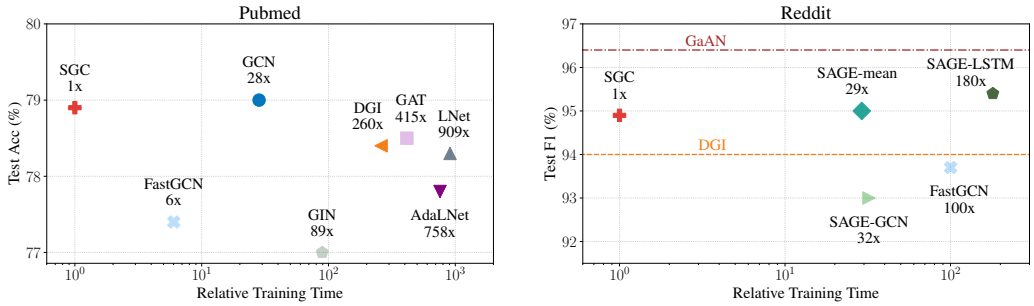


Figure 3. Performance over training time on Pubmed and Reddit. SGC is the fastest while achieving competitive performance. We are not able to benchmark the training time of GaAN and DGI on Reddit because the implementations are not released.

Table 1. Dataset statistics of the citation networks and Reddit.

Dataset	# Nodes	# Edges	Train/Dev/Test Nodes
Cora	2,708	5,429	140/500/1,000
Citeseer	3,327	4,732	120/500/1,000
Pubmed	19,717	44,338	60/500/1,000
Reddit	233K	11.6M	152K/24K/55K

5. Experiments and Discussion

We first evaluate SGC on citation networks and social networks and then extend our empirical analysis to a wide range of downstream tasks.

5.1. Citation Networks & Social Networks

We evaluate the semi-supervised node classification performance of SGC on the Cora, Citeseer, and Pubmed citation network datasets (Table 2) (Sen et al., 2008). We supplement our citation network analysis by using SGC to inductively predict community structure on Reddit (Table 3), which consists of a much larger graph. Dataset statistics are summarized in Table 1.

Datasets and experimental setup. On the citation networks, we train SGC for 100 epochs using Adam (Kingma & Ba, 2015) with learning rate 0.2. In addition, we use weight decay and tune this hyperparameter on each dataset using hyperopt (Bergstra et al., 2015) for 60 iterations on the public split validation set. Experiments on citation networks are conducted *transductively*. On the Reddit dataset, we train SGC with L-BFGS (Liu & Nocedal, 1989) using no regularization, and remarkably, training converges in 2 steps. We evaluate SGC *inductively* by following Chen et al. (2018): we train SGC on a subgraph comprising only training nodes and test with the original graph. On all datasets, we tune the number of epochs based on both convergence behavior and validation accuracy.

Table 2. Test accuracy (%) averaged over 10 runs on citation networks. †We remove the outliers (accuracy $< 75/65/75\%$) when calculating their statistics due to high variance.

	Cora	Citeseer	Pubmed
Numbers from literature:			
GCN	81.5	70.3	79.0
GAT	83.0 ± 0.7	72.5 ± 0.7	79.0 ± 0.3
GLN	81.2 ± 0.1	70.9 ± 0.1	78.9 ± 0.1
AGNN	83.1 ± 0.1	71.7 ± 0.1	79.9 ± 0.1
LNet	79.5 ± 1.8	66.2 ± 1.9	78.3 ± 0.3
AdaLNet	80.4 ± 1.1	68.7 ± 1.0	78.1 ± 0.4
DeepWalk	70.7 ± 0.6	51.4 ± 0.5	76.8 ± 0.6
DGI	82.3 ± 0.6	71.8 ± 0.7	76.8 ± 0.6
Our experiments:			
GCN	81.4 ± 0.4	70.9 ± 0.5	79.0 ± 0.4
GAT	83.3 ± 0.7	72.6 ± 0.6	78.5 ± 0.3
FastGCN	79.8 ± 0.3	68.8 ± 0.6	77.4 ± 0.3
GIN	77.6 ± 1.1	66.1 ± 0.9	77.0 ± 1.2
LNet	$80.2 \pm 3.0^{\dagger}$	67.3 ± 0.5	$78.3 \pm 0.6^{\dagger}$
AdaLNet	$81.9 \pm 1.9^{\dagger}$	$70.6 \pm 0.8^{\dagger}$	$77.8 \pm 0.7^{\dagger}$
DGI	82.5 ± 0.7	71.6 ± 0.7	78.4 ± 0.7
SGC	81.0 ± 0.0	71.9 ± 0.1	78.9 ± 0.0

Table 3. Test Micro F1 Score (%) averaged over 10 runs on Reddit. Performances of models are cited from their original papers. **OOM:** Out of memory.

Setting	Model	Test F1
Supervised	GaAN	96.4
	SAGE-mean	95.0
	SAGE-LSTM	95.4
	SAGE-GCN	93.0
	FastGCN	93.7
	GCN	**OOM**
Unsupervised	SAGE-mean	89.7
	SAGE-LSTM	90.7
	SAGE-GCN	90.8
	DGI	94.0
No Learning	Random-Init DGI	93.3
	SGC	94.9

Baselines. For citation networks, we compare against GCN (Kipf & Welling, 2017) GAT (Velickovic et al., 2018) FastGCN (Chen et al., 2018) LNet, AdaLNet (Liao et al., 2019) and DGI (Velikovi et al., 2019) using the publicly released implementations. Since GIN is not initially evaluated on citation networks, we implement GIN following Xu et al. (2019) and use hyperopt to tune weight decay and learning rate for 60 iterations. Moreover, we tune the hidden dimension by hand.

For Reddit, we compare SGC to the reported performance of GaAN (Zhang et al., 2018a), supervised and unsupervised variants of GraphSAGE (Hamilton et al., 2017), FastGCN, and DGI. Table 3 also highlights the setting of the feature extraction step for each method. We note that SGC involves no learning because the feature extraction step, $\mathbf{S}^K\mathbf{X}$, has no parameter. Both unsupervised and no-learning approaches train logistic regression models with labels afterward.

Performance. Based on results in Table 2 and Table 3, we conclude that SGC is very competitive. Table 2 shows the performance of SGC can match the performance of GCN and state-of-the-art graph networks on citation networks. In particular on Citeseer, SGC is about 1% better than GCN, and we reason this performance boost is caused by SGC having fewer parameters and therefore suffering less from overfitting. Remarkably, GIN performs slight worse because of overfitting. Also, both LNet and AdaLNet are unstable on citation networks. On Reddit, Table 3 shows that SGC outperforms the previous sampling-based GCN variants, SAGE-GCN and FastGCN by more than 1%.

Notably, Velikovi et al. (2019) report that the performance of a randomly initialized DGI encoder nearly matches that of a trained encoder; however, both models underperform SGC on Reddit. This result may suggest that the extra weights and nonlinearities in the DGI encoder are superfluous, if not outright detrimental.

Efficiency. In Figure 3, we plot the performance of the state-of-the-arts graph networks over their training time relative to that of SGC on the Pubmed and Reddit datasets. In particular, we precompute $\mathbf{S}^K\mathbf{X}$ and the training time of SGC takes into account this precomputation time. We measure the training time on a NVIDIA GTX 1080 Ti GPU and present the benchmark details in supplementary materials.

On large graphs (e.g. Reddit), GCN cannot be trained due to excessive memory requirements. Previous approaches tackle this limitation by either sampling to reduce neighborhood size (Chen et al., 2018; Hamilton et al., 2017) or limiting their model sizes (Velikovi et al., 2019). By applying a fixed filter and precomputing $\mathbf{S}^K\mathbf{X}$, SGC minimizes memory usage and only learns a single weight matrix during training. Since $\mathbf{S}$ is typically sparse and K is usually small, we can exploit fast sparse-dense matrix multiplication to compute $\mathbf{S}^K\mathbf{X}$. Figure 3 shows that SGC can be trained up to two orders of magnitude faster than fast sampling-based methods while having little or no drop in performance.

Table 4. Test Accuracy (%) on text classification datasets. The numbers are averaged over 10 runs.

Dataset	Model	Test Acc. ↑	Time (seconds) ↓
20NG	GCN	87.9 ± 0.2	1205.1 ± 144.5
	SGC	88.5 ± 0.1	19.06 ± 0.15
R8	GCN	97.0 ± 0.2	129.6 ± 9.9
	SGC	97.2 ± 0.1	1.90 ± 0.03
R52	GCN	93.8 ± 0.2	245.0 ± 13.0
	SGC	94.0 ± 0.2	3.01 ± 0.01
Ohsumed	GCN	68.2 ± 0.4	252.4 ± 14.7
	SGC	68.5 ± 0.3	3.02 ± 0.02
MR	GCN	76.3 ± 0.3	16.1 ± 0.4
	SGC	75.9 ± 0.3	4.00 ± 0.04

Table 5. Test accuracy (%) within 161 miles on semi-supervised user geolocation. The numbers are averaged over 5 runs.

Dataset	Model	Acc.@161↑	Time ↓
GEOTEXT	GCN+H	60.6 ± 0.2	153.0s
	SGC	61.1 ± 0.1	5.6s
TWITTER-US	GCN+H	61.9 ± 0.2	9h 54m
	SGC	62.5 ± 0.1	4h 33m
TWITTER-WORLD	GCN+H	53.6 ± 0.2	2d 05h 17m
	SGC	54.1 ± 0.2	22h 53m

5.2. Downstream Tasks

We extend our empirical evaluation to 5 downstream applications — text classification, semi-supervised user geolocation, relation extraction, zero-shot image classification, and graph classification — to study the applicability of SGC. We describe experimental setup in supplementary materials.

Text classification assigns labels to documents. Yao et al. (2019) use a 2-layer GCN to achieve state-of-the-art results by creating a corpus-level graph which treats both documents and words as nodes in a graph. Word-word edge weights are pointwise mutual information (PMI) and word-document edge weights are normalized TF-IDF scores. Table 4 shows that an SGC ($K = 2$) rivals their model on 5 benchmark datasets, while being up to $83.6\times$ faster.

Semi-supervised user geolocation locates the "home" position of users on social media given users' posts, connections among users, and a small number of labelled users. Rahimi et al. (2018) apply GCNs with highway connections on this task and achieve close to state-of-the-art results. Ta-

Table 6. Test Accuracy (%) on Relation Extraction. The numbers are averaged over 10 runs.

TACRED	Test Accuracy ↑
C-GCN (Zhang et al., 2018c)	66.4
C-GCN	66.4 ± 0.4
C-SGC	67.0 ± 0.4

Table 7. Top-1 accuracy (%) averaged over 10 runs in the 2-hop and 3-hop setting of the zero-shot image task on ImageNet. ADGPM (Kampffmeyer et al., 2018) and EXEM 1-nns (Changpinyo et al., 2018) use more powerful visual features.

Model	# Param. ↓	2-hop Acc. ↑	3-hop Acc. ↑
Unseen categories only:			
EXEM 1-nns	-	27.0	7.1
ADGPM	-	26.6	6.3
GCNZ	-	19.8	4.1
GCNZ (ours)	$9.5M$	20.9 ± 0.2	4.3 ± 0.0
MLP-SGCZ (ours)	$4.3M$	21.2 ± 0.2	4.4 ± 0.1
Unseen categories & seen categories:			
ADGPM	-	10.3	2.9
GCNZ	-	9.7	2.2
GCNZ (ours)	$9.5M$	10.0 ± 0.2	2.4 ± 0.0
MLP-SGCZ (ours)	$4.3M$	10.5 ± 0.1	2.5 ± 0.0

ble 5 shows that SGC outperforms GCN with highway connections on GEOTEXT (Eisenstein et al., 2010), TWITTER-US (Roller et al., 2012), and TWITTER-WORLD (Han et al., 2012) under Rahimi et al. (2018)'s framework, while saving 30+ hours on TWITTER-WORLD.

Relation extraction involves predicting the relation between subject and object in a sentence. Zhang et al. (2018c) propose C-GCN which uses an LSTM (Hochreiter & Schmidhuber, 1997) followed by a GCN and an MLP. We replace GCN with SGC ($K = 2$) and call the resulting model C-SGC. Table 6 shows that C-SGR sets new state-of-the-art on TACRED (Zhang et al., 2017).

Zero-shot image classification consists of learning an image classifier without access to any images or labels from the test categories. GCNZ (Wang et al., 2018) uses a GCN to map the category names — based on their relations in WordNet (Miller, 1995) — to image feature domain, and find the most similar category to a query image feature vector. Table 7 shows that replacing GCN with an MLP followed by SGC can improve performance while reducing the number of parameters by 55%. We find that an MLP feature extractor is necessary in order to map the pretrained GloVe vectors to the space of visual features extracted by a ResNet-50. Again, this downstream application demonstrates that learned graph convolution filters are superfluous; similar to Changpinyo et al. (2018)'s observation that GCNs may not be necessary.

Graph classification requires models to use graph structure to categorize graphs. Xu et al. (2019) theoretically show that GCNs are not sufficient to distinguish certain graph structures and show that their GIN is more expressive and achieves state-of-the-art results on various graph classification datasets. We replace the GCN in DCGCN (Zhang et al., 2018b) with an SGC and get 71.0% and 76.2% on NCI1 and COLLAB datasets (Yanardag & Vishwanathan, 2015) respectively, which is on par with an GCN counterpart, but far behind GIN. Similarly, on QM8 quantum chemistry dataset (Ramakrishnan et al., 2015), more advanced AdaLNet and LNet (Liao et al., 2019) get 0.01 MAE on QM8, outperforming SGC's 0.03 MAE by a large margin.

6. Conclusion

In order to better understand and explain the mechanisms of GCNs, we explore the simplest possible formulation of a graph convolutional model, SGC. The algorithm is almost trivial, a graph based pre-processing step followed by standard multi-class logistic regression. However, the performance of SGC rivals — if not surpasses — the performance of GCNs and state-of-the-art graph neural network models across a wide range of graph learning tasks. Moreover by precomputing the fixed feature extractor $\mathbf{S}^K$, training time is reduced to a record low. For example on the Reddit dataset, SGC can be trained up to two orders of magnitude faster than sampling-based GCN variants.

In addition to our empirical analysis, we analyze SGC from a convolution perspective and manifest this method as a low-pass-type filter on the spectral domain. Low-pass-type filters capture low-frequency signals, which corresponds with smoothing features across a graph in this setting. Our analysis also provides insight into the empirical boost of the "renormalization trick" and demonstrates how shrinking the spectral domain leads to a low-pass-type filter which underpins SGC.

Ultimately, the strong performance of SGC sheds light onto GCNs. It is likely that the expressive power of GCNs originates primarily from the repeated graph propagation (which SGC preserves) rather than the nonlinear feature extraction (which it doesn't.)

Given its empirical performance, efficiency, and interpretability, we argue that the SGC should be highly beneficial to the community in at least three ways: (1) as a first model to try, especially for node classification tasks; (2) as a simple baseline for comparison with future graph learning models; (3) as a starting point for future research in graph learning — returning to the historic machine learning practice to develop complex from simple models.

Acknowledgement

This research is supported in part by grants from the National Science Foundation (III-1618134, III-1526012, IIS1149882, IIS-1724282, and TRIPODS-1740822), the Office of Naval Research DOD (N00014-17-1-2175), Bill and Melinda Gates Foundation, and Facebook Research. We are thankful for generous support by SAP America Inc. Amauri Holanda de Souza Jr. thanks CNPq (Brazilian Council for Scientific and Technological Development) for the financial support. We appreciate the discussion with Xiang Fu, Shengyuan Hu, Shangdi Yu, Wei-Lun Chao and Geoff Pleiss as well as the figure design support from Boyi Li.

References

Abu-El-Haija, S., Kapoor, A., Perozzi, B., and Lee, J. N-GCN: Multi-scale graph convolution for semi-supervised node classification. *arXiv preprint arXiv:1802.08888*, 2018.

Atwood, J. and Towsley, D. Diffusion-convolutional neural networks. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 29*, pp. 1993–2001. Curran Associates, Inc., 2016.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Belkin, M. and Niyogi, P. Semi-supervised learning on riemannian manifolds. *Machine Learning*, 56(1-3):209–239, 2004.

Belkin, M., Niyogi, P., and Sindhwani, V. Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. *Journal of Machine Learning Research*, 7:2399–2434, 2006.

Bergstra, J., Komer, B., Eliasmith, C., Yamins, D., and Cox, D. D. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. *Computational Science & Discovery*, 8(1), 2015.

Bottou, L. Large-scale machine learning with stochastic gradient descent. In *Proceedings of 19th International Conference on Computational Statistics*, pp. 177–186. Springer, 2010.

Bruna, J., Zaremba, W., Szlam, A., and Lecun, Y. Spectral networks and locally connected networks on graphs. In *International Conference on Learning Representations (ICLR'2014)*, 2014.

Cai, C. and Wang, Y. A simple yet effective baseline for non-attribute graph classification, 2018.

Changpinyo, S., Chao, W.-L., Gong, B., and Sha, F. Classifier and exemplar synthesis for zero-shot learning. *arXiv preprint arXiv:1812.06423*, 2018.

Chen, J., Ma, T., and Xiao, C. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *International Conference on Learning Representations (ICLR'2018)*, 2018.

Defferrard, M., Bresson, X., and Vandergheynst, P. Convolutional neural networks on graphs with fast localized spectral filtering. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 29*, pp. 3844–3852. Curran Associates, Inc., 2016.

Duran, A. G. and Niepert, M. Learning graph representations with embedding propagation. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30*, pp. 5119–5130. Curran Associates, Inc., 2017.

Eisenstein, J., O'Connor, B., Smith, N. A., and Xing, E. P. A latent variable model for geographic lexical variation. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pp. 1277–1287. Association for Computational Linguistics, 2010.

Eliav, B. and Cohen, E. Bootstrapped graph diffusions: Exposing the power of nonlinearity. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2 (1):10:1–10:19, 2018.

Hamilton, W., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30*, pp. 1024–1034. Curran Associates, Inc., 2017.

Han, B., Cook, P., and Baldwin, T. Geolocation prediction in social media data by finding location indicative words. In *Proceedings of the 24th International Conference on Computational Linguistics*, pp. 1045–1062, 2012.

Hariharan, B., Arbeláez, P., Girshick, R., and Malik, J. Hypercolumns for object segmentation and fine-grained localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 447–456, 2015.

Harris, C. and Stephens, M. A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*, pp. 147–151, 1988.

Hochreiter, S. and Schmidhuber, J. Long Short-Term Memory. *Neural Computation*, 9:1735–1780, 1997.

Huang, G., Sun, Y., Liu, Z., Sedra, D., and Weinberger, K. Q. Deep networks with stochastic depth. In *European Conference on Computer Vision*, pp. 646–661. Springer, 2016.

Kampffmeyer, M., Chen, Y., Liang, X., Wang, H., Zhang, Y., and Xing, E. P. Rethinking knowledge graph propagation for zero-shot learning. *arXiv preprint arXiv:1805.11724*, 2018.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR'2015)*, 2015.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR'2017)*, 2017.

Klicpera, J., Bojchevski, A., and Günnemann, S. Predict then propagate: Graph neural networks meet personalized pagerank. In *International Conference on Learning Representations (ICLR'2019)*, 2019.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

Lee, J. B., Rossi, R. A., Kim, S., Ahmed, N. K., and Koh, E. Attention models in graphs: A survey. *arXiv e-prints*, 2018.

Li, Q., Han, Z., and Wu, X. Deeper insights into graph convolutional networks for semi-supervised learning. *CoRR*, abs/1801.07606, 2018.

Li, Q., Wu, X.-M., Liu, H., Zhang, X., and Guan, Z. Label efficient semi-supervised learning via graph filtering. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

Liao, R., Zhao, Z., Urtasun, R., and Zemel, R. Lanczosnet: Multi-scale deep graph convolutional networks. In *International Conference on Learning Representations (ICLR'2019)*, 2019.

Liu, D. C. and Nocedal, J. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1-3):503–528, 1989.

Miller, G. A. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

Monti, F., Boscaini, D., Masci, J., Rodolà, E., Svoboda, J., and Bronstein, M. M. Geometric deep learning on graphs and manifolds using mixture model cnns. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pp. 5425–5434, 2017.

Perozzi, B., Al-Rfou, R., and Skiena, S. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD'14, pp. 701–710. ACM, 2014.

Rahimi, S., Cohn, T., and Baldwin, T. Semi-supervised user geolocation via graph convolutional networks. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2009–2019. Association for Computational Linguistics, 2018.

Ramakrishnan, R., Hartmann, M., Tapavicza, E., and von Lilienfeld, O. A. Electronic spectra from TDDFT and machine learning in chemical space. *The Journal of chemical physics*, 143(8):084111, 2015.

Roller, S., Speriosu, M., Rallapalli, S., Wing, B., and Baldridge, J. Supervised text-based geolocation using language models on an adaptive grid. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pp. 1500–1510. Association for Computational Linguistics, 2012.

Rosenblatt, F. The Perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Rosenblatt, F. Principles of neurodynamics: Perceptrons and the theory of brain mechanisms. Technical report, Cornell Aeronautical Lab Inc, 1961.

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.

Sobel, I. and Feldman, G. A 3x3 isotropic gradient operator for image processing. *A talk at the Stanford Artificial Project*, pp. 271–272, 1968.

Thekumparampil, K. K., Wang, C., Oh, S., and Li, L. Attention-based graph neural network for semi-supervised learning. *arXiv e-prints*, 2018.

Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph Attention Networks. In *International Conference on Learning Representations (ICLR'2018)*, 2018.

Velikovi, P., Fedus, W., Hamilton, W. L., Li, P., Bengio, Y., and Hjelm, R. D. Deep Graph InfoMax. In *International Conference on Learning Representations (ICLR'2019)*, 2019.

Waibel, A., Hanazawa, T., and Hinton, G. Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech. and Signal Processing*, 37(3), 1989.

Wang, X., Ye, Y., and Gupta, A. Zero-shot recognition via semantic embeddings and knowledge graphs. In *Computer Vision and Pattern Recognition (CVPR)*, pp. 6857–6866. IEEE Computer Society, 2018.

Weston, J., Ratle, F., and Collobert, R. Deep learning via semi-supervised embedding. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pp. 1168–1175, 2008.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Yu, P. S. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR'2019)*, 2019.

Yanardag, P. and Vishwanathan, S. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1365–1374. ACM, 2015.

Yang, Z., Cohen, W. W., and Salakhutdinov, R. Revisiting semi-supervised learning with graph embeddings. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pp. 40–48, 2016.

Yao, L., Mao, C., and Luo, Y. Graph convolutional networks for text classification. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI'19)*, 2019.

Zhang, J., Shi, X., Xie, J., Ma, H., King, I., and Yeung, D. Gaan: Gated attention networks for learning on large and spatiotemporal graphs. In *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence (UAI'2018)*, pp. 339–349. AUAI Press, 2018a.

Zhang, M., Cui, Z., Neumann, M., and Chen, Y. An end-to-end deep learning architecture for graph classification. 2018b.

Zhang, Y., Zhong, V., Chen, D., Angeli, G., and Manning, C. D. Position-aware attention and supervised data improve slot filling. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pp. 35–45. Association for Computational Linguistics, 2017.

Zhang, Y., Qi, P., and Manning, C. D. Graph convolution over pruned dependency trees improves relation extraction. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2018c.

Zhou, D., Bousquet, O., Thomas, N. L., Weston, J., and Schölkopf, B. Learning with local and global consistency. In Thrun, S., Saul, L. K., and Schölkopf, B. (eds.), *Advances in Neural Information Processing Systems 16*, pp. 321–328. MIT Press, 2004.

Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., and Sun, M. Graph Neural Networks: A Review of Methods and Applications. *arXiv e-prints*, 2018.

Zhu, X., Ghahramani, Z., and Lafferty, J. Semi-supervised learning using gaussian fields and harmonic functions. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, pp. 912–919. AAAI Press, 2003.

FASTGCN: FAST LEARNING WITH GRAPH CONVOLUTIONAL NETWORKS VIA IMPORTANCE SAMPLING

Jie Chen*, **Tengfei Ma***, **Cao Xiao**
IBM Research
chenjie@us.ibm.com, Tengfei.Ma1@ibm.com, cxiao@us.ibm.com

ABSTRACT

The graph convolutional networks (GCN) recently proposed by Kipf and Welling are an effective graph model for semi-supervised learning. This model, however, was originally designed to be learned with the presence of both training and test data. Moreover, the recursive neighborhood expansion across layers poses time and memory challenges for training with large, dense graphs. To relax the requirement of simultaneous availability of test data, we interpret graph convolutions as integral transforms of embedding functions under probability measures. Such an interpretation allows for the use of Monte Carlo approaches to consistently estimate the integrals, which in turn leads to a batched training scheme as we propose in this work—FastGCN. Enhanced with importance sampling, FastGCN not only is efficient for training but also generalizes well for inference. We show a comprehensive set of experiments to demonstrate its effectiveness compared with GCN and related models. In particular, training is orders of magnitude more efficient while predictions remain comparably accurate.

1 INTRODUCTION

Graphs are universal representations of pairwise relationship. Many real world data come naturally in the form of graphs; e.g., social networks, gene expression networks, and knowledge graphs. To improve the performance of graph-based learning tasks, such as node classification and link prediction, recently much effort is made to extend well-established network architectures, including recurrent neural networks (RNN) and convolutional neural networks (CNN), to graph data; see, e.g., Bruna et al. (2013); Duvenaud et al. (2015); Li et al. (2015); Jain et al. (2015); Henaff et al. (2015); Niepert et al. (2016); Kipf & Welling (2016a b).

Whereas learning feature representations for graphs is an important subject among this effort, here, we focus on the feature representations for graph vertices. In this vein, the closest work that applies a convolution architecture is the graph convolutional network (GCN) (Kipf & Welling, 2016a b). Borrowing the concept of a convolution filter for image pixels or a linear array of signals, GCN uses the connectivity structure of the graph as the filter to perform neighborhood mixing. The architecture may be elegantly summarized by the following expression:

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

where $\hat{A}$ is some normalization of the graph adjacency matrix, $H^{(l)}$ contains the embedding (row-wise) of the graph vertices in the lth layer, $W^{(l)}$ is a parameter matrix, and σ is nonlinearity.

As with many graph algorithms, the adjacency matrix encodes the pairwise relationship for both training and test data. The learning of the model as well as the embedding is performed for both data simultaneously, at least as the authors proposed. For many applications, however, test data may not be readily available, because the graph may be constantly expanding with new vertices (e.g. new members of a social network, new products to a recommender system, and new drugs for functionality tests). Such scenarios require an inductive scheme that learns a model from only a training set of vertices and that generalizes well to any augmentation of the graph.

*These two authors contribute equally.

A more severe challenge for GCN is that the recursive expansion of neighborhoods across layers incurs expensive computations in batched training. Particularly for dense graphs and powerlaw graphs, the expansion of the neighborhood for a single vertex quickly fills up a large portion of the graph. Then, a usual mini-batch training will involve a large amount of data for every batch, even with a small batch size. Hence, scalability is a pressing issue to resolve for GCN to be applicable to large, dense graphs.

To address both challenges, we propose to view graph convolutions from a different angle and interpret them as integral transforms of embedding functions under probability measures. Such a view provides a principled mechanism for inductive learning, starting from the formulation of the loss to the stochastic version of the gradient. Specifically, we interpret that graph vertices are iid samples of some probability distribution and write the loss and each convolution layer as integrals with respect to vertex embedding functions. Then, the integrals are evaluated through Monte Carlo approximation that defines the sample loss and the sample gradient. One may further alter the sampling distribution (as in importance sampling) to reduce the approximation variance.

The proposed approach, coined FastGCN, not only rids the reliance on the test data but also yields a controllable cost for per-batch computation. At the time of writing, we notice a newly published work GraphSAGE (Hamilton et al., 2017) that proposes also the use of sampling to reduce the computational footprint of GCN. Our sampling scheme is more economic, resulting in a substantial saving in the gradient computation, as will be analyzed in more detail in Section 3.3. Experimental results in Section 4 indicate that the per-batch computation of FastGCN is more than an order of magnitude faster than that of GraphSAGE, while classification accuracies are highly comparable.

2 Related Work

Over the past few years, several graph-based convolution network models emerged for addressing applications of graph-structured data, such as the representation of molecules (Duvenaud et al., 2015). An important stream of work is built on spectral graph theory (Bruna et al., 2013; Henaff et al., 2015; Defferrard et al., 2016). They define parameterized filters in the spectral domain, inspired by graph Fourier transform. These approaches learn a feature representation for the whole graph and may be used for graph classification.

Another line of work learns embeddings for graph vertices, for which Goyal & Ferrara (2017) is a recent survey that covers comprehensively several categories of methods. A major category consists of factorization based algorithms that yield the embedding through matrix factorizations; see, e.g., Roweis & Saul (2000); Belkin & Niyogi (2001); Ahmed et al. (2013); Cao et al. (2015); Ou et al. (2016). These methods learn the representations of training and test data jointly. Another category is random walk based methods (Perozzi et al., 2014; Grover & Leskovec, 2016) that compute node representations through exploration of neighborhoods. LINE (Tang et al., 2015) is also such a technique that is motivated by the preservation of the first and second-order proximities. Meanwhile, there appear a few deep neural network architectures, which better capture the nonlinearity within graphs, such as SDNE (Wang et al., 2016). As motivated earlier, GCN (Kipf & Welling, 2016a) is the model on which our work is based.

The most relevant work to our approach is GraphSAGE (Hamilton et al., 2017), which learns node representations through aggregation of neighborhood information. One of the proposed aggregators employs the GCN architecture. The authors also acknowledge the memory bottleneck of GCN and hence propose an ad hoc sampling scheme to restrict the neighborhood size. Our sampling approach is based on a different and more principled formulation. The major distinction is that we sample vertices rather than neighbors. The resulting computational savings are analyzed in Section 3.3.

3 Training and Inference through Sampling

One striking difference between GCN and many standard neural network architectures is the lack of independence in the sample loss. Training algorithms such as SGD and its batch generalization are designed based on the additive nature of the loss function with respect to independent data samples. For graphs, on the other hand, each vertex is convolved with all its neighbors and hence defining a sample gradient that is efficient to compute is beyond straightforward.

Concretely, consider the standard SGD scenario where the loss is the expectation of some function g with respect to a data distribution D:

$$L = \mathrm{E}_{x\sim D}[g(W;x)].$$

Here, W denotes the model parameter to be optimized. Of course, the data distribution is generally unknown and one instead minimizes the empirical loss through accessing n iid samples $x_1, \ldots, x_n$:

$$L_{\text{emp}} = \frac{1}{n}\sum_{i=1}^{n} g(W;x_i), \qquad x_i \sim D,\ \forall\, i.$$

In each step of SGD, the gradient is approximated by $\nabla g(W;x_i)$, an (assumed) unbiased sample of ∇L. One may interpret that each gradient step makes progress toward the sample loss $g(W;x_i)$. The sample loss and the sample gradient involve only one single sample x_i.

For graphs, one may no longer leverage the independence and compute the sample gradient $\nabla g(W;x_i)$ by discarding the information of i's neighboring vertices and their neighbors, recursively. We therefore seek an alternative formulation. In order to cast the learning problem under the same sampling framework, let us assume that there is a (possibly infinite) graph G' with the vertex set V' associated with a probability space (V', F, P), such that for the given graph G, it is an induced subgraph of G' and its vertices are iid samples of V' according to the probability measure P. For the probability space, V' serves as the sample space and F may be any event space (e.g., the power set $F = 2^{V'}$). The probability measure P defines a sampling distribution.

To resolve the problem of lack of independence caused by convolution, we interpret that each layer of the network defines an embedding function of the vertices (random variable) that are tied to the same probability measure but are independent. See Figure 1. Specifically, recall the architecture of GCN

$$\tilde{H}^{(l+1)} = \hat{A}H^{(l)}W^{(l)}, \quad H^{(l+1)} = \sigma(\tilde{H}^{(l+1)}), \quad l = 0, \ldots, M-1, \quad L = \frac{1}{n}\sum_{i=1}^{n} g(H^{(M)}(i,:)). \tag{1}$$

For the functional generalization, we write

$$\tilde{h}^{(l+1)}(v) = \int \hat{A}(v,u)h^{(l)}(u)W^{(l)}\,dP(u), \quad h^{(l+1)}(v) = \sigma(\tilde{h}^{(l+1)}(v)), \quad l = 0, \ldots, M-1, \tag{2}$$

$$L = \mathrm{E}_{v\sim P}[g(h^{(M)}(v))] = \int g(h^{(M)}(v))\,dP(v). \tag{3}$$

Here, u and v are independent random variables, both of which have the same probability measure P. The function $h^{(l)}$ is interpreted as the embedding function from the lth layer. The embedding functions from two consecutive layers are related through convolution, expressed as an integral transform, where the kernel $\hat{A}(v,u)$ corresponds to the (v,u) element of the matrix $\hat{A}$. The loss is the expectation of $g(h^{(M)})$ for the final embedding $h^{(M)}$. Note that the integrals are not the usual Riemann–Stieltjes integrals, because the variables u and v are graph vertices but not real numbers; however, this distinction is only a matter of formalism.

Writing GCN in the functional form allows for evaluating the integrals in the Monte Carlo manner, which leads to a batched training algorithm and also to a natural separation of training and test data, as in inductive learning. For each layer l, we use t_l iid samples $u_1^{(l)}, \ldots, u_{t_l}^{(l)} \sim P$ to approximately evaluate the integral transform (2); that is,

$$\tilde{h}_{t_{l+1}}^{(l+1)}(v) := \frac{1}{t_l}\sum_{j=1}^{t_l} \hat{A}(v, u_j^{(l)})h_{t_l}^{(l)}(u_j^{(l)})W^{(l)}, \quad h_{t_{l+1}}^{(l+1)}(v) := \sigma(\tilde{h}_{t_{l+1}}^{(l+1)}(v)), \quad l = 0, \ldots, M-1,$$

with the convention $h_{t_0}^{(0)} \equiv h^{(0)}$. Then, the loss L in (3) admits an estimator

$$L_{t_0,t_1,\ldots,t_M} := \frac{1}{t_M}\sum_{i=1}^{t_M} g(h_{t_M}^{(M)}(u_i^{(M)})).$$

The follow result establishes that the estimator is consistent. The proof is a recursive application of the law of large numbers and the continuous mapping theorem; it is given in the appendix.

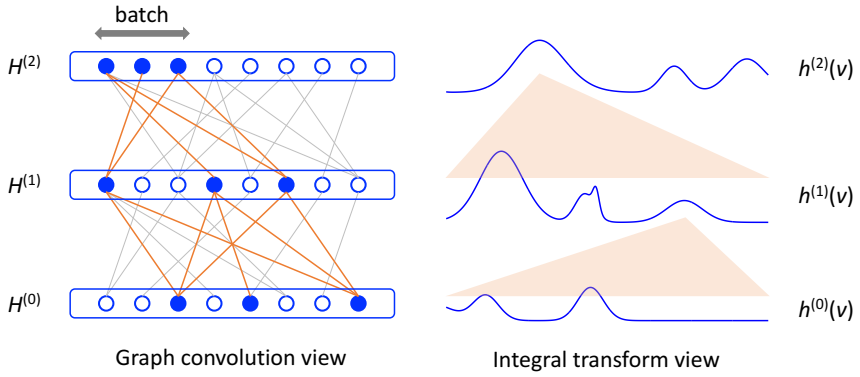


Figure 1: Two views of GCN. On the left (graph convolution view), each circle represents a graph vertex. On two consecutive rows, a circle i is connected (in gray line) with circle j if the two corresponding vertices in the graph are connected. A convolution layer uses the graph connectivity structure to mix the vertex features/embeddings. On the right (integral transform view), the embedding function in the next layer is an integral transform (illustrated by the orange fanout shape) of the one in the previous layer. For the proposed method, all integrals (including the loss function) are evaluated by using Monte Carlo sampling. Correspondingly in the graph view, vertices are subsampled in a bootstrapping manner in each layer to approximate the convolution. The sampled portions are collectively denoted by the solid blue circles and the orange lines.

Theorem 1. *If g and σ are continuous, then*

$$\lim_{t_0,t_1,\ldots,t_M\to\infty} L_{t_0,t_1,\ldots,t_M} = L \quad \textit{with probability one.}$$

In practical use, we are given a graph whose vertices are already assumed to be samples. Hence, we will need bootstrapping to obtain a consistent estimate. In particular, for the network architecture (1), the output $H^{(M)}$ is split into batches as usual. We will still use $u_1^{(M)}, \ldots, u_{t_M}^{(M)}$ to denote a batch of vertices, which come from the given graph. For each batch, we sample (with replacement) uniformly each layer and obtain samples $u_i^{(l)}$, $i = 1, \ldots, t_l$, $l = 0, \ldots, M-1$. Such a procedure is equivalent to uniformly sampling the rows of $H^{(l)}$ for each l. Then, we obtain the batch loss

$$L_{\text{batch}} = \frac{1}{t_M} \sum_{i=1}^{t_M} g(H^{(M)}(u_i^{(M)}, :)), \tag{4}$$

where, recursively,

$$H^{(l+1)}(v, :) = \sigma\left(\frac{n}{t_l} \sum_{j=1}^{t_l} \hat{A}(v, u_j^{(l)}) H^{(l)}(u_j^{(l)}, :) W^{(l)} \right), \quad l = 0, \ldots, M-1. \tag{5}$$

Here, the n inside the activation function σ is the number of vertices in the given graph and is used to account for the normalization difference between the matrix form (1) and the integral form (2). The corresponding batch gradient may be straightforwardly obtained through applying the chain rule on each $H^{(l)}$. See Algorithm 1.

3.1 Variance Reduction

As for any estimator, one is interested in improving its variance. Whereas computing the full variance is highly challenging because of nonlinearity in all the layers, it is possible to consider each single layer and aim at improving the variance of the embedding function before nonlinearity. Specifically, consider for the lth layer, the function $\tilde{h}_{t_{l+1}}^{(l+1)}(v)$ as an approximation to the convolution $\int \hat{A}(v, u) h_{t_l}^{(l)}(u) W^{(l)} \, dP(u)$. When taking t_{l+1} samples $v = u_1^{(l+1)}, \ldots, u_{t_{l+1}}^{(l+1)}$, the sample average of $\tilde{h}_{t_{l+1}}^{(l+1)}(v)$ admits a variance that captures the deviation from the eventual loss contributed by this layer. Hence, we seek an improvement of this variance. Now that we consider each layer separately, we will do the following change of notation to keep the expressions less cumbersome:

Algorithm 1 FastGCN batched training (one epoch)

for each batch **do**
  For each layer l, sample uniformly t_l vertices $u_1^{(l)}, \dots, u_{t_l}^{(l)}$
  for each layer l **do** ▷ Compute batch gradient ∇L_{batch}
    If v is sampled in the next layer,

$$\nabla \tilde{H}^{(l+1)}(v,:) \leftarrow \frac{n}{t_l} \sum_{j=1}^{t_l} \hat{A}(v, u_j^{(l)}) \nabla \left\{ H^{(l)}(u_j^{(l)},:) W^{(l)} \right\}$$

  end for
  $W \leftarrow W - \eta \nabla L_{\text{batch}}$ ▷ SGD step
end for

	Function	Samples	Num. samples
Layer $l+1$; random variable v	$\tilde{h}_{t_{l+1}}^{(l+1)}(v) \to y(v)$	$u_i^{(l+1)} \to v_i$	$t_{l+1} \to s$
Layer l; random variable u	$h_{t_l}^{(l)}(u) W^{(l)} \to x(u)$	$u_j^{(l)} \to u_j$	$t_l \to t$

Under the joint distribution of v and u, the aforementioned sample average is

$$G := \frac{1}{s} \sum_{i=1}^{s} y(v_i) = \frac{1}{s} \sum_{i=1}^{s} \left(\frac{1}{t} \sum_{j=1}^{t} \hat{A}(v_i, u_j) x(u_j) \right).$$

First, we have the following result.

Proposition 2. *The variance of G admits*

$$\mathrm{Var}\{G\} = R + \frac{1}{st} \iint \hat{A}(v,u)^2 x(u)^2 \, dP(u) \, dP(v), \tag{6}$$

where

$$R = \frac{1}{s} \left(1 - \frac{1}{t}\right) \int e(v)^2 \, dP(v) - \frac{1}{s} \left(\int e(v) \, dP(v) \right)^2 \quad \textit{and} \quad e(v) = \int \hat{A}(v,u) x(u) \, dP(u).$$

The variance (6) consists of two parts. The first part R leaves little room for improvement, because the sampling in the v space is not done in this layer. The second part (the double integral), on the other hand, depends on how the u_j's in this layer are sampled. The current result (6) is the consequence of sampling u_j's by using the probability measure P. One may perform importance sampling, altering the sampling distribution to reduce variance. Specifically, let $Q(u)$ be the new probability measure, where the u_j's are drawn from. We hence define the new sample average approximation

$$y_Q(v) := \frac{1}{t} \sum_{j=1}^{t} \hat{A}(v, u_j) x(u_j) \left(\left. \frac{dP(u)}{dQ(u)} \right|_{u_j} \right), \qquad u_1, \dots, u_t \sim Q,$$

and the quantity of interest

$$G_Q := \frac{1}{s} \sum_{i=1}^{s} y_Q(v_i) = \frac{1}{s} \sum_{i=1}^{s} \left(\frac{1}{t} \sum_{j=1}^{t} \hat{A}(v_i, u_j) x(u_j) \left(\left. \frac{dP(u)}{dQ(u)} \right|_{u_j} \right) \right).$$

Clearly, the expectation of G_Q is the same as that of G, regardless of the new measure Q. The following result gives the optimal Q.

Theorem 3. *If*

$$dQ(u) = \frac{b(u)|x(u)| \, dP(u)}{\int b(u)|x(u)| \, dP(u)} \quad \textit{where} \quad b(u) = \left[\int \hat{A}(v,u)^2 \, dP(v) \right]^{\frac{1}{2}}, \tag{7}$$

then the variance of G_Q admits

$$\mathrm{Var}\{G_Q\} = R + \frac{1}{st}\left[\int b(u)|x(u)|\,dP(u)\right]^2, \tag{8}$$

where R is defined in Proposition 2. The variance is minimum among all choices of Q.

A drawback of defining the sampling distribution Q in this manner is that it involves $|x(u)|$, which constantly changes during training. It corresponds to the product of the embedding matrix $H^{(l)}$ and the parameter matrix $W^{(l)}$. The parameter matrix is updated in every iteration; and the matrix product is expensive to compute. Hence, the cost of computing the optimal measure Q is quite high.

As a compromise, we consider a different choice of Q, which involves only $b(u)$. The following proposition gives the precise definition. The resulting variance may or may not be smaller than (6). In practice, however, we find that it is almost always helpful.

Proposition 4. *If*

$$dQ(u) = \frac{b(u)^2\,dP(u)}{\int b(u)^2\,dP(u)}$$

where $b(u)$ is defined in (7), then the variance of G_Q admits

$$\mathrm{Var}\{G_Q\} = R + \frac{1}{st}\int b(u)^2\,dP(u)\int x(u)^2\,dP(u), \tag{9}$$

where R is defined in Proposition 2.

With this choice of the probability measure Q, the ratio $dQ(u)/dP(u)$ is proportional to $b(u)^2$, which is simply the integral of $\hat{A}(v,u)^2$ with respect to v. In practical use, for the network architecture (1), we define a probability mass function for all the vertices in the given graph:

$$q(u) = \|\hat{A}(:,u)\|^2 / \sum_{u'\in V}\|\hat{A}(:,u')\|^2, \quad u \in V$$

and sample t vertices $u_1,\dots,u_t$ according to this distribution. From the expression of q, we see that it has no dependency on l; that is, the sampling distribution is the same for all layers. To summarize, the batch loss L_{batch} in (4) now is recursively expanded as

$$H^{(l+1)}(v,:) = \sigma\left(\frac{1}{t_l}\sum_{j=1}^{t_l}\frac{\hat{A}(v,u_j^{(l)})H^{(l)}(u_j^{(l)},:)W^{(l)}}{q(u_j^{(l)})}\right), \quad u_j^{(l)} \sim q, \quad l = 0,\dots,M-1. \tag{10}$$

The major difference between (5) and (10) is that the former obtains samples uniformly whereas the latter according to q. Accordingly, the scaling inside the summation changes. The corresponding batch gradient may be straightforwardly obtained through applying the chain rule on each $H^{(l)}$. See Algorithm 2.

Algorithm 2 FastGCN batched training (one epoch), improved version

1: For each vertex u, compute sampling probability $q(u) \propto \|\hat{A}(:,u)\|^2$
2: **for** each batch **do**
3: For each layer l, sample t_l vertices $u_1^{(l)},\dots,u_{t_l}^{(l)}$ according to distribution q
4: **for** each layer l **do** ▷ Compute batch gradient ∇L_{batch}
5: If v is sampled in the next layer,

$$\nabla\tilde{H}^{(l+1)}(v,:) \leftarrow \frac{1}{t_l}\sum_{j=1}^{t_l}\frac{\hat{A}(v,u_j^{(l)})}{q(u_j^{(l)})}\nabla\left\{H^{(l)}(u_j^{(l)},:)W^{(l)}\right\}$$

6: **end for**
7: $W \leftarrow W - \eta\nabla L_{\text{batch}}$ ▷ SGD step
8: **end for**

3.2 INFERENCE

The sampling approach described in the preceding subsection clearly separates out test data from training. Such an approach is inductive, as opposed to transductive that is common for many graph algorithms. The essence is to cast the set of graph vertices as iid samples of a probability distribution, so that the learning algorithm may use the gradient of a consistent estimator of the loss to perform parameter update. Then, for inference, the embedding of a new vertex may be either computed by using the full GCN architecture (1), or approximated through sampling as is done in parameter learning. Generally, using the full architecture is more straightforward and easier to implement.

3.3 COMPARISON WITH GRAPHSAGE

GraphSAGE (Hamilton et al., 2017) is a newly proposed architecture for generating vertex embeddings through aggregating neighborhood information. It shares the same memory bottleneck with GCN, caused by recursive neighborhood expansion. To reduce the computational footprint, the authors propose restricting the immediate neighborhood size for each layer. Using our notation for the sample size, if one samples t_l neighbors for each vertex in the lth layer, then the size of the expanded neighborhood is, in the worst case, the product of the t_l's. On the other hand, FastGCN samples vertices rather than neighbors in each layer. Then, the total number of involved vertices is at most the sum of the t_l's, rather than the product. See experimental results in Section 4 for the order-of-magnitude saving in actual computation time.

4 EXPERIMENTS

We follow the experiment setup in Kipf & Welling (2016a) and Hamilton et al. (2017) to demonstrate the effective use of FastGCN, comparing with the original GCN model as well as GraphSAGE, on the following benchmark tasks: (1) classifying research topics using the Cora citation data set (McCallum et al., 2000); (2) categorizing academic papers with the Pubmed database; and (3) predicting the community structure of a social network modeled with Reddit posts. These data sets are downloaded from the accompany websites of the aforementioned references. The graphs have increasingly more nodes and higher node degrees, representative of the large and dense setting under which our method is motivated. Statistics are summarized in Table 1. We adjusted the training/validation/test split of Cora and Pubmed to align with the supervised learning scenario. Specifically, all labels of the training examples are used for training, as opposed to only a small portion in the semi-supervised setting (Kipf & Welling, 2016a). Such a split is coherent with that of the other data set, Reddit, used in the work of GraphSAGE. Additional experiments using the original split of Cora and Pubmed are reported in the appendix.

Table 1: Dataset Statistics

Dataset	Nodes	Edges	Classes	Features	Training/Validation/Test
Cora	$2,708$	$5,429$	7	$1,433$	$1,208/500/1,000$
Pubmed	$19,717$	$44,338$	3	500	$18,217/500/1,000$
Reddit	$232,965$	$11,606,919$	41	602	$152,410/23,699/55,334$

Implementation details are as following. All networks (including those under comparison) contain two layers as usual. The codes of GraphSAGE and GCN are downloaded from the accompany websites and the latter is adapted for FastGCN. Inference with FastGCN is done with the full GCN network, as mentioned in Section 3.2. Further details are contained in the appendix.

We first consider the use of sampling in FastGCN. The left part of Table 2 (columns under "Sampling") lists the time and classification accuracy as the number of samples increases. For illustration purpose, we equalize the sample size on both layers. Clearly, with more samples, the per-epoch training time increases, but the accuracy (as measured by using micro F1 scores) also improves generally.

An interesting observation is that given input features $H^{(0)}$, the product $\hat{A}H^{(0)}$ in the bottom layer does not change, which means that the chained expansion of the gradient with respect to $W^{(0)}$ in

Table 2: Benefit of precomputing $\hat{A}H^{(0)}$ for the input layer. Data set: Pubmed. Training time is in seconds, per-epoch (batch size 1024). Accuracy is measured by using micro F1 score.

	Sampling		Precompute	
t_1	Time	F1	Time	F1
5	0.737	0.859	0.139	0.849
10	0.755	0.863	0.141	0.870
25	0.760	0.873	0.144	0.879
50	0.774	0.864	0.142	0.880

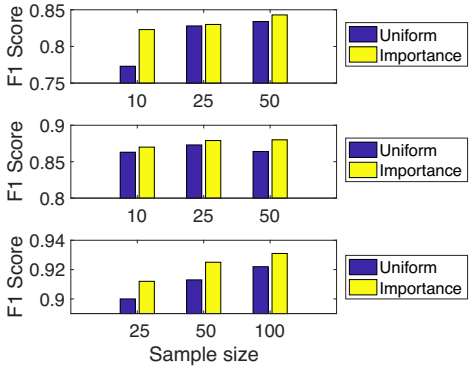
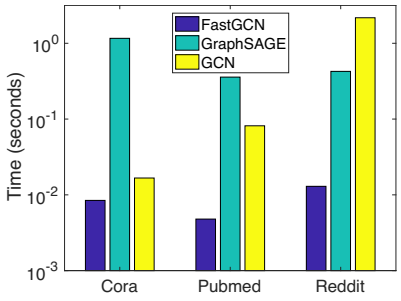


Figure 2: Prediction accuracy: uniform versus importance sampling. The three data sets from top to bottom are ordered the same as Table 1.

the last step is a constant throughout training. Hence, one may precompute the product rather than sampling this layer to gain efficiency. The compared results are listed on the right part of Table 2 (columns under "Precompute"). One sees that the training time substantially decreases while the accuracy is comparable. Hence, all the experiments that follow use precomputation.

Next, we compare the sampling approaches for FastGCN: uniform and importance sampling. Figure 2 summarizes the prediction accuracy under both approaches. It shows that importance sampling consistently yields higher accuracy than does uniform sampling. Since the altered sampling distribution (see Proposition 4 and Algorithm 2) is a compromise alternative of the optimal distribution that is impractical to use, this result suggests that the variance of the used sampling indeed is smaller than that of uniform sampling; i.e., the term (9) stays closer to (8) than does (6). A possible reason is that $b(u)$ correlates with $|x(u)|$. Hence, later experiments will apply importance sampling.

We now demonstrate that the proposed method is significantly faster than the original GCN as well as GraphSAGE, while maintaining comparable prediction performance. See Figure 3. The bar heights indicate the per-batch training time, in the log scale. One sees that GraphSAGE is a substantial improvement of GCN for large and dense graphs (e.g., Reddit), although for smaller ones (Cora and Pubmed), GCN trains faster. FastGCN is the fastest, with at least an order of magnitude improvement compared with the runner up (except for Cora), and approximately two orders of magnitude speed up compared with the slowest. Here, the training time of FastGCN is with respect to the sample size that achieves the best prediction accuracy. As seen from the table on the right, this accuracy is highly comparable with the best of the other two methods.



Micro F1 Score

	Cora	Pubmed	Reddit
FastGCN	0.850	0.880	0.937
GraphSAGE-GCN	0.829	0.849	0.923
GraphSAGE-mean	0.822	0.888	0.946
GCN (batched)	0.851	0.867	0.930
GCN (original)	0.865	0.875	NA

Figure 3: Per-batch training time in seconds (left) and prediction accuracy (right). For timing, GraphSAGE refers to GraphSAGE-GCN in Hamilton et al. (2017). The timings of using other aggregators, such as GraphSAGE-mean, are similar. GCN refers to using batched learning, as opposed to the original version that is nonbatched; for more details of the implementation, see the appendix. The nonbatched version of GCN runs out of memory on the large graph Reddit. The sample sizes for FastGCN are 400, 100, and 400, respectively for the three data sets.

In the discussion period, the authors of GraphSAGE offered an improved implementation of their codes and alerted that GraphSAGE was better suited for massive graphs. The reason is that for small graphs, the sample size (recalling that it is the product across layers) is comparable to the graph size and hence improvement is marginal; moreover, sampling overhead might then adversely affect the timing. For fair comparison, the authors of GraphSAGE kept the sampling strategy but improved the implementation of their original codes by eliminating redundant calculations of the sampled nodes. Now the per-batch training time of GraphSAGE compares more favorably on the smallest graph Cora; see Table 3. Note that this implementation does not affect large graphs (e.g., Reddit) and our observation of orders of magnitude faster training remains valid.

Table 3: Further comparison of per-batch training time (in seconds) with new implementation of GraphSAGE for small graphs. The new implementation is in PyTorch whereas the rest are in TensorFlow.

	Cora	Pubmed	Reddit
FastGCN	0.0084	0.0047	0.0129
GraphSAGE-GCN (old impl)	1.1630	0.3579	0.4260
GraphSAGE-GCN (new impl)	0.0380	0.3989	NA
GCN (batched)	0.0166	0.0815	2.1731

5 Conclusions

We have presented FastGCN, a fast improvement of the GCN model recently proposed by Kipf & Welling (2016a) for learning graph embeddings. It generalizes transductive training to an inductive manner and also addresses the memory bottleneck issue of GCN caused by recursive expansion of neighborhoods. The crucial ingredient is a sampling scheme in the reformulation of the loss and the gradient, well justified through an alternative view of graph convoluntions in the form of integral transforms of embedding functions. We have compared the proposed method with additionally GraphSAGE (Hamilton et al., 2017), a newly published work that also proposes using sampling to restrict the neighborhood size, although the two sampling schemes substantially differ in both algorithm and cost. Experimental results indicate that our approach is orders of magnitude faster than GCN and GraphSAGE, while maintaining highly comparable prediction performance with the two.

The simplicity of the GCN architecture allows for a natural interpretation of graph convolutions in terms of integral transforms. Such a view, yet, generalizes to many graph models whose formulations are based on first-order neighborhoods, examples of which include MoNet that applies to (meshed) manifolds (Monti et al., 2017), as well as many message-passing neural networks (see e.g., Scarselli et al. (2009); Gilmer et al. (2017)). The proposed work elucidates the basic Monte Carlo ingredients for consistently estimating the integrals. When generalizing to other networks aforementioned, an additional effort is to investigate whether and how variance reduction may improve the estimator, a possibly rewarding avenue of future research.

References

Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy, Vanja Josifovski, and Alexander J. Smola. Distributed large-scale natural graph factorization. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pp. 37–48, 2013. ISBN 978-1-4503-2035-1.

Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, pp. 585–591, 2001.

Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *CoRR*, abs/1312.6203, 2013.

Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, pp. 891–900, 2015. ISBN 978-1-4503-3794-6.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *CoRR*, abs/1606.09375, 2016.

David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alan Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 28*, pp. 2224–2232. Curran Associates, Inc., 2015.

J. Gilmer, S.S. Schoenholz, P.F. Riley, O. Vinyals, and G.E. Dahl. Neural message passing for quantum chemistry. In *ICML*, 2017.

Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *CoRR*, abs/1705.02801, 2017.

Aditya Grover and Jure Leskovec. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pp. 855–864, 2016. ISBN 978-1-4503-4232-2.

William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *CoRR*, abs/1706.02216, 2017.

Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. *CoRR*, abs/1506.05163, 2015.

Ashesh Jain, Amir Roshan Zamir, Silvio Savarese, and Ashutosh Saxena. Structural-rnn: Deep learning on spatio-temporal graphs. *CoRR*, abs/1511.05298, 2015.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016a.

TN. Kipf and M. Welling. Variational graph auto-encoders. In *NIPS Workshop on Bayesian Deep Learning*. 2016b.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. *CoRR*, abs/1511.05493, 2015.

Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the construction of internet portals with machine learning. *Inf. Retr.*, 3(2):127–163, July 2000. ISSN 1386-4564.

F. Monti, D. Boscaini, J. Masci, E. Rodala, J. Svoboda, and M.M. Bronstein. Geometric deep learning on graphs and manifolds using mixture model CNNs. In *CVPR*, 2017.

Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. *CoRR*, abs/1605.05273, 2016.

Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pp. 1105–1114, 2016. ISBN 978-1-4503-4232-2.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pp. 701–710, 2014. ISBN 978-1-4503-2956-9.

Sam T. Roweis and Lawrence K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, 2000. ISSN 0036-8075. doi: 10.1126/science.290.5500.2323.

F. Scarselli, M. Gori, A.C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20, 2009.

Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pp. 1067–1077, 2015. ISBN 978-1-4503-3469-3.

Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pp. 1225–1234, 2016. ISBN 978-1-4503-4232-2.

A PROOFS

Proof of Theorem 1. Because the samples $u_j^{(0)}$ are iid, by the strong law of large numbers,

$$\tilde{h}_{t_1}^{(1)}(v) = \frac{1}{t_0}\sum_{j=1}^{t_0} \hat{A}(v, u_j^{(0)}) h^{(0)}(u_j^{(0)}) W^{(0)}$$

converges almost surely to $\tilde{h}^{(1)}(v)$. Then, because the activation function σ is continuous, the continuous mapping theorem implies that $h_{t_1}^{(1)}(v) = \sigma(\tilde{h}_{t_1}^{(1)}(v))$ converges almost surely to $h^{(1)}(v) = \sigma(\tilde{h}^{(1)}(v))$. Thus, $\int \hat{A}(v,u) h_{t_1}^{(1)}(u) W^{(1)}\, dP(u)$ converges almost surely to $\tilde{h}^{(2)}(v) = \int \hat{A}(v,u) h^{(1)}(u) W^{(1)}\, dP(u)$, where note that the probability space is with respect to the 0th layer and hence has nothing to do with that of the variable u or v in this statement. Similarly,

$$\tilde{h}_{t_2}^{(2)}(v) = \frac{1}{t_1}\sum_{j=1}^{t_1} \hat{A}(v, u_j^{(1)}) h_{t_1}^{(1)}(u_j^{(1)}) W^{(1)}$$

converges almost surely to $\int \hat{A}(v,u) h_{t_1}^{(1)}(u) W^{(1)}\, dP(u)$ and thus to $\tilde{h}^{(2)}(v)$. A simple induction completes the rest of the proof. □

Proof of Proposition 2. Conditioned on v, the expectation of $y(v)$ is

$$\mathrm{E}[y(v)|v] = \int \hat{A}(v,u) x(u)\, dP(u) = e(v), \tag{11}$$

and the variance is $1/t$ times that of $\hat{A}(v,u)x(u)$, i.e.,

$$\mathrm{Var}\{y(v)|v\} = \frac{1}{t}\left(\int \hat{A}(v,u)^2 x(u)^2\, dP(u) - e(v)^2\right). \tag{12}$$

Instantiating (11) and (12) with iid samples $v_1, \ldots, v_s \sim P$ and taking variance and expectation in the front, respectively, we obtain

$$\mathrm{Var}\left\{\mathrm{E}\left[\frac{1}{s}\sum_{i=1}^{s} y(v_i)\middle| v_1, \ldots, v_s\right]\right\} = \mathrm{Var}\left\{\frac{1}{s}\sum_{i=1}^{s} e(v_i)\right\} = \frac{1}{s}\int e(v)^2\, dP(v) - \frac{1}{s}\left(\int e(v)\, dP(v)\right)^2,$$

and

$$\mathrm{E}\left[\mathrm{Var}\left\{\frac{1}{s}\sum_{i=1}^{s} y(v_i)\middle| v_1, \ldots, v_s\right\}\right] = \frac{1}{st}\iint \hat{A}(v,u)^2 x(u)^2\, dP(u)\, dP(v) - \frac{1}{st}\int e(v)^2\, dP(v).$$

Then, applying the law of total variance

$$\mathrm{Var}\left\{\frac{1}{s}\sum_{i=1}^{s} y(v_i)\right\} = \mathrm{Var}\left\{\mathrm{E}\left[\frac{1}{s}\sum_{i=1}^{s} y(v_i)\middle| v_1, \ldots, v_s\right]\right\} + \mathrm{E}\left[\mathrm{Var}\left\{\frac{1}{s}\sum_{i=1}^{s} y(v_i)\middle| v_1, \ldots, v_s\right\}\right],$$

we conclude the proof. □

Proof of Theorem 3. Conditioned on v, the variance of $y_Q(v)$ is $1/t$ times that of

$$\hat{A}(v,u)x(u)\frac{dP(u)}{dQ(u)} \quad (\text{where } u \sim Q),$$

i.e.,

$$\mathrm{Var}\{y_Q(v)|v\} = \frac{1}{t}\left(\int \frac{\hat{A}(v,u)^2 x(u)^2 dP(u)^2}{dQ(u)} - e(v)^2\right).$$

Then, following the proof of Proposition 2, the overall variance is

$$\mathrm{Var}\{G_Q\} = R + \frac{1}{st}\iint \frac{\hat{A}(v,u)^2 x(u)^2\, dP(u)^2\, dP(v)}{dQ(u)} = R + \frac{1}{st}\int \frac{b(u)^2 x(u)^2 dP(u)^2}{dQ(u)}.$$

Hence, the optimal $dQ(u)$ must be proportional to $b(u)|x(u)|\, dP(u)$. Because it also must integrate to unity, we have

$$dQ(u) = \frac{b(u)|x(u)|\, dP(u)}{\int b(u)|x(u)|\, dP(u)},$$

in which case

$$\mathrm{Var}\{G_Q\} = R + \frac{1}{st}\left[\int b(u)|x(u)|\, dP(u)\right]^2.$$

□

Proof of Proposition 4. Conditioned on v, the variance of $y_Q(v)$ is $1/t$ times that of

$$\hat{A}(v,u)x(u)\frac{dP(u)}{dQ(u)} = \frac{\hat{A}(v,u)\,\mathrm{sgn}(x(u))}{b(u)}\int b(u)|x(u)|\, dP(u),$$

i.e.,

$$\mathrm{Var}\{y_Q(v)|v\} = \frac{1}{t}\left(\left[\int b(u)|x(u)|\, dP(u)\right]^2 \int \frac{\hat{A}(v,u)^2}{b(u)^2}\, dQ(u) - e(v)^2\right).$$

The rest of the proof follows that of Proposition 2. □

B Additional Experiment Details

B.1 Baselines

GCN: The original GCN cannot work on very large graphs (e.g., Reddit). So we modified it into a batched version by simply removing the sampling in our FastGCN (i.e., using all the nodes instead of sampling a few in each batch). For relatively small graphs (Cora and Pubmed), we also compared the results with the original GCN.

GraphSAGE: For training time comparison, we use GraphSAGE-GCN that employs GCN as the aggregator. It is also the fastest version among all choices of the aggregators. For accuracy comparison, we also compared with GraphSAGE-mean. We used the codes from `https://github.com/williamleif/GraphSAGE`. Following the setting of Hamilton et al. (2017), we use two layers with neighborhood sample sizes $S_1 = 25$ and $S_2 = 10$. For fair comparison with our method, the batch size is set to be the same as FastGCN, and the hidden dimension is 128.

B.2 Experiment Setup

Datasets: The Cora and Pubmed data sets are from `https://github.com/tkipf/gcn`. As we explained in the paper, we kept the validation index and test index unchanged but changed the training index to use all the remaining nodes in the graph. The Reddit data is from `http://snap.stanford.edu/graphsage/`.

Experiment Setting: We preformed hyperparameter selection for the learning rate and model dimension. We swept learning rate in the set $\{0.01, 0.001, 0.0001\}$. The hidden dimension of FastGCN for Reddit is set as 128, and for the other two data sets, it is 16. The batch size is 256

for Cora and Reddit, and 1024 for Pubmed. Dropout rate is set as 0. We use Adam as the optimization method for training. In the test phase, we use the trained parameters and all the graph nodes instead of sampling. For more details please check our codes in a temporary git repository `https://github.com/matenure/FastGCN`

Hardware: Running time is compared on a single machine with 4-core 2.5 GHz Intel Core i7, and 16G RAM.

C Additional Experiments

C.1 Training Time Comparison

Figure 3 in the main text compares the per-batch training time for different methods. Here, we list the total training time for reference. It is impacted by the convergence of SGD, whose contributing factors include learning rate, batch size, and sample size. See Table 4. Although the orders-of-magnitude speedup of per-batch time is slightly weakened by the convergence speed, one still sees a substantial advantage of the proposed method in the overall training time. Note that even though the original GCN trains faster than the batched version, it does not scale because of memory limitation. Hence, a fair comparison should be gauged with the batched version. We additionally show in Figure 4 the evolution of prediction accuracy as training progresses.

Table 4: Total training time (in seconds).

	Cora	Pubmed	Reddit
FastGCN	2.7	15.5	638.6
GraphSAGE-GCN	72.4	259.6	3318.5
GCN (batched)	6.9	210.8	58346.6
GCN (original)	1.7	21.4	NA

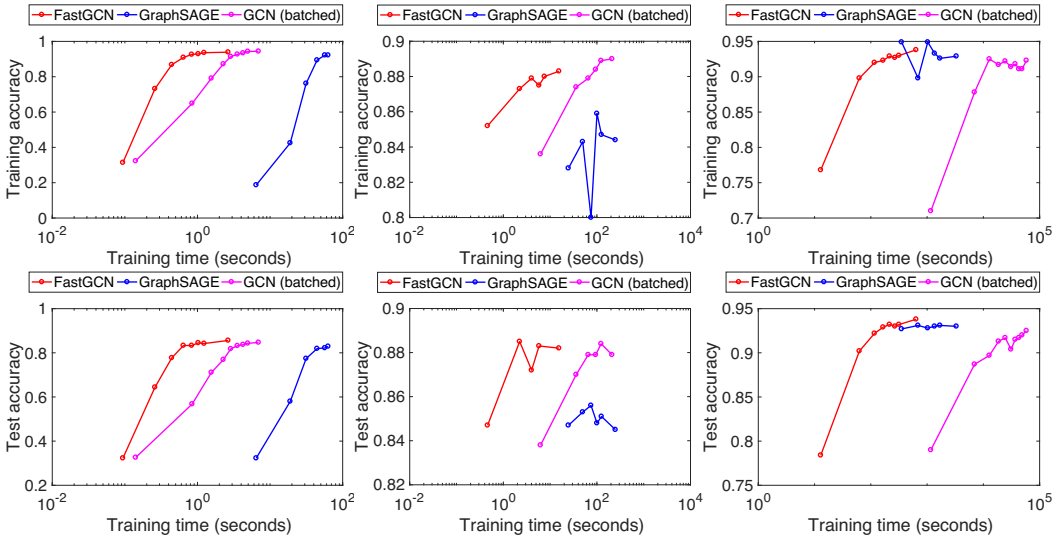


Figure 4: Training/test accuracy versus training time. From left to right, the data sets are Cora, Pubmed, and Reddit, respectively.

C.2 Original Data Split for Cora and Pubmed

As explained in Section 4, we increased the number of labels used for training in Cora and Pubmed, to align with the supervised learning setting of Reddit. For reference, here we present results by using the original data split with substantially fewer training labels. We also fork a separate version of FastGCN, called FastGCN-transductive, that uses both training and test data for learning. See Table 5.

The results for GCN are consistent with those reported by Kipf & Welling (2016a). Because labeled data are scarce, the training of GCN is quite fast. FastGCN beats it only on Pubmed. The accuracy results of FastGCN are inferior to GCN, also because of the limited number of training labels. The transductive version FastGCN-transductive matches the accuracy of that of GCN. The results for GraphSAGE are curious. We suspect that the model significantly overfits the data, because perfect training accuracy (i.e., 1) is attained.

One may note a subtlety that the training of GCN (original) is slower than what is reported in Table 4, even though fewer labels are used here. The reason is that we adopt the same hyperparameters as in Kipf & Welling (2016a) to reproduce the F1 scores of their work, whereas for Table 4, a better learning rate is found that boosts the performance on the new split of the data, in which case GCN (original) converges faster.

Table 5: Total training time and test accuracy for Cora and Pubmed, original data split. Time is in seconds.

	Cora		Pubmed	
	Time	F1	Time	F1
FastGCN	2.52	0.723	0.97	0.721
FastGCN-transductive	5.88	0.818	8.97	0.776
GraphSAGE-GCN	107.95	0.334	39.34	0.386
GCN (original)	2.18	0.814	32.65	0.795

D CONVERGENCE

Strictly speaking, the training algorithms proposed in Section 3 do not precisely follow the existing theory of SGD, because the gradient estimator, though consistent, is biased. In this section, we fill the gap by deriving a convergence result. Similar to the case of standard SGD where the convergence rate depends on the properties of the objective function, here we analyze only a simple case; a comprehensive treatment is out of the scope of the present work. For convenience, we will need a separate system of notations and the same notations appearing in the main text may bear a different meaning here. We abbreviate "with probability one" to "w.p.1" for short.

We use $f(x)$ to denote the objective function and assume that it is differentiable. Differentiability is not a restriction because for the nondifferentiable case, the analysis that follows needs simply change the gradient to the subgradient. The key assumption made on f is that it is l-strictly convex; that is, there exists a positive real number l such that

$$f(x) - f(y) \geq \langle \nabla f(y), x - y \rangle + \frac{l}{2}\|x - y\|^2, \tag{13}$$

for all x and y. We use g to denote the gradient estimator. Specifically, denote by $g(x; \xi_N)$, with ξ_N being a random variable, a strongly consistent estimator of $\nabla f(x)$; that is,

$$\lim_{N\to\infty} g(x; \xi_N) = \nabla f(x) \quad \text{w.p.1.}$$

Moreover, we consider the SGD update rule

$$x_{k+1} = x_k - \gamma_k \; g(x_k; \xi_N^{(k)}), \tag{14}$$

where $\xi_N^{(k)}$ is an indepedent sample of ξ_N for the kth update. The following result states that the update converges on the order of $O(1/k)$.

Theorem 5. *Let x^* be the (global) minimum of f and assume that $\|\nabla f(x)\|$ is uniformly bounded by some constant $G > 0$. If $\gamma_k = (lk)^{-1}$, then there exists a sequence B_k with*

$$B_k \leq \frac{\max\{\|x_1 - x^*\|^2, \; G^2/l^2\}}{k}$$

such that $\|x_k - x^*\|^2 \to B_k$ w.p.1.

Proof. Expanding $\|x_{k+1} - x^*\|^2$ by using the update rule (14), we obtain

$$\|x_{k+1} - x^*\|^2 = \|x_k - x^*\|^2 - 2\gamma_k \langle g_k, x_k - x^* \rangle + \gamma_k^2 \|g_k\|^2,$$

where $g_k \equiv g(x_k; \xi_N^{(k)})$. Because for a given x_k, g_k converges to $\nabla f(x_k)$ w.p.1, we have that conditioned on x_k,

$$\|x_{k+1} - x^*\|^2 \to \|x_k - x^*\|^2 - 2\gamma_k \langle \nabla f(x_k), x_k - x^* \rangle + \gamma_k^2 \|\nabla f(x_k)\|^2 \quad \text{w.p.1.} \tag{15}$$

On the other hand, applying the strict convexity (13), by first taking $x = x_k, y = x^*$ and then taking $x = x^*, y = x_k$, we obtain

$$\langle \nabla f(x_k), x_k - x^* \rangle \geq l \|x_k - x^*\|^2. \tag{16}$$

Substituting (16) to (15), we have that conditioned on x_k,

$$\|x_{k+1} - x^*\|^2 \to C_k \quad \text{w.p.1}$$

for some

$$C_k \leq (1 - 2l\gamma_k)\|x_k - x^*\|^2 + \gamma_k^2 G^2 = (1 - 2/k)\|x_k - x^*\|^2 + G^2/(l^2 k^2). \tag{17}$$

Now consider the randomness of x_k and apply induction. For the base case $k = 2$, the theorem clearly holds with $B_2 = C_1$. If the theorem holds for $k = T$, let $L = \max\{\|x_1 - x^*\|^2,\ G^2/l^2\}$. Then, taking the probabilistic limit of x_T on both sides of (17), we have that C_T converges w.p.1 to some limit that is less than or equal to $(1 - 2/T)(L/T) + G^2/(l^2 T^2) \leq L/(T + 1)$. Letting this limit be B_{T+1}, we complete the induction proof. □

Learning Combinatorial Optimization Algorithms over Graphs

Hanjun Dai[†*], **Elias B. Khalil**[†*], **Yuyu Zhang**[†], **Bistra Dilkina**[†], **Le Song**[†§]
[†] College of Computing, Georgia Institute of Technology
[§] Ant Financial
{hanjun.dai, elias.khalil, yuyu.zhang, bdilkina, lsong}@cc.gatech.edu

Abstract

The design of good heuristics or approximation algorithms for NP-hard combinatorial optimization problems often requires significant specialized knowledge and trial-and-error. Can we automate this challenging, tedious process, and learn the algorithms instead? In many real-world applications, it is typically the case that the same optimization problem is solved again and again on a regular basis, maintaining the same problem structure but differing in the data. This provides an opportunity for learning heuristic algorithms that exploit the structure of such recurring problems. In this paper, we propose a unique combination of reinforcement learning and graph embedding to address this challenge. The learned greedy policy behaves like a meta-algorithm that incrementally constructs a solution, and the action is determined by the output of a graph embedding network capturing the current state of the solution. We show that our framework can be applied to a diverse range of optimization problems over graphs, and learns effective algorithms for the Minimum Vertex Cover, Maximum Cut and Traveling Salesman problems.

1 Introduction

Combinatorial optimization problems over graphs arising from numerous application domains, such as social networks, transportation, telecommunications and scheduling, are NP-hard, and have thus attracted considerable interest from the theory and algorithm design communities over the years. In fact, of Karp's 21 problems in the seminal paper on reducibility [19], 10 are decision versions of graph optimization problems, while most of the other 11 problems, such as set covering, can be naturally formulated on graphs. Traditional approaches to tackling an NP-hard graph optimization problem have three main flavors: exact algorithms, approximation algorithms and heuristics. Exact algorithms are based on enumeration or branch-and-bound with an integer programming formulation, but may be prohibitive for large instances. On the other hand, polynomial-time approximation algorithms are desirable, but may suffer from weak optimality guarantees or empirical performance, or may not even exist for inapproximable problems. Heuristics are often fast, effective algorithms that lack theoretical guarantees, and may also require substantial problem-specific research and trial-and-error on the part of algorithm designers.

All three paradigms seldom exploit a common trait of real-world optimization problems: instances of the same type of problem are solved again and again on a regular basis, maintaining the same combinatorial structure, but differing mainly in their data. That is, in many applications, values of the coefficients in the objective function or constraints can be thought of as being sampled from the same underlying distribution. For instance, an advertiser on a social network targets a limited set of users with ads, in the hope that they spread them to their neighbors; such covering instances need to be solved repeatedly, since the influence pattern between neighbors may be different each time. Alternatively, a package delivery company routes trucks on a daily basis in a given city; thousands of similar optimizations need to be solved, since the underlying demand locations can differ.

[*]Both authors contributed equally to the paper.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

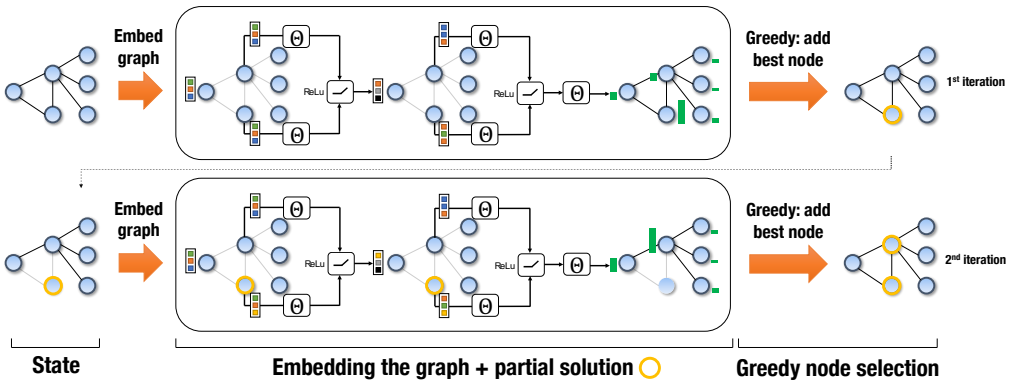


Figure 1: Illustration of the proposed framework as applied to an instance of Minimum Vertex Cover. The middle part illustrates two iterations of the graph embedding, which results in node scores (green bars).

Despite the inherent similarity between problem instances arising in the same domain, classical algorithms do not systematically exploit this fact. However, in industrial settings, a company may be willing to invest in upfront, offline computation and learning if such a process can speed up its real-time decision-making and improve its quality. This motivates the main problem we address:

> **Problem Statement:** Given a graph optimization problem G and a distribution $\mathbb{D}$ of problem instances, can we learn better heuristics that generalize to unseen instances from $\mathbb{D}$?

Recently, there has been some seminal work on using deep architectures to learn heuristics for combinatorial problems, including the Traveling Salesman Problem [37, 6, 14]. However, the architectures used in these works are generic, not yet effectively reflecting the combinatorial structure of graph problems. As we show later, these architectures often require a huge number of instances in order to learn to generalize to new ones. Furthermore, existing works typically use the policy gradient for training [6], a method that is not particularly sample-efficient. While the methods in [37, 6] can be used on graphs with different sizes – a desirable trait – they require manual, ad-hoc input/output engineering to do so (e.g. padding with zeros).

In this paper, we address the challenge of learning algorithms for graph problems using a unique combination of reinforcement learning and graph embedding. The learned policy behaves like a meta-algorithm that incrementally constructs a solution, with the action being determined by a graph embedding network over the current state of the solution. More specifically, our proposed solution framework is different from previous work in the following aspects:

1. Algorithm design pattern. We will adopt a *greedy* meta-algorithm design, whereby a feasible solution is constructed by successive addition of nodes based on the graph structure, and is maintained so as to satisfy the problem's graph constraints. Greedy algorithms are a popular pattern for designing approximation and heuristic algorithms for graph problems. As such, the same high-level design can be seamlessly used for different graph optimization problems.

2. Algorithm representation. We will use a *graph embedding* network, called `structure2vec` (S2V) [9], to represent the policy in the greedy algorithm. This novel deep learning architecture over the instance graph "featurizes" the nodes in the graph, capturing the properties of a node in the context of its graph neighborhood. This allows the policy to discriminate among nodes based on their usefulness, and generalizes to problem instances of different sizes. This contrasts with recent approaches [37, 6] that adopt a graph-agnostic sequence-to-sequence mapping that does not fully exploit graph structure.

3. Algorithm training. We will use fitted Q-learning to learn a greedy policy that is parametrized by the graph embedding network. The framework is set up in such a way that the policy will aim to optimize the objective function of the original problem instance *directly*. The main advantage of this approach is that it can deal with delayed rewards, which here represent the remaining increase in objective function value obtained by the greedy algorithm, in a data-efficient way; in each step of the greedy algorithm, the graph embeddings are updated according to the partial solution to reflect new knowledge of the benefit of *each node* to the final objective value. In contrast, the policy gradient approach of [6] updates the model parameters only once w.r.t. the whole solution (e.g. the tour in TSP).

The application of a greedy heuristic learned with our framework is illustrated in Figure 1. To demonstrate the effectiveness of the proposed framework, we apply it to three extensively studied graph optimization problems. Experimental results show that our framework, a single meta-learning algorithm, efficiently learns effective heuristics for each of the problems. Furthermore, we show that our learned heuristics preserve their effectiveness even when used on graphs much larger than the ones they were trained on. Since many combinatorial optimization problems, such as the set covering problem, can be explicitly or implicitly formulated on graphs, we believe that our work opens up a new avenue for graph algorithm design and discovery with deep learning.

2 Common Formulation for Greedy Algorithms on Graphs

We will illustrate our framework using three optimization problems over weighted graphs. Let $G(V, E, w)$ denote a weighted graph, where V is the set of nodes, E the set of edges and $w : E \to \mathbb{R}^+$ the edge weight function, i.e. $w(u, v)$ is the weight of edge $(u, v) \in E$. These problems are:

- **Minimum Vertex Cover (MVC):** Given a graph G, find a subset of nodes $S \subseteq V$ such that every edge is covered, i.e. $(u, v) \in E \Leftrightarrow u \in S$ or $v \in S$, and $|S|$ is minimized.
- **Maximum Cut (MAXCUT):** Given a graph G, find a subset of nodes $S \subseteq V$ such that the weight of the cut-set $\sum_{(u,v)\in C} w(u, v)$ is maximized, where cut-set $C \subseteq E$ is the set of edges with one end in S and the other end in $V \setminus S$.
- **Traveling Salesman Problem (TSP):** Given a set of points in 2-dimensional space, find a tour of minimum total weight, where the corresponding graph G has the points as nodes and is fully connected with edge weights corresponding to distances between points; a tour is a cycle that visits each node of the graph *exactly* once.

We will focus on a popular pattern for designing approximation and heuristic algorithms, namely a greedy algorithm. A greedy algorithm will construct a solution by sequentially adding nodes to a partial solution S, based on maximizing some *evaluation function* Q that measures the quality of a node in the context of the current partial solution. We will show that, despite the diversity of the combinatorial problems above, greedy algorithms for them can be expressed using a common formulation. Specifically:

1. A problem instance G of a given optimization problem is sampled from a distribution $\mathbb{D}$, i.e. the V, E and w of the instance graph G are generated according to a model or real-world data.
2. A partial solution is represented as an ordered list $S = (v_1, v_2, \ldots, v_{|S|})$, $v_i \in V$, and $\overline{S} = V \setminus S$ the set of candidate nodes for addition, conditional on S. Furthermore, we use a vector of binary decision variables x, with each dimension x_v corresponding to a node $v \in V$, $x_v = 1$ if $v \in S$ and 0 otherwise. One can also view x_v as a tag or extra feature on v.
3. A maintenance (or helper) procedure $h(S)$ will be needed, which maps an ordered list S to a combinatorial structure satisfying the specific constraints of a problem.
4. The quality of a partial solution S is given by an objective function $c(h(S), G)$ based on the combinatorial structure h of S.
5. A generic greedy algorithm selects a node v to add next such that v maximizes an evaluation function, $Q(h(S), v) \in \mathbb{R}$, which depends on the combinatorial structure $h(S)$ of the current partial solution. Then, the partial solution S will be extended as
$$S \coloneqq (S, v^*), \quad \text{where } v^* \coloneqq \operatorname{argmax}_{v\in\overline{S}} Q(h(S), v), \tag{1}$$
and (S, v^*) denotes appending v^* to the end of a list S. This step is repeated until a termination criterion $t(h(S))$ is satisfied.

In our formulation, we assume that the distribution $\mathbb{D}$, the helper function h, the termination criterion t and the cost function c are all given. Given the above abstract model, various optimization problems can be expressed by using different helper functions, cost functions and termination criteria:

- **MVC:** The helper function does not need to do any work, and $c(h(S), G) = -|S|$. The termination criterion checks whether all edges have been covered.
- **MAXCUT:** The helper function divides V into two sets, S and its complement $\overline{S} = V \setminus S$, and maintains a cut-set $C = \{(u, v) \,|\, (u, v) \in E, u \in S, v \in \overline{S}\}$. Then, the cost is $c(h(S), G) = \sum_{(u,v)\in C} w(u, v)$, and the termination criterion does nothing.
- **TSP:** The helper function will maintain a tour according to the order of the nodes in S. The simplest way is to append nodes to the end of partial tour in the same order as S. Then the cost $c(h(S), G) = -\sum_{i=1}^{|S|-1} w(S(i), S(i + 1)) - w(S(|S|), S(1))$, and the termination criterion is

activated when $S = V$. Empirically, inserting a node u in the partial tour at the position which increases the tour length the least is a better choice. We adopt this insertion procedure as a helper function for TSP.

An estimate of the quality of the solution resulting from adding a node to partial solution S will be determined by the *evaluation function* Q, which will be learned using a collection of problem instances. This is in contrast with traditional greedy algorithm design, where the *evaluation function* Q is typically hand-crafted, and requires substantial problem-specific research and trial-and-error. In the following, we will design a powerful deep learning parameterization for the evaluation function, $\widehat{Q}(h(S), v; \Theta)$, with parameters Θ.

3 Representation: Graph Embedding

Since we are optimizing over a graph G, we expect that the evaluation function $\widehat{Q}$ should take into account the current partial solution S as it maps to the graph. That is, $x_v = 1$ for all nodes $v \in S$, and the nodes are connected according to the graph structure. Intuitively, $\widehat{Q}$ should summarize the state of such a "tagged" graph G, and figure out the value of a new node if it is to be added in the context of such a graph. Here, both the state of the graph and the context of a node v can be very complex, hard to describe in closed form, and may depend on complicated statistics such as global/local degree distribution, triangle counts, distance to tagged nodes, etc. In order to represent such complex phenomena over combinatorial structures, we will leverage a deep learning architecture over graphs, in particular the `structure2vec` of [9], to parameterize $\widehat{Q}(h(S), v; \Theta)$.

3.1 Structure2Vec

We first provide an introduction to `structure2vec`. This graph embedding network will compute a p-dimensional feature embedding μ_v for each node $v \in V$, given the current partial solution S. More specifically, `structure2vec` defines the network architecture recursively according to an input graph structure G, and the computation graph of `structure2vec` is inspired by graphical model inference algorithms, where node-specific tags or features x_v are aggregated recursively according to G's graph topology. After a few steps of recursion, the network will produce a new embedding for each node, taking into account both graph characteristics and long-range interactions between these node features. One variant of the `structure2vec` architecture will initialize the embedding $\mu_v^{(0)}$ at each node as 0, and for all $v \in V$ update the embeddings synchronously at each iteration as

$$\mu_v^{(t+1)} \leftarrow F\left(x_v, \{\mu_u^{(t)}\}_{u \in \mathcal{N}(v)}, \{w(v,u)\}_{u \in \mathcal{N}(v)}; \Theta\right), \tag{2}$$

where $\mathcal{N}(v)$ is the set of neighbors of node v in graph G, and F is a generic nonlinear mapping such as a neural network or kernel function.

Based on the update formula, one can see that the embedding update process is carried out based on the graph topology. A new round of embedding sweeping across the nodes will start only after the embedding update for all nodes from the previous round has finished. It is easy to see that the update also defines a process where the node features x_v are propagated to other nodes via the nonlinear propagation function F. Furthermore, the more update iterations one carries out, the farther away the node features will propagate and get aggregated nonlinearly at distant nodes. In the end, if one terminates after T iterations, each node embedding $\mu_v^{(T)}$ will contain information about its T-hop neighborhood as determined by graph topology, the involved node features and the propagation function F. An illustration of two iterations of graph embedding can be found in Figure 1.

3.2 Parameterizing $\widehat{Q}(h(S), v; \Theta)$

We now discuss the parameterization of $\widehat{Q}(h(S), v; \Theta)$ using the embeddings from `structure2vec`. In particular, we design F to update a p-dimensional embedding μ_v as:

$$\mu_v^{(t+1)} \leftarrow \mathrm{relu}\big(\theta_1 x_v + \theta_2 \sum\nolimits_{u \in \mathcal{N}(v)} \mu_u^{(t)} + \theta_3 \sum\nolimits_{u \in \mathcal{N}(v)} \mathrm{relu}(\theta_4\, w(v,u))\big), \tag{3}$$

where $\theta_1 \in \mathbb{R}^p$, $\theta_2, \theta_3 \in \mathbb{R}^{p \times p}$ and $\theta_4 \in \mathbb{R}^p$ are the model parameters, and relu is the rectified linear unit ($\mathrm{relu}(z) = \max(0, z)$) applied elementwise to its input. The summation over neighbors is one way of aggregating neighborhood information that is invariant to permutations over neighbors. For simplicity of exposition, x_v here is a binary scalar as described earlier; it is straightforward to extend x_v to a vector representation by incorporating any additional useful node information. To make the

nonlinear transformations more powerful, we can add some more layers of relu before we pool over the neighboring embeddings μ_u.

Once the embedding for each node is computed after T iterations, we will use these embeddings to define the $\widehat{Q}(h(S), v; \Theta)$ function. More specifically, we will use the embedding $\mu_v^{(T)}$ for node v and the pooled embedding over the entire graph, $\sum_{u \in V} \mu_u^{(T)}$, as the surrogates for v and $h(S)$, respectively, i.e.

$$\widehat{Q}(h(S), v; \Theta) = \theta_5^\top \text{ relu}([\theta_6 \sum_{u \in V} \mu_u^{(T)}, \theta_7 \mu_v^{(T)}]) \tag{4}$$

where $\theta_5 \in \mathbb{R}^{2p}$, $\theta_6, \theta_7 \in \mathbb{R}^{p \times p}$ and $[\cdot, \cdot]$ is the concatenation operator. Since the embedding $\mu_u^{(T)}$ is computed based on the parameters from the graph embedding network, $\widehat{Q}(h(S), v)$ will depend on a collection of 7 parameters $\Theta = \{\theta_i\}_{i=1}^7$. The number of iterations T for the graph embedding computation is usually small, such as $T = 4$.

The parameters Θ will be learned. Previously, [9] required a ground truth label for every input graph G in order to train the `structure2vec` architecture. There, the output of the embedding is linked with a softmax-layer, so that the parameters can by trained end-to-end by minimizing the cross-entropy loss. This approach is not applicable to our case due to the lack of training labels. Instead, we train these parameters together *end-to-end* using reinforcement learning.

4 Training: Q-learning

We show how reinforcement learning is a natural framework for learning the evaluation function $\widehat{Q}$. The definition of the evaluation function $\widehat{Q}$ naturally lends itself to a *reinforcement learning* (RL) formulation [36], and we will use $\widehat{Q}$ as a model for the state-value function in RL. We note that we would like to learn a function $\widehat{Q}$ *across a set of* m *graphs from distribution* $\mathbb{D}$, $\mathcal{D} = \{G_i\}_{i=1}^m$, with potentially different sizes. The advantage of the graph embedding parameterization in our previous section is that we can deal with different graph instances and sizes seamlessly.

4.1 Reinforcement learning formulation

We define the states, actions and rewards in the reinforcement learning framework as follows:

1. *States*: a state S is a sequence of actions (nodes) on a graph G. Since we have already represented nodes in the tagged graph with their embeddings, the state is a vector in p-dimensional space, $\sum_{v \in V} \mu_v$. It is easy to see that this embedding representation of the state can be used across different graphs. The terminal state $\widehat{S}$ will depend on the problem at hand;
2. *Transition*: transition is deterministic here, and corresponds to tagging the node $v \in G$ that was selected as the last action with feature $x_v = 1$;
3. *Actions*: an action v is a node of G that is not part of the current state S. Similarly, we will represent actions as their corresponding p-dimensional node embedding μ_v, and such a definition is applicable across graphs of various sizes;
4. *Rewards*: the reward function $r(S, v)$ at state S is defined as the change in the cost function after taking action v and transitioning to a new state $S' := (S, v)$. That is,
$$r(S, v) = c(h(S'), G) - c(h(S), G), \tag{5}$$
and $c(h(\emptyset), G) = 0$. As such, the *cumulative reward* R of a terminal state $\widehat{S}$ coincides exactly with the objective function value of the $\widehat{S}$, i.e. $R(\widehat{S}) = \sum_{i=1}^{|\widehat{S}|} r(S_i, v_i)$ is equal to $c(h(\widehat{S}), G)$;
5. *Policy*: based on $\widehat{Q}$, a deterministic greedy policy $\pi(v|S) := \text{argmax}_{v' \in \overline{S}} \widehat{Q}(h(S), v')$ will be used. Selecting action v corresponds to adding a node of G to the current partial solution, which results in collecting a reward $r(S, v)$.

Table 1 shows the instantiations of the reinforcement learning framework for the three optimization problems considered herein. We let Q^* denote the *optimal* Q-function for each RL problem. Our graph embedding parameterization $\widehat{Q}(h(S), v; \Theta)$ from Section 3 will then be a function approximation model for it, which will be learned via n-step Q-learning.

4.2 Learning algorithm

In order to perform end-to-end learning of the parameters in $\widehat{Q}(h(S), v; \Theta)$, we use a combination of n-step Q-learning [36] and *fitted Q-iteration* [33], as illustrated in Algorithm 1. We use the term

Table 1: Definition of reinforcement learning components for each of the three problems considered.

Problem	State	Action	Helper function	Reward	Termination
MVC	subset of nodes selected so far	add node to subset	None	-1	all edges are covered
MAXCUT	subset of nodes selected so far	add node to subset	None	change in cut weight	cut weight cannot be improved
TSP	partial tour	grow tour by one node	Insertion operation	change in tour cost	tour includes all nodes

episode to refer to a complete sequence of node additions starting from an empty solution, and until termination; a *step* within an episode is a single action (node addition).

Standard (1-step) Q-learning updates the function approximator's parameters *at each step* of an episode by performing a gradient step to minimize the squared loss:

$$(y - \widehat{Q}(h(S_t), v_t; \Theta))^2, \tag{6}$$

where $y = \gamma \max_{v'} \widehat{Q}(h(S_{t+1}), v'; \Theta) + r(S_t, v_t)$ for a non-terminal state S_t. The n-step Q-learning helps deal with the issue of *delayed rewards*, where the final reward of interest to the agent is only received far in the future during an episode. In our setting, the final objective value of a solution is only revealed after many node additions. As such, the 1-step update may be too myopic. A natural extension of 1-step Q-learning is to wait n steps before updating the approximator's parameters, so as to collect a more accurate estimate of the future rewards. Formally, the update is over the same squared loss (6), but with a different target, $y = \sum_{i=0}^{n-1} r(S_{t+i}, v_{t+i}) + \gamma \max_{v'} \widehat{Q}(h(S_{t+n}), v'; \Theta)$. The fitted Q-iteration approach has been shown to result in faster learning convergence when using a neural network as a function approximator [33, 28], a property that also applies in our setting, as we use the embedding defined in Section 3.2. Instead of updating the Q-function sample-by-sample as in Equation (6), the fitted Q-iteration approach uses *experience replay* to update the function approximator with a batch of samples from a dataset E, rather than the single sample being currently experienced. The dataset E is populated during previous episodes, such that at step $t + n$, the tuple $(S_t, a_t, R_{t,t+n}, S_{t+n})$ is added to E, with $R_{t,t+n} = \sum_{i=0}^{n-1} r(S_{t+i}, a_{t+i})$. Instead of performing a gradient step in the loss of the current sample as in (6), stochastic gradient descent updates are performed on a random sample of tuples drawn from E.

It is known that *off-policy* reinforcement learning algorithms such as Q-learning can be more sample efficient than their policy gradient counterparts [15]. This is largely due to the fact that policy gradient methods require *on-policy* samples for the new policy obtained after each parameter update of the function approximator.

Algorithm 1 Q-learning for the Greedy Algorithm

Initialize experience replay memory $\mathcal{M}$ to capacity N
for episode $e = 1$ **to** L **do**
  Draw graph G from distribution $\mathbb{D}$
  Initialize the state to empty $S_1 = ()$
  for step $t = 1$ **to** T **do**
    $v_t = \begin{cases} \text{random node } v \in \overline{S}_t, & \text{w.p. } \epsilon \\ \text{argmax}_{v \in \overline{S}_t} \widehat{Q}(h(S_t), v; \Theta), & \text{otherwise} \end{cases}$
    Add v_t to partial solution: $S_{t+1} := (S_t, v_t)$
    if $t \geq n$ **then**
      Add tuple $(S_{t-n}, v_{t-n}, R_{t-n,t}, S_t)$ to $\mathcal{M}$
      Sample random batch from $B \overset{iid.}{\sim} \mathcal{M}$
      Update Θ by SGD over (6) for B
    end if
  end for
end for
return Θ

5 Experimental Evaluation

Instance generation. To evaluate the proposed method against other approximation/heuristic algorithms and deep learning approaches, we generate graph instances for each of the three problems. For the MVC and MAXCUT problems, we generate Erdős-Renyi (ER) [11] and Barabasi-Albert (BA) [1] graphs which have been used to model many real-world networks. For a given range on the number of nodes, e.g. 50-100, we first sample the number of nodes uniformly at random from that

range, then generate a graph according to either ER or BA. For the two-dimensional TSP problem, we use an instance generator from the DIMACS TSP Challenge [18] to generate uniformly random or clustered points in the 2-D grid. We refer the reader to the Appendix D.1 for complete details on instance generation. We have also tackled the Set Covering Problem, for which the description and results are deferred to Appendix B.

Structure2Vec Deep Q-learning. For our method, S2V-DQN, we use the graph representations and hyperparameters described in Appendix D.4. The hyperparameters are selected via preliminary results on small graphs, and then fixed for large ones. Note that for TSP, where the graph is fully-connected, we build the K-nearest neighbor graph ($K = 10$) to scale up to large graphs. For MVC, where we train the model on graphs with up to 500 nodes, we use the model trained on small graphs as initialization for training on larger ones. We refer to this trick as "pre-training", which is illustrated in Figure D.2.

Pointer Networks with Actor-Critic. We compare our method to a method, based on Recurrent Neural Networks (RNNs), which does not make full use of graph structure [6]. We implement and train their algorithm (PN-AC) for all three problems. The original model only works on the Euclidian TSP problem, where each node is represented by its (x, y) coordinates, and is not designed for problems with graph structure. To handle other graph problems, we describe each node by its adjacency vector instead of coordinates. To handle different graph sizes, we use a singular value decomposition (SVD) to obtain a rank-8 approximation for the adjacency matrix, and use the low-rank embeddings as inputs to the pointer network.

Baseline Algorithms. Besides the PN-AC, we also include powerful approximation or heuristic algorithms from the literature. These algorithms are specifically designed for each type of problem:

- **MVC:** *MVCApprox* iteratively selects an uncovered edge and adds both of its endpoints [30]. We designed a stronger variant, called *MVCApprox-Greedy*, that greedily picks the uncovered edge with maximum sum of degrees of its endpoints. Both algorithms are 2-approximations.
- **MAXCUT:** We include *MaxcutApprox*, which maintains the cut set $(S, V \setminus S)$ and moves a node from one side to the other side of the cut if that operation results in cut weight improvement [25]. To make *MaxcutApprox* stronger, we greedily move the node that results in the largest improvement in cut weight. A randomized, non-greedy algorithm, referred to as SDP, is also implemented based on [12]; 100 solutions are generated for each graph, and the best one is taken.
- **TSP:** We include the following approximation algorithms: Minimum Spanning Tree (MST), Farthest insertion (Farthest), Cheapest insertion (Cheapest), Closest insertion (Closest), Christofides and 2-opt. We also add the Nearest Neighbor heuristic (Nearest); see [4] for algorithmic details.

Details on Validation and Testing. For S2V-DQN and PN-AC, we use a CUDA K80-enabled cluster for training and testing. Training convergence for S2V-DQN is discussed in Appendix D.6. S2V-DQN and PN-AC use 100 held-out graphs for validation, and we report the test results on another 1000 graphs. We use CPLEX[17] to get optimal solutions for MVC and MAXCUT, and Concorde [3] for TSP (details in Appendix D.1). All approximation ratios reported in the paper are with respect to the best (possibly optimal) solution found by the solvers within 1 hour. For MVC, we vary the training and test graph sizes in the ranges {15–20, 40–50, 50–100, 100–200, 400–500}. For MAXCUT and TSP, which involve edge weights, we train up to 200–300 nodes due to the limited computation resource. For all problems, we test on graphs of size up to 1000–1200.

During testing, instead of using Active Search as in [6], we simply use the greedy policy. This gives us much faster inference, while still being powerful enough. We modify existing open-source code to implement both S2V-DQN [2] and PN-AC [3]. Our code is publicly available [4].

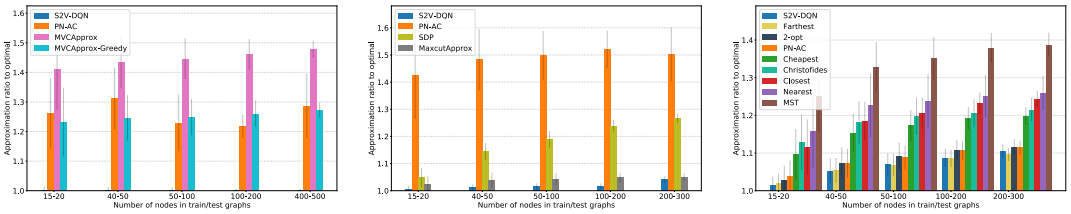
5.1 Comparison of solution quality

To evaluate the solution quality on test instances, we use the *approximation ratio* of each method relative to the optimal solution, averaged over the set of test instances. The approximation ratio of a solution S to a problem instance G is defined as $\mathcal{R}(S, G) = \max(\frac{OPT(G)}{c(h(S))}, \frac{c(h(S))}{OPT(G)})$, where $c(h(S))$ is the objective value of solution S, and $OPT(G)$ is the best-known solution value of instance G.

[2] https://github.com/Hanjun-Dai/graphnn

[3] https://github.com/devsisters/pointer-network-tensorflow

[4] https://github.com/Hanjun-Dai/graph_comb_opt



(a) MVC BA (b) MAXCUT BA (c) TSP random

Figure 2: Approximation ratio on 1000 test graphs. Note that on MVC, our performance is pretty close to optimal. In this figure, training and testing graphs are generated according to the same distribution.

Figure 2 shows the average approximation ratio across the three problems; other graph types are in Figure D.1 in the appendix. In all of these figures, a lower approximation ratio is better. Overall, our proposed method, S2V-DQN, performs significantly better than other methods. In MVC, the performance of S2V-DQN is particularly good, as the approximation ratio is roughly 1 and the bar is barely visible.

The PN-AC algorithm performs well on TSP, as expected. Since the TSP graph is essentially fully-connected, graph structure is not as important. On problems such as MVC and MAXCUT, where graph information is more crucial, our algorithm performs significantly better than PN-AC. For TSP, The Farthest and 2-opt algorithm perform as well as S2V-DQN, and slightly better in some cases. However, we will show later that in real-world TSP data, our algorithm still performs better.

5.2 Generalization to larger instances

The graph embedding framework enables us to train and test on graphs of different sizes, since the same set of model parameters are used. How does the performance of the learned algorithm using small graphs generalize to test graphs of larger sizes? To investigate this, we train S2V-DQN on graphs with 50–100 nodes, and test its generalization ability on graphs with up to 1200 nodes. Table 2 summarizes the results, and full results are in Appendix D.3.

Table 2: S2V-DQN's generalization ability. Values are average approximation ratios over 1000 test instances. These test results are produced by S2V-DQN algorithms trained on graphs with 50-100 nodes.

Test Size	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
MVC (BA)	1.0033	1.0041	1.0045	1.0040	1.0045	1.0048	1.0062
MAXCUT (BA)	1.0150	1.0181	1.0202	1.0188	1.0123	1.0177	1.0038
TSP (clustered)	1.0730	1.0895	1.0869	1.0918	1.0944	1.0975	1.1065

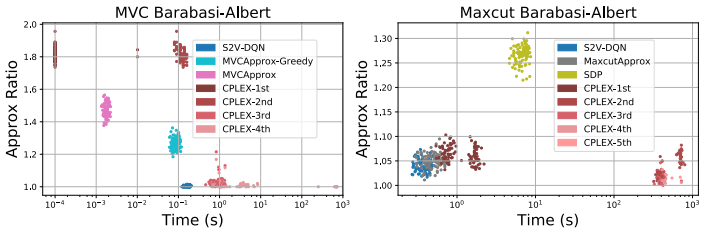
We can see that S2V-DQN achieves a very good approximation ratio. Note that the "optimal" value used in the computation of approximation ratios may not be truly optimal (due to the solver time cutoff at 1 hour), and so CPLEX's solutions do typically get worse as problem size grows. This is why sometimes we can even get better approximation ratio on larger graphs.

5.3 Scalability & Trade-off between running time and approximation ratio

To construct a solution on a test graph, our algorithm has polynomial complexity of $O(k|E|)$ where k is number of greedy steps (at most the number of nodes $|V|$) and $|E|$ is number of edges. For instance, on graphs with 1200 nodes, we can find the solution of MVC within 11 seconds using a single GPU, while getting an approximation ratio of 1.0062. For dense graphs, we can also sample the edges for the graph embedding computation to save time, a measure we will investigate in the future.

Figure 3 illustrates the approximation ratios of various approaches as a function of running time. All algorithms report a single solution at termination, whereas CPLEX reports multiple improving solutions, for which we recorded the corresponding running time and approximation ratio. Figure D.3 (Appendix D.7) includes other graph sizes and types, where the results are consistent with Figure 3.

Figure 3 shows that, for MVC, we are slightly slower than the approximation algorithms but enjoy a much better approximation ratio. Also note that although CPLEX found the first feasible solution quickly, it also has much worse ratio; the second improved solution found by CPLEX takes similar or longer time than our S2V-DQN, but is still of worse quality. For MAXCUT, the observations are still consistent. One should be aware that sometimes our algorithm can obtain better results than 1-hour CPLEX, which gives ratios below 1.0. Furthermore, sometimes S2V-DQN is even faster than the



(a) MVC BA 200-300 (b) MAXCUT BA 200-300

Figure 3: Time-approximation trade-off for MVC and MAXCUT. In this figure, each dot represents a solution found for a single problem instance, for 100 instances. For CPLEX, we also record the time and quality of each solution it finds, e.g. CPLEX-1st means the first feasible solution found by CPLEX.

MaxcutApprox, although this comparison is not exactly fair, since we use GPUs; however, we can still see that our algorithm is efficient.

5.4 Experiments on real-world datasets

In addition to the experiments for synthetic data, we identified sets of publicly available benchmark or real-world instances for each problem, and performed experiments on them. A summary of results is in Table 3, and details are given in Appendix C. S2V-DQN significantly outperforms all competing methods for MVC, MAXCUT and TSP.

Table 3: Realistic data experiments, results summary. Values are average approximation ratios.

Problem	Dataset	S2V-DQN	Best Competitor	2nd Best Competitor
MVC	MemeTracker	**1.0021**	1.2220 (MVCApprox-Greedy)	1.4080 (MVCApprox)
MAXCUT	Physics	**1.0223**	1.2825 (MaxcutApprox)	1.8996 (SDP)
TSP	TSPLIB	**1.0475**	1.0800 (Farthest)	1.0947 (2-opt)

5.5 Discovery of interesting new algorithms

We further examined the algorithms learned by S2V-DQN, and tried to interpret what greedy heuristics have been learned. We found that S2V-DQN is able to discover new and interesting algorithms which intuitively make sense but have not been analyzed before. For instance, S2V-DQN discovers an algorithm for MVC where nodes are selected to balance between their degrees and the connectivity of the remaining graph (Appendix Figures D.4 and D.7). For MAXCUT, S2V-DQN discovers an algorithm where nodes are picked to avoid cancelling out existing edges in the cut set (Appendix Figure D.5). These results suggest that S2V-DQN may also be a good assistive tool for discovering new algorithms, especially in cases when the graph optimization problems are new and less well-studied.

6 Conclusions

We presented an end-to-end machine learning framework for automatically designing greedy heuristics for hard combinatorial optimization problems on graphs. Central to our approach is the combination of a deep graph embedding with reinforcement learning. Through extensive experimental evaluation, we demonstrate the effectiveness of the proposed framework in learning greedy heuristics as compared to manually-designed greedy algorithms. The excellent performance of the learned heuristics is consistent across multiple different problems, graph types, and graph sizes, suggesting that the framework is a promising new tool for designing algorithms for graph problems.

Acknowledgments

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. Dilkina is supported by NSF grant CCF-1522054 and ExxonMobil.

References

[1] Albert, Réka and Barabási, Albert-László. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[2] Andrychowicz, Marcin, Denil, Misha, Gomez, Sergio, Hoffman, Matthew W, Pfau, David, Schaul, Tom, and de Freitas, Nando. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pp. 3981–3989, 2016.

[3] Applegate, David, Bixby, Robert, Chvatal, Vasek, and Cook, William. Concorde TSP solver, 2006.

[4] Applegate, David L, Bixby, Robert E, Chvatal, Vasek, and Cook, William J. *The traveling salesman problem: a computational study*. Princeton university press, 2011.

[5] Balas, Egon and Ho, Andrew. Set covering algorithms using cutting planes, heuristics, and subgradient optimization: a computational study. *Combinatorial Optimization*, pp. 37–60, 1980.

[6] Bello, Irwan, Pham, Hieu, Le, Quoc V, Norouzi, Mohammad, and Bengio, Samy. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[7] Boyan, Justin and Moore, Andrew W. Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research*, 1(Nov):77–112, 2000.

[8] Chen, Yutian, Hoffman, Matthew W, Colmenarejo, Sergio Gomez, Denil, Misha, Lillicrap, Timothy P, and de Freitas, Nando. Learning to learn for global optimization of black box functions. *arXiv preprint arXiv:1611.03824*, 2016.

[9] Dai, Hanjun, Dai, Bo, and Song, Le. Discriminative embeddings of latent variable models for structured data. In *ICML*, 2016.

[10] Du, Nan, Song, Le, Gomez-Rodriguez, Manuel, and Zha, Hongyuan. Scalable influence estimation in continuous-time diffusion networks. In *NIPS*, 2013.

[11] Erdos, Paul and Rényi, A. On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci*, 5:17–61, 1960.

[12] Goemans, M.X. and Williamson, D. P. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6): 1115–1145, 1995.

[13] Gomez-Rodriguez, Manuel, Leskovec, Jure, and Krause, Andreas. Inferring networks of diffusion and influence. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1019–1028. ACM, 2010.

[14] Graves, Alex, Wayne, Greg, Reynolds, Malcolm, Harley, Tim, Danihelka, Ivo, Grabska-Barwińska, Agnieszka, Colmenarejo, Sergio Gómez, Grefenstette, Edward, Ramalho, Tiago, Agapiou, John, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[15] Gu, Shixiang, Lillicrap, Timothy, Ghahramani, Zoubin, Turner, Richard E, and Levine, Sergey. Q-prop: Sample-efficient policy gradient with an off-policy critic. *arXiv preprint arXiv:1611.02247*, 2016.

[16] He, He, Daume III, Hal, and Eisner, Jason M. Learning to search in branch and bound algorithms. In *Advances in Neural Information Processing Systems*, pp. 3293–3301, 2014.

[17] IBM. CPLEX User's Manual, Version 12.6.1, 2014.

[18] Johnson, David S and McGeoch, Lyle A. Experimental analysis of heuristics for the stsp. In *The traveling salesman problem and its variations*, pp. 369–443. Springer, 2007.

[19] Karp, Richard M. Reducibility among combinatorial problems. In *Complexity of computer computations*, pp. 85–103. Springer, 1972.

[20] Kempe, David, Kleinberg, Jon, and Tardos, Éva. Maximizing the spread of influence through a social network. In *KDD*, pp. 137–146. ACM, 2003.

[21] Khalil, Elias B., Dilkina, B., and Song, L. Scalable diffusion-aware optimization of network topology. In *Knowledge Discovery and Data Mining (KDD)*, 2014.

[22] Khalil, Elias B., Le Bodic, Pierre, Song, Le, Nemhauser, George L, and Dilkina, Bistra N. Learning to branch in mixed integer programming. In *AAAI*, pp. 724–731, 2016.

[23] Khalil, Elias B., Dilkina, Bistra, Nemhauser, George, Ahmed, Shabbir, and Shao, Yufen. Learning to run heuristics in tree search. In *26th International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

[24] Kingma, Diederik and Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[25] Kleinberg, Jon and Tardos, Eva. *Algorithm design*. Pearson Education India, 2006.

[26] Lagoudakis, Michail G and Littman, Michael L. Learning to select branching rules in the dpll procedure for satisfiability. *Electronic Notes in Discrete Mathematics*, 9:344–359, 2001.

[27] Li, Ke and Malik, Jitendra. Learning to optimize. *arXiv preprint arXiv:1606.01885*, 2016.

[28] Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Graves, Alex, Antonoglou, Ioannis, Wierstra, Daan, and Riedmiller, Martin A. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

[29] Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[30] Papadimitriou, C. H. and Steiglitz, K. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, New Jersey, 1982.

[31] Peleg, David, Schechtman, Gideon, and Wool, Avishai. Approximating bounded 0-1 integer linear programs. In *Theory and Computing Systems, 1993., Proceedings of the 2nd Israel Symposium on the*, pp. 69–77. IEEE, 1993.

[32] Reinelt, Gerhard. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3 (4):376–384, 1991.

[33] Riedmiller, Martin. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pp. 317–328. Springer, 2005.

[34] Sabharwal, Ashish, Samulowitz, Horst, and Reddy, Chandra. Guiding combinatorial optimization with uct. In *CPAIOR*, pp. 356–361. Springer, 2012.

[35] Samulowitz, Horst and Memisevic, Roland. Learning to solve QBF. In *AAAI*, 2007.

[36] Sutton, R.S. and Barto, A.G. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[37] Vinyals, Oriol, Fortunato, Meire, and Jaitly, Navdeep. Pointer networks. In *Advances in Neural Information Processing Systems*, pp. 2692–2700, 2015.

[38] Zhang, Wei and Dietterich, Thomas G. Solving combinatorial optimization tasks by reinforcement learning: A general methodology applied to resource-constrained scheduling. *Journal of Artificial Intelligence Reseach*, 1:1–38, 2000.

HOW POWERFUL ARE GRAPH NEURAL NETWORKS?

Keyulu Xu *†
MIT
keyulu@mit.edu

Weihua Hu *‡
Stanford University
weihuahu@stanford.edu

Jure Leskovec
Stanford University
jure@cs.stanford.edu

Stefanie Jegelka
MIT
stefje@mit.edu

ABSTRACT

Graph Neural Networks (GNNs) are an effective framework for representation learning of graphs. GNNs follow a neighborhood aggregation scheme, where the representation vector of a node is computed by recursively aggregating and transforming representation vectors of its neighboring nodes. Many GNN variants have been proposed and have achieved state-of-the-art results on both node and graph classification tasks. However, despite GNNs revolutionizing graph representation learning, there is limited understanding of their representational properties and limitations. Here, we present a theoretical framework for analyzing the expressive power of GNNs to capture different graph structures. Our results characterize the discriminative power of popular GNN variants, such as Graph Convolutional Networks and GraphSAGE, and show that they cannot learn to distinguish certain simple graph structures. We then develop a simple architecture that is provably the most expressive among the class of GNNs and is as powerful as the Weisfeiler-Lehman graph isomorphism test. We empirically validate our theoretical findings on a number of graph classification benchmarks, and demonstrate that our model achieves state-of-the-art performance.

1 INTRODUCTION

Learning with graph structured data, such as molecules, social, biological, and financial networks, requires effective representation of their graph structure (Hamilton et al., 2017b). Recently, there has been a surge of interest in Graph Neural Network (GNN) approaches for representation learning of graphs (Li et al., 2016; Hamilton et al., 2017a; Kipf & Welling, 2017; Velickovic et al., 2018; Xu et al., 2018). GNNs broadly follow a recursive neighborhood aggregation (or message passing) scheme, where each node aggregates feature vectors of its neighbors to compute its new feature vector (Xu et al., 2018; Gilmer et al., 2017). After k iterations of aggregation, a node is represented by its transformed feature vector, which captures the structural information within the node's k-hop neighborhood. The representation of an entire graph can then be obtained through pooling (Ying et al., 2018), for example, by summing the representation vectors of all nodes in the graph.

Many GNN variants with different neighborhood aggregation and graph-level pooling schemes have been proposed (Scarselli et al., 2009b; Battaglia et al., 2016; Defferrard et al., 2016; Duvenaud et al., 2015; Hamilton et al., 2017a; Kearnes et al., 2016; Kipf & Welling, 2017; Li et al., 2016; Velickovic et al., 2018; Santoro et al., 2017; Xu et al., 2018; Santoro et al., 2018; Verma & Zhang, 2018; Ying et al., 2018; Zhang et al., 2018). Empirically, these GNNs have achieved state-of-the-art performance in many tasks such as node classification, link prediction, and graph classification. However, the design of new GNNs is mostly based on empirical intuition, heuristics, and experimental trial-and-error. There is little theoretical understanding of the properties and limitations of GNNs, and formal analysis of GNNs' representational capacity is limited.

*Equal contribution.
†Work partially performed while in Tokyo, visiting Prof. Ken-ichi Kawarabayashi.
‡Work partially performed while at RIKEN AIP and University of Tokyo.

Here, we present a theoretical framework for analyzing the representational power of GNNs. We formally characterize how expressive different GNN variants are in learning to represent and distinguish between different graph structures. Our framework is inspired by the close connection between GNNs and the Weisfeiler-Lehman (WL) graph isomorphism test (Weisfeiler & Lehman, 1968), a powerful test known to distinguish a broad class of graphs (Babai & Kucera, 1979). Similar to GNNs, the WL test iteratively updates a given node's feature vector by aggregating feature vectors of its network neighbors. What makes the WL test so powerful is its injective aggregation update that maps different node neighborhoods to different feature vectors. Our key insight is that a GNN can have as large discriminative power as the WL test if the GNN's aggregation scheme is highly expressive and can model injective functions.

To mathematically formalize the above insight, our framework first represents the set of feature vectors of a given node's neighbors as a *multiset*, *i.e.*, a set with possibly repeating elements. Then, the neighbor aggregation in GNNs can be thought of as an *aggregation function over the multiset*. Hence, to have strong representational power, a GNN must be able to aggregate different multisets into different representations. We rigorously study several variants of multiset functions and theoretically characterize their discriminative power, *i.e.*, how well different aggregation functions can distinguish different multisets. The more discriminative the multiset function is, the more powerful the representational power of the underlying GNN.

Our main results are summarized as follows:

1) We show that GNNs are *at most* as powerful as the WL test in distinguishing graph structures.
2) We establish conditions on the neighbor aggregation and graph readout functions under which the resulting GNN is *as powerful as* the WL test.
3) We identify graph structures that cannot be distinguished by popular GNN variants, such as GCN (Kipf & Welling, 2017) and GraphSAGE (Hamilton et al., 2017a), and we precisely characterize the kinds of graph structures such GNN-based models can capture.
4) We develop a simple neural architecture, *Graph Isomorphism Network (GIN)*, and show that its discriminative/representational power is equal to the power of the WL test.

We validate our theory via experiments on graph classification datasets, where the expressive power of GNNs is crucial to capture graph structures. In particular, we compare the performance of GNNs with various aggregation functions. Our results confirm that the most powerful GNN by our theory, *i.e.*, Graph Isomorphism Network (GIN), also empirically has high representational power as it almost perfectly fits the training data, whereas the less powerful GNN variants often severely underfit the training data. In addition, the representationally more powerful GNNs outperform the others by test set accuracy and achieve state-of-the-art performance on many graph classification benchmarks.

2 PRELIMINARIES

We begin by summarizing some of the most common GNN models and, along the way, introduce our notation. Let $G = (V, E)$ denote a graph with node feature vectors X_v for $v \in V$. There are two tasks of interest: (1) *Node classification*, where each node $v \in V$ has an associated label y_v and the goal is to learn a representation vector h_v of v such that v's label can be predicted as $y_v = f(h_v)$; (2) *Graph classification*, where, given a set of graphs $\{G_1, ..., G_N\} \subseteq \mathcal{G}$ and their labels $\{y_1, ..., y_N\} \subseteq \mathcal{Y}$, we aim to learn a representation vector h_G that helps predict the label of an entire graph, $y_G = g(h_G)$.

Graph Neural Networks. GNNs use the graph structure and node features X_v to learn a representation vector of a node, h_v, or the entire graph, h_G. Modern GNNs follow a neighborhood aggregation strategy, where we iteratively update the representation of a node by aggregating representations of its neighbors. After k iterations of aggregation, a node's representation captures the structural information within its k-hop network neighborhood. Formally, the k-th layer of a GNN is

$$a_v^{(k)} = \text{AGGREGATE}^{(k)}\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right), \quad h_v^{(k)} = \text{COMBINE}^{(k)}\left(h_v^{(k-1)}, a_v^{(k)}\right), \tag{2.1}$$

where $h_v^{(k)}$ is the feature vector of node v at the k-th iteration/layer. We initialize $h_v^{(0)} = X_v$, and $\mathcal{N}(v)$ is a set of nodes adjacent to v. The choice of $\text{AGGREGATE}^{(k)}(\cdot)$ and $\text{COMBINE}^{(k)}(\cdot)$ in

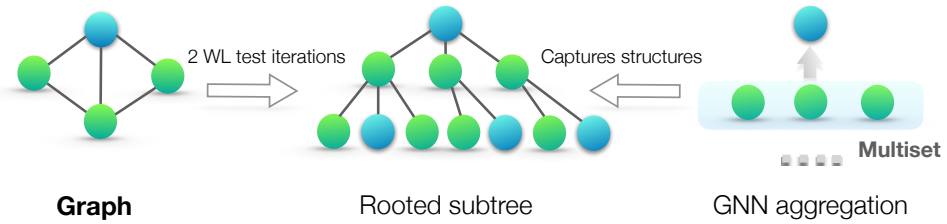


Figure 1: **An overview of our theoretical framework.** Middle panel: rooted subtree structures (at the blue node) that the WL test uses to distinguish different graphs. Right panel: if a GNN's aggregation function captures the *full multiset* of node neighbors, the GNN can capture the rooted subtrees in a recursive manner and be as powerful as the WL test.

GNNs is crucial. A number of architectures for AGGREGATE have been proposed. In the pooling variant of GraphSAGE (Hamilton et al., 2017a), AGGREGATE has been formulated as

$$a_v^{(k)} = \text{MAX}\left(\left\{\text{ReLU}\left(W \cdot h_u^{(k-1)}\right),\ \forall u \in \mathcal{N}(v)\right\}\right), \tag{2.2}$$

where W is a learnable matrix, and MAX represents an element-wise max-pooling. The COMBINE step could be a concatenation followed by a linear mapping $W \cdot \left[h_v^{(k-1)}, a_v^{(k)}\right]$ as in GraphSAGE. In Graph Convolutional Networks (GCN) (Kipf & Welling, 2017), the element-wise *mean* pooling is used instead, and the AGGREGATE and COMBINE steps are integrated as follows:

$$h_v^{(k)} = \text{ReLU}\left(W \cdot \text{MEAN}\left\{h_u^{(k-1)},\ \forall u \in \mathcal{N}(v) \cup \{v\}\right\}\right). \tag{2.3}$$

Many other GNNs can be represented similarly to Eq. 2.1 (Xu et al., 2018; Gilmer et al., 2017).

For node classification, the node representation $h_v^{(K)}$ of the final iteration is used for prediction. For graph classification, the READOUT function aggregates node features from the final iteration to obtain the entire graph's representation h_G:

$$h_G = \text{READOUT}\left(\left\{h_v^{(K)} \mid v \in G\right\}\right). \tag{2.4}$$

READOUT can be a simple permutation invariant function such as summation or a more sophisticated graph-level pooling function (Ying et al., 2018; Zhang et al., 2018).

Weisfeiler-Lehman test. The graph isomorphism problem asks whether two graphs are topologically identical. This is a challenging problem: no polynomial-time algorithm is known for it yet (Garey, 1979; Garey & Johnson, 2002; Babai, 2016). Apart from some corner cases (Cai et al., 1992), the Weisfeiler-Lehman (WL) test of graph isomorphism (Weisfeiler & Lehman, 1968) is an effective and computationally efficient test that distinguishes a broad class of graphs (Babai & Kucera, 1979). Its 1-dimensional form, "naïve vertex refinement", is analogous to neighbor aggregation in GNNs. The WL test iteratively (1) aggregates the labels of nodes and their neighborhoods, and (2) hashes the aggregated labels into *unique* new labels. The algorithm decides that two graphs are non-isomorphic if at some iteration the labels of the nodes between the two graphs differ.

Based on the WL test, Shervashidze et al. (2011) proposed the WL subtree kernel that measures the similarity between graphs. The kernel uses the counts of node labels at different iterations of the WL test as the feature vector of a graph. Intuitively, a node's label at the k-th iteration of WL test represents a subtree structure of height k rooted at the node (Figure 1). Thus, the graph features considered by the WL subtree kernel are essentially counts of different rooted subtrees in the graph.

3 Theoretical Framework: Overview

We start with an overview of our framework for analyzing the expressive power of GNNs. Figure 1 illustrates our idea. A GNN recursively updates each node's feature vector to capture the network structure and features of other nodes around it, *i.e.*, its rooted subtree structures (Figure 1). Throughout the paper, we assume node input features are from a countable universe. For finite graphs, node feature vectors at deeper layers of any fixed model are also from a countable universe. For notational simplicity, we can assign each feature vector a unique label in $\{a, b, c \ldots\}$. Then, feature vectors of a set of neighboring nodes form a *multiset* (Figure 1): the same element can appear multiple times since different nodes can have identical feature vectors.

Definition 1 (Multiset)**.** A multiset is a generalized concept of a set that allows multiple instances for its elements. More formally, a multiset is a 2-tuple $X = (S, m)$ where S is the *underlying set* of X that is formed from its *distinct elements*, and $m : S \to \mathbb{N}_{\geq 1}$ gives the *multiplicity* of the elements.

To study the representational power of a GNN, we analyze when a GNN maps two nodes to the same location in the embedding space. Intuitively, a maximally powerful GNN maps two nodes to the same location *only if* they have identical subtree structures with identical features on the corresponding nodes. Since subtree structures are defined recursively via node neighborhoods (Figure 1), we can reduce our analysis to the question whether a GNN maps two neighborhoods (*i.e.*, two multisets) to the same embedding or representation. A maximally powerful GNN would *never* map two different neighborhoods, *i.e.*, multisets of feature vectors, to the same representation. This means its aggregation scheme must be *injective*. Thus, we abstract a GNN's aggregation scheme as a class of functions over multisets that their neural networks can represent, and analyze whether they are able to represent injective multiset functions.

Next, we use this reasoning to develop a maximally powerful GNN. In Section 5, we study popular GNN variants and see that their aggregation schemes are inherently not injective and thus less powerful, but that they can capture other interesting properties of graphs.

4 Building powerful graph neural networks

First, we characterize the maximum representational capacity of a general class of GNN-based models. Ideally, a maximally powerful GNN could distinguish different graph structures by mapping them to different representations in the embedding space. This ability to map any two different graphs to different embeddings, however, implies solving the challenging graph isomorphism problem. That is, we want isomorphic graphs to be mapped to the same representation and non-isomorphic ones to different representations. In our analysis, we characterize the representational capacity of GNNs via a slightly weaker criterion: a powerful heuristic called *Weisfeiler-Lehman (WL) graph isomorphism test*, that is known to work well in general, with a few exceptions, e.g., regular graphs (Cai et al., 1992; Douglas, 2011; Evdokimov & Ponomarenko, 1999).

Lemma 2. *Let G_1 and G_2 be any two non-isomorphic graphs. If a graph neural network $\mathcal{A} : \mathcal{G} \to \mathbb{R}^d$ maps G_1 and G_2 to different embeddings, the Weisfeiler-Lehman graph isomorphism test also decides G_1 and G_2 are not isomorphic.*

Proofs of all Lemmas and Theorems can be found in the Appendix. Hence, any aggregation-based GNN is at most as powerful as the WL test in distinguishing different graphs. A natural follow-up question is whether there exist GNNs that are, in principle, as powerful as the WL test? Our answer, in Theorem 3, is yes: if the neighbor aggregation and graph-level readout functions are injective, then the resulting GNN is as powerful as the WL test.

Theorem 3. *Let $\mathcal{A} : \mathcal{G} \to \mathbb{R}^d$ be a GNN. With a sufficient number of GNN layers, $\mathcal{A}$ maps any graphs G_1 and G_2 that the Weisfeiler-Lehman test of isomorphism decides as non-isomorphic, to different embeddings if the following conditions hold:*

a) $\mathcal{A}$ aggregates and updates node features iteratively with

$$h_v^{(k)} = \phi\left(h_v^{(k-1)}, f\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)\right),$$

where the functions f, which operates on multisets, and ϕ are injective.

b) $\mathcal{A}$'s graph-level readout, which operates on the multiset of node features $\left\{h_v^{(k)}\right\}$, is injective.

We prove Theorem 3 in the appendix. For countable sets, injectiveness well characterizes whether a function preserves the distinctness of inputs. Uncountable sets, where node features are continuous, need some further considerations. In addition, it would be interesting to characterize how close together the learned features lie in a function's image. We leave these questions for future work, and focus on the case where input node features are from a countable set (that can be a subset of an uncountable set such as $\mathbb{R}^n$).

Lemma 4. *Assume the input feature space $\mathcal{X}$ is countable. Let $g^{(k)}$ be the function parameterized by a GNN's k-th layer for $k = 1, ..., L$, where $g^{(1)}$ is defined on multisets $X \subset \mathcal{X}$ of bounded size. The range of $g^{(k)}$, i.e., the space of node hidden features $h_v^{(k)}$, is also countable for all $k = 1, ..., L$.*

Here, it is also worth discussing an important benefit of GNNs beyond distinguishing different graphs, that is, capturing similarity of graph structures. Note that node feature vectors in the WL test are essentially one-hot encodings and thus cannot capture the similarity between subtrees. In contrast, a GNN satisfying the criteria in Theorem 3 generalizes the WL test by *learning to embed* the subtrees to low-dimensional space. This enables GNNs to not only discriminate different structures, but also to learn to map similar graph structures to similar embeddings and capture dependencies between graph structures. Capturing structural similarity of the node labels is shown to be helpful for generalization particularly when the co-occurrence of subtrees is sparse across different graphs or there are noisy edges and node features (Yanardag & Vishwanathan, 2015).

4.1 Graph Isomorphism Network (GIN)

Having developed conditions for a maximally powerful GNN, we next develop a simple architecture, *Graph Isomorphism Network (GIN)*, that provably satisfies the conditions in Theorem 3. This model generalizes the WL test and hence achieves maximum discriminative power among GNNs.

To model injective multiset functions for the neighbor aggregation, we develop a theory of "deep multisets", *i.e.*, parameterizing universal multiset functions with neural networks. Our next lemma states that sum aggregators can represent injective, in fact, *universal* functions over multisets.

Lemma 5. *Assume $\mathcal{X}$ is countable. There exists a function $f : \mathcal{X} \to \mathbb{R}^n$ so that $h(X) = \sum_{x \in X} f(x)$ is unique for each multiset $X \subset \mathcal{X}$ of bounded size. Moreover, any multiset function g can be decomposed as $g(X) = \phi\left(\sum_{x \in X} f(x)\right)$ for some function ϕ.*

We prove Lemma 5 in the appendix. The proof extends the setting in (Zaheer et al., 2017) from sets to multisets. An important distinction between deep multisets and sets is that certain popular injective set functions, such as the mean aggregator, are not injective multiset functions. With the mechanism for modeling universal multiset functions in Lemma 5 as a building block, we can conceive aggregation schemes that can represent universal functions over a node and the multiset of its neighbors, and thus will satisfy the injectiveness condition (a) in Theorem 3. Our next corollary provides a simple and concrete formulation among many such aggregation schemes.

Corollary 6. *Assume $\mathcal{X}$ is countable. There exists a function $f : \mathcal{X} \to \mathbb{R}^n$ so that for infinitely many choices of ϵ, including all irrational numbers, $h(c, X) = (1+\epsilon) \cdot f(c) + \sum_{x \in X} f(x)$ is unique for each pair (c, X), where $c \in \mathcal{X}$ and $X \subset \mathcal{X}$ is a multiset of bounded size. Moreover, any function g over such pairs can be decomposed as $g(c, X) = \varphi\left((1+\epsilon) \cdot f(c) + \sum_{x \in X} f(x)\right)$ for some function φ.*

We can use multi-layer perceptrons (MLPs) to model and learn f and φ in Corollary 6, thanks to the universal approximation theorem (Hornik et al., 1989; Hornik, 1991). In practice, we model $f^{(k+1)} \circ \varphi^{(k)}$ with one MLP, because MLPs can represent the composition of functions. In the first iteration, we do not need MLPs before summation if input features are one-hot encodings as their summation alone is injective. We can make ϵ a learnable parameter or a fixed scalar. Then, GIN updates node representations as

$$h_v^{(k)} = \mathrm{MLP}^{(k)}\left(\left(1+\epsilon^{(k)}\right) \cdot h_v^{(k-1)} + \sum\nolimits_{u \in \mathcal{N}(v)} h_u^{(k-1)}\right). \tag{4.1}$$

Generally, there may exist many other powerful GNNs. GIN is one such example among many maximally powerful GNNs, while being simple.

4.2 Graph-level readout of GIN

Node embeddings learned by GIN can be directly used for tasks like node classification and link prediction. For graph classification tasks we propose the following "readout" function that, given embeddings of individual nodes, produces the embedding of the entire graph.

An important aspect of the graph-level readout is that node representations, corresponding to subtree structures, get more refined and global as the number of iterations increases. A sufficient number of iterations is key to achieving good discriminative power. Yet, features from earlier iterations may sometimes generalize better. To consider all structural information, we use information from all depths/iterations of the model. We achieve this by an architecture similar to Jumping Knowledge

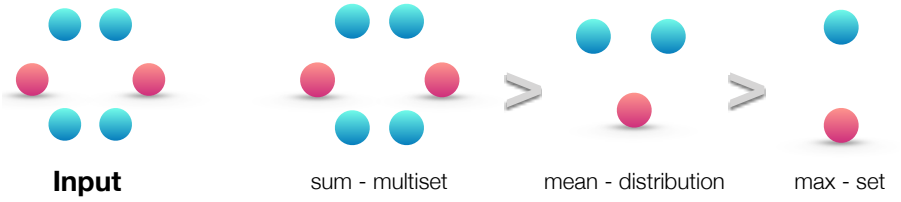
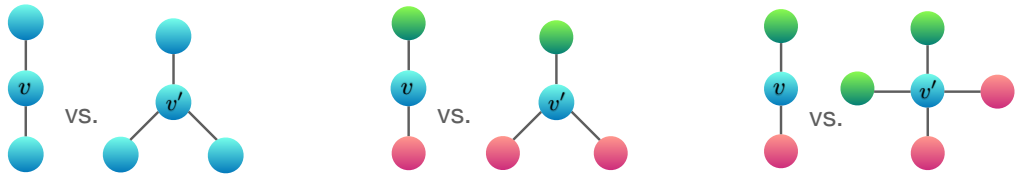


Figure 2: **Ranking by expressive power for sum, mean and max aggregators over a multiset**. Left panel shows the input multiset, *i.e.*, the network neighborhood to be aggregated. The next three panels illustrate the aspects of the multiset a given aggregator is able to capture: sum captures the full multiset, mean captures the proportion/distribution of elements of a given type, and the max aggregator ignores multiplicities (reduces the multiset to a simple set).



(a) Mean and Max both fail (b) Max fails (c) Mean and Max both fail

Figure 3: **Examples of graph structures that mean and max aggregators fail to distinguish.** Between the two graphs, nodes v and v' get the same embedding even though their corresponding graph structures differ. Figure 2 gives reasoning about how different aggregators "compress" different multisets and thus fail to distinguish them.

Networks (Xu et al., 2018), where we replace Eq. 2.4 with graph representations concatenated across *all iterations/layers* of GIN:

$$h_G = \text{CONCAT}\Big(\text{READOUT}\Big(\Big\{h_v^{(k)} | v \in G\Big\}\Big) \mid k = 0, 1, \ldots, K\Big). \tag{4.2}$$

By Theorem 3 and Corollary 6, if GIN replaces READOUT in Eq. 4.2 with summing all node features from the same iterations (we do not need an extra MLP before summation for the same reason as in Eq. 4.1), it provably generalizes the WL test and the WL subtree kernel.

5 Less powerful but still interesting GNNs

Next, we study GNNs that do not satisfy the conditions in Theorem 3, including GCN (Kipf & Welling, 2017) and GraphSAGE (Hamilton et al., 2017a). We conduct ablation studies on two aspects of the aggregator in Eq. 4.1: (1) 1-layer perceptrons instead of MLPs and (2) mean or max-pooling instead of the sum. We will see that these GNN variants get confused by surprisingly simple graphs and are less powerful than the WL test. Nonetheless, models with mean aggregators like GCN perform well for *node classification* tasks. To better understand this, we precisely characterize what different GNN variants can and cannot capture about a graph and discuss the implications for learning with graphs.

5.1 1-layer perceptrons are not sufficient

The function f in Lemma 5 helps map distinct multisets to unique embeddings. It can be parameterized by an MLP by the universal approximation theorem (Hornik, 1991). Nonetheless, many existing GNNs instead use a 1-layer perceptron $\sigma \circ W$ (Duvenaud et al., 2015; Kipf & Welling, 2017; Zhang et al., 2018), a linear mapping followed by a non-linear activation function such as a ReLU. Such 1-layer mappings are examples of Generalized Linear Models (Nelder & Wedderburn, 1972). Therefore, we are interested in understanding whether 1-layer perceptrons are enough for graph learning. Lemma 7 suggests that there are indeed network neighborhoods (multisets) that models with 1-layer perceptrons can never distinguish.

Lemma 7. *There exist finite multisets* $X_1 \neq X_2$ *so that for any linear mapping* W, $\sum_{x \in X_1} \text{ReLU}(Wx) = \sum_{x \in X_2} \text{ReLU}(Wx)$.

The main idea of the proof for Lemma 7 is that 1-layer perceptrons can behave much like linear mappings, so the GNN layers degenerate into simply summing over neighborhood features. Our proof builds on the fact that the bias term is lacking in the linear mapping. With the bias term and sufficiently large output dimensionality, 1-layer perceptrons might be able to distinguish different multisets. Nonetheless, unlike models using MLPs, the 1-layer perceptron (even with the bias term) is *not a universal approximator* of multiset functions. Consequently, even if GNNs with 1-layer perceptrons can embed different graphs to different locations to some degree, such embeddings may not adequately capture structural similarity, and can be difficult for simple classifiers, *e.g.*, linear classifiers, to fit. In Section 7, we will empirically see that GNNs with 1-layer perceptrons, when applied to graph classification, sometimes severely underfit training data and often perform worse than GNNs with MLPs in terms of test accuracy.

5.2 Structures that confuse mean and max-pooling

What happens if we replace the sum in $h\left(X\right) = \sum_{x \in X} f(x)$ with mean or max-pooling as in GCN and GraphSAGE? Mean and max-pooling aggregators are still well-defined multiset functions because they are permutation invariant. But, they are *not* injective. Figure 2 ranks the three aggregators by their representational power, and Figure 3 illustrates pairs of structures that the mean and max-pooling aggregators fail to distinguish. Here, node colors denote different node features, and we assume the GNNs aggregate neighbors first before combining them with the central node labeled as v and v'.

In Figure 3a, every node has the same feature a and $f(a)$ is the same across all nodes (for any function f). When performing neighborhood aggregation, the mean or maximum over $f(a)$ remains $f(a)$ and, by induction, we always obtain the same node representation everywhere. Thus, in this case mean and max-pooling aggregators fail to capture any structural information. In contrast, the sum aggregator distinguishes the structures because $2 \cdot f(a)$ and $3 \cdot f(a)$ give different values. The same argument can be applied to any unlabeled graph. If node degrees instead of a constant value is used as node input features, in principle, mean can recover sum, but max-pooling cannot.

Fig. 3a suggests that mean and max have trouble distinguishing graphs with nodes that have repeating features. Let h_{color} (r for red, g for green) denote node features transformed by f. Fig. 3b shows that maximum over the neighborhood of the blue nodes v and v' yields $\max\left(h_{\text{g}}, h_{\text{r}}\right)$ and $\max\left(h_{\text{g}}, h_{\text{r}}, h_{\text{r}}\right)$, which collapse to the same representation (even though the corresponding graph structures are different). Thus, max-pooling fails to distinguish them. In contrast, the sum aggregator still works because $\frac{1}{2}\left(h_{\text{g}} + h_{\text{r}}\right)$ and $\frac{1}{3}\left(h_{\text{g}} + h_{\text{r}} + h_{\text{r}}\right)$ are in general not equivalent. Similarly, in Fig. 3c, both mean and max fail as $\frac{1}{2}\left(h_{\text{g}} + h_{\text{r}}\right) = \frac{1}{4}\left(h_{\text{g}} + h_{\text{g}} + h_{\text{r}} + h_{\text{r}}\right)$.

5.3 Mean learns distributions

To characterize the class of multisets that the mean aggregator can distinguish, consider the example $X_1 = (S, m)$ and $X_2 = (S, k \cdot m)$, where X_1 and X_2 have the same set of distinct elements, but X_2 contains k copies of each element of X_1. Any mean aggregator maps X_1 and X_2 to the same embedding, because it simply takes averages over individual element features. Thus, the mean captures the *distribution* (proportions) of elements in a multiset, but not the *exact* multiset.

Corollary 8. *Assume $\mathcal{X}$ is countable. There exists a function $f : \mathcal{X} \to \mathbb{R}^n$ so that for $h(X) = \frac{1}{|X|}\sum_{x \in X} f(x)$, $h(X_1) = h(X_2)$ if and only if multisets X_1 and X_2 have the same distribution. That is, assuming $|X_2| \geq |X_1|$, we have $X_1 = (S, m)$ and $X_2 = (S, k \cdot m)$ for some $k \in \mathbb{N}_{\geq 1}$.*

The mean aggregator may perform well if, for the task, the statistical and distributional information in the graph is more important than the exact structure. Moreover, when node features are diverse and rarely repeat, the mean aggregator is as powerful as the sum aggregator. This may explain why, despite the limitations identified in Section 5.2, GNNs with mean aggregators are effective for node classification tasks, such as classifying article subjects and community detection, where node features are rich and the distribution of the neighborhood features provides a strong signal for the task.

5.4 Max-pooling learns sets with distinct elements

The examples in Figure 3 illustrate that max-pooling considers multiple nodes with the same feature as *only one* node (*i.e.*, treats a multiset as a set). Max-pooling captures neither the exact structure nor

the distribution. However, it may be suitable for tasks where it is important to identify representative elements or the "skeleton", rather than to distinguish the exact structure or distribution. Qi et al. (2017) empirically show that the max-pooling aggregator learns to identify the skeleton of a 3D point cloud and that it is robust to noise and outliers. For completeness, the next corollary shows that the max-pooling aggregator captures the underlying set of a multiset.

Corollary 9. *Assume $\mathcal{X}$ is countable. Then there exists a function $f : \mathcal{X} \to \mathbb{R}^\infty$ so that for $h(X) = \max_{x \in X} f(x)$, $h(X_1) = h(X_2)$ if and only if X_1 and X_2 have the same underlying set.*

5.5 Remarks on other aggregators

There are other non-standard neighbor aggregation schemes that we do not cover, *e.g.*, weighted average via attention (Velickovic et al., 2018) and LSTM pooling (Hamilton et al., 2017a; Murphy et al., 2018). We emphasize that our theoretical framework is general enough to characterize the representaional power of any aggregation-based GNNs. In the future, it would be interesting to apply our framework to analyze and understand other aggregation schemes.

6 Other related work

Despite the empirical success of GNNs, there has been relatively little work that mathematically studies their properties. An exception to this is the work of Scarselli et al. (2009a) who shows that the perhaps earliest GNN model (Scarselli et al., 2009b) can approximate measurable functions in probability. Lei et al. (2017) show that their proposed architecture lies in the RKHS of graph kernels, but do not study explicitly which graphs it can distinguish. Each of these works focuses on a specific architecture and do not easily generalize to multple architectures. In contrast, our results above provide a general framework for analyzing and characterizing the expressive power of a broad class of GNNs. Recently, many GNN-based architectures have been proposed, including sum aggregation and MLP encoding (Battaglia et al., 2016; Scarselli et al., 2009b; Duvenaud et al., 2015), and most without theoretical derivation. In contrast to many prior GNN architectures, our Graph Isomorphism Network (GIN) is theoretically motivated, simple yet powerful.

7 Experiments

We evaluate and compare the training and test performance of GIN and less powerful GNN variants.[1] Training set performance allows us to compare different GNN models based on their representational power and test set performance quantifies generalization ability.

Datasets. We use 9 graph classification benchmarks: 4 bioinformatics datasets (MUTAG, PTC, NCI1, PROTEINS) and 5 social network datasets (COLLAB, IMDB-BINARY, IMDB-MULTI, REDDIT-BINARY and REDDIT-MULTI5K) (Yanardag & Vishwanathan, 2015). Importantly, our goal here is not to allow the models to rely on the input node features but mainly learn from the network structure. Thus, in the bioinformatic graphs, the nodes have categorical input features but in the social networks, they have no features. For social networks we create node features as follows: for the REDDIT datasets, we set all node feature vectors to be the same (thus, features here are uninformative); for the other social graphs, we use one-hot encodings of node degrees. Dataset statistics are summarized in Table 1, and more details of the data can be found in Appendix I.

Models and configurations. We evaluate GINs (Eqs. 4.1 and 4.2) and the less powerful GNN variants. Under the GIN framework, we consider two variants: (1) a GIN that learns ϵ in Eq. 4.1 by gradient descent, which we call GIN-ϵ, and (2) a simpler (slightly less powerful)[2] GIN, where ϵ in Eq. 4.1 is fixed to 0, which we call GIN-0. As we will see, GIN-0 shows strong empirical performance: not only does GIN-0 fit training data equally well as GIN-ϵ, it also demonstrates good generalization, slightly but consistently outperforming GIN-ϵ in terms of test accuracy. For the less powerful GNN variants, we consider architectures that replace the sum in the GIN-0 aggregation with mean or max-pooling[3], or replace MLPs with 1-layer perceptrons, *i.e.*, a linear mapping followed

[1] The code is available at `https://github.com/weihua916/powerful-gnns`.

[2] There exist certain (somewhat contrived) graphs that GIN-ϵ can distinguish but GIN-0 cannot.

[3] For REDDIT-BINARY, REDDIT–MULTI5K, and COLLAB, we did not run experiments for max-pooling due to GPU memory constraints.

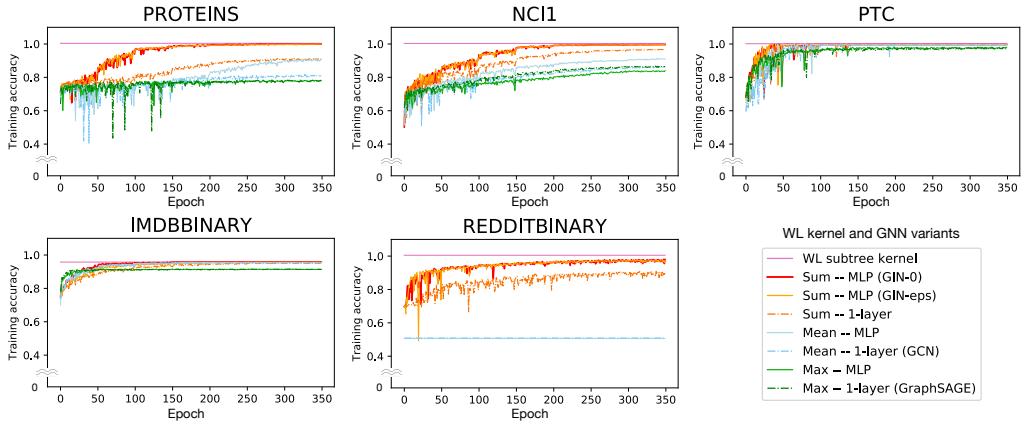


Figure 4: Training set performance of GINs, less powerful GNN variants, and the WL subtree kernel.

by ReLU. In Figure 4 and Table 1, a model is named by the aggregator/perceptron it uses. Here mean–1-layer and max–1-layer correspond to GCN and GraphSAGE, respectively, up to minor architecture modifications. We apply the same graph-level readout (READOUT in Eq. 4.2) for GINs and all the GNN variants, specifically, sum readout on bioinformatics datasets and mean readout on social datasets due to better test performance.

Following (Yanardag & Vishwanathan, 2015; Niepert et al., 2016), we perform 10-fold cross-validation with LIB-SVM (Chang & Lin, 2011). We report the average and standard deviation of validation accuracies across the 10 folds within the cross-validation. For all configurations, 5 GNN layers (including the input layer) are applied, and all MLPs have 2 layers. Batch normalization (Ioffe & Szegedy, 2015) is applied on every hidden layer. We use the Adam optimizer (Kingma & Ba, 2015) with initial learning rate 0.01 and decay the learning rate by 0.5 every 50 epochs. The hyper-parameters we tune for each dataset are: (1) the number of hidden units $\in \{16, 32\}$ for bioinformatics graphs and 64 for social graphs; (2) the batch size $\in \{32, 128\}$; (3) the dropout ratio $\in \{0, 0.5\}$ after the dense layer (Srivastava et al., 2014); (4) the number of epochs, *i.e.*, a single epoch with the best cross-validation accuracy averaged over the 10 folds was selected. Note that due to the small dataset sizes, an alternative setting, where hyper-parameter selection is done using a validation set, is extremely unstable, *e.g.*, for MUTAG, the validation set only contains 18 data points. We also report the training accuracy of different GNNs, where all the hyper-parameters were fixed across the datasets: 5 GNN layers (including the input layer), hidden units of size 64, minibatch of size 128, and 0.5 dropout ratio. For comparison, the training accuracy of the WL subtree kernel is reported, where we set the number of iterations to 4, which is comparable to the 5 GNN layers.

Baselines. We compare the GNNs above with a number of state-of-the-art baselines for graph classification: (1) the WL subtree kernel (Shervashidze et al., 2011), where C-SVM (Chang & Lin, 2011) was used as a classifier; the hyper-parameters we tune are C of the SVM and the number of WL iterations $\in \{1, 2, \ldots, 6\}$; (2) state-of-the-art deep learning architectures, i.e., Diffusion-convolutional neural networks (DCNN) (Atwood & Towsley, 2016), PATCHY-SAN (Niepert et al., 2016) and Deep Graph CNN (DGCNN) (Zhang et al., 2018); (3) Anonymous Walk Embeddings (AWL) (Ivanov & Burnaev, 2018). For the deep learning methods and AWL, we report the accuracies reported in the original papers.

7.1 Results

Training set performance. We validate our theoretical analysis of the representational power of GNNs by comparing their training accuracies. Models with higher representational power should have higher training set accuracy. Figure 4 shows training curves of GINs and less powerful GNN variants with the same hyper-parameter settings. First, both the theoretically most powerful GNN, *i.e.* GIN-ϵ and GIN-0, are able to almost perfectly fit all the training sets. In our experiments, explicit learning of ϵ in GIN-ϵ yields no gain in fitting training data compared to fixing ϵ to 0 as in GIN-0. In comparison, the GNN variants using mean/max pooling or 1-layer perceptrons severely underfit on many datasets. In particular, the training accuracy patterns align with our ranking by the models'

	Datasets	IMDB-B	IMDB-M	RDT-B	RDT-M5K	COLLAB	MUTAG	PROTEINS	PTC	NCI1
Datasets	# graphs	1000	1500	2000	5000	5000	188	1113	344	4110
	# classes	2	3	2	5	3	2	2	2	2
	Avg # nodes	19.8	13.0	429.6	508.5	74.5	17.9	39.1	25.5	29.8
Baselines	WL subtree	73.8 ± 3.9	50.9 ± 3.8	81.0 ± 3.1	52.5 ± 2.1	78.9 ± 1.9	90.4 ± 5.7	75.0 ± 3.1	59.9 ± 4.3	**86.0 ± 1.8** *
	DCNN	49.1	33.5	–	–	52.1	67.0	61.3	56.6	62.6
	PATCHYSAN	71.0 ± 2.2	45.2 ± 2.8	86.3 ± 1.6	49.1 ± 0.7	72.6 ± 2.2	**92.6 ± 4.2** *	75.9 ± 2.8	60.0 ± 4.8	78.6 ± 1.9
	DGCNN	70.0	47.8	–	–	73.7	85.8	75.5	58.6	74.4
	AWL	74.5 ± 5.9	51.5 ± 3.6	87.9 ± 2.5	54.7 ± 2.9	73.9 ± 1.9	87.9 ± 9.8	–	–	–
GNN variants	SUM–MLP (**GIN-0**)	**75.1 ± 5.1**	**52.3 ± 2.8**	**92.4 ± 2.5**	**57.5 ± 1.5**	**80.2 ± 1.9**	**89.4 ± 5.6**	**76.2 ± 2.8**	**64.6 ± 7.0**	**82.7 ± 1.7**
	SUM–MLP (**GIN-ϵ**)	**74.3 ± 5.1**	**52.1 ± 3.6**	**92.2 ± 2.3**	**57.0 ± 1.7**	**80.1 ± 1.9**	**89.0 ± 6.0**	**75.9 ± 3.8**	63.7 ± 8.2	**82.7 ± 1.6**
	SUM–1-LAYER	74.1 ± 5.0	**52.2 ± 2.4**	90.0 ± 2.7	55.1 ± 1.6	**80.6 ± 1.9**	**90.0 ± 8.8**	**76.2 ± 2.6**	63.1 ± 5.7	82.0 ± 1.5
	MEAN–MLP	73.7 ± 3.7	**52.3 ± 3.1**	50.0 ± 0.0	20.0 ± 0.0	79.2 ± 2.3	83.5 ± 6.3	75.5 ± 3.4	**66.6 ± 6.9**	80.9 ± 1.8
	MEAN–1-LAYER (GCN)	74.0 ± 3.4	51.9 ± 3.8	50.0 ± 0.0	20.0 ± 0.0	79.0 ± 1.8	85.6 ± 5.8	76.0 ± 3.2	64.2 ± 4.3	80.2 ± 2.0
	MAX–MLP	73.2 ± 5.8	51.1 ± 3.6	–	–	–	84.0 ± 6.1	76.0 ± 3.2	64.6 ± 10.2	77.8 ± 1.3
	MAX–1-LAYER (GraphSAGE)	72.3 ± 5.3	50.9 ± 2.2	–	–	–	85.1 ± 7.6	75.9 ± 3.2	63.9 ± 7.7	77.7 ± 1.5

Table 1: **Test set classification accuracies (%).** The best-performing GNNs are highlighted with boldface. On datasets where GINs' accuracy is not strictly the highest among GNN variants, we see that GINs are still comparable to the best GNN because a paired t-test at significance level 10% does not distinguish GINs from the best; thus, GINs are also highlighted with boldface. If a baseline performs significantly better than all GNNs, we highlight it with boldface and asterisk.

representational power: GNN variants with MLPs tend to have higher training accuracies than those with 1-layer perceptrons, and GNNs with sum aggregators tend to fit the training sets better than those with mean and max-pooling aggregators.

On our datasets, training accuracies of the GNNs never exceed those of the WL subtree kernel. This is expected because GNNs generally have lower discriminative power than the WL test. For example, on IMDBBINARY, none of the models can perfectly fit the training set, and the GNNs achieve at most the same training accuracy as the WL kernel. This pattern aligns with our result that the WL test provides an upper bound for the representational capacity of the aggregation-based GNNs. However, the WL kernel is not able to learn how to combine node features, which might be quite informative for a given prediction task as we will see next.

Test set performance. Next, we compare test accuracies. Although our theoretical results do not directly speak about the generalization ability of GNNs, it is reasonable to expect that GNNs with strong expressive power can accurately capture graph structures of interest and thus generalize well. Table 1 compares test accuracies of GINs (Sum–MLP), other GNN variants, as well as the state-of-the-art baselines.

First, GINs, especially GIN-0, outperform (or achieve comparable performance as) the less powerful GNN variants on all the 9 datasets, achieving state-of-the-art performance. GINs shine on the social network datasets, which contain a relatively large number of training graphs. For the Reddit datasets, all nodes share the same scalar as node feature. Here, GINs and sum-aggregation GNNs accurately capture the graph structure and significantly outperform other models. Mean-aggregation GNNs, however, fail to capture any structures of the unlabeled graphs (as predicted in Section 5.2) and do not perform better than random guessing. Even if node degrees are provided as input features, mean-based GNNs perform much worse than sum-based GNNs (the accuracy of the GNN with mean–MLP aggregation is 71.2±4.6% on REDDIT-BINARY and 41.3±2.1% on REDDIT-MULTI5K). Comparing GINs (GIN-0 and GIN-ϵ), we observe that GIN-0 slightly but consistently outperforms GIN-ϵ. Since both models fit training data equally well, the better generalization of GIN-0 may be explained by its simplicity compared to GIN-ϵ.

8 CONCLUSION

In this paper, we developed theoretical foundations for reasoning about the expressive power of GNNs, and proved tight bounds on the representational capacity of popular GNN variants. We also designed a provably maximally powerful GNN under the neighborhood aggregation framework. An interesting direction for future work is to go beyond neighborhood aggregation (or message passing) in order to pursue possibly even more powerful architectures for learning with graphs. To complete the picture, it would also be interesting to understand and improve the generalization properties of GNNs as well as better understand their optimization landscape.

ACKNOWLEDGMENTS

This research was supported by NSF CAREER award 1553284, a DARPA D3M award and DARPA DSO's Lagrange program under grant FA86501827838. This research was also supported in part by NSF, ARO MURI, Boeing, Huawei, Stanford Data Science Initiative, and Chan Zuckerberg Biohub. Weihua Hu was supported by Funai Overseas Scholarship. We thank Prof. Ken-ichi Kawarabayashi and Prof. Masashi Sugiyama for supporting this research with computing resources and providing great advice. We thank Tomohiro Sonobe and Kento Nozawa for managing servers. We thank Rex Ying and William Hamilton for helpful feedback. We thank Simon S. Du, Yasuo Tabei, Chengtao Li, and Jingling Li for helpful discussions and positive comments.

REFERENCES

James Atwood and Don Towsley. Diffusion-convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 1993–2001, 2016.

László Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pp. 684–697. ACM, 2016.

László Babai and Ludik Kucera. Canonical labelling of graphs in linear average time. In *Foundations of Computer Science, 1979., 20th Annual Symposium on*, pp. 39–46. IEEE, 1979.

Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. Interaction networks for learning about objects, relations and physics. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 4502–4510, 2016.

Jin-Yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, 1992.

Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27, 2011.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 3844–3852, 2016.

Brendan L Douglas. The weisfeiler-lehman method and graph isomorphism testing. *arXiv preprint arXiv:1101.5211*, 2011.

David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. pp. 2224–2232, 2015.

Sergei Evdokimov and Ilia Ponomarenko. Isomorphism of coloured graphs with slowly increasing multiplicity of jordan blocks. *Combinatorica*, 19(3):321–333, 1999.

Michael R Garey. A guide to the theory of np-completeness. *Computers and intractability*, 1979.

Michael R Garey and David S Johnson. *Computers and intractability*, volume 29. wh freeman New York, 2002.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning (ICML)*, pp. 1273–1272, 2017.

William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 1025–1035, 2017a.

William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *IEEE Data Engineering Bulletin*, 40(3):52–74, 2017b.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2): 251–257, 1991.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning (ICML)*, pp. 448–456, 2015.

Sergey Ivanov and Evgeny Burnaev. Anonymous walk embeddings. In *International Conference on Machine Learning (ICML)*, pp. 2191–2200, 2018.

Steven Kearnes, Kevin McCloskey, Marc Berndl, Vijay Pande, and Patrick Riley. Molecular graph convolutions: moving beyond fingerprints. *Journal of computer-aided molecular design*, 30(8): 595–608, 2016.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

Tao Lei, Wengong Jin, Regina Barzilay, and Tommi Jaakkola. Deriving neural architectures from sequence and graph kernels. pp. 2024–2033, 2017.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. In *International Conference on Learning Representations (ICLR)*, 2016.

Ryan L Murphy, Balasubramaniam Srinivasan, Vinayak Rao, and Bruno Ribeiro. Janossy pooling: Learning deep permutation-invariant functions for variable-size inputs. *arXiv preprint arXiv:1811.01900*, 2018.

J. A. Nelder and R. W. M. Wedderburn. Generalized linear models. *Journal of the Royal Statistical Society, Series A, General*, 135:370–384, 1972.

Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *International Conference on Machine Learning (ICML)*, pp. 2014–2023, 2016.

Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, 1(2):4, 2017.

Adam Santoro, David Raposo, David G Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Timothy Lillicrap. A simple neural network module for relational reasoning. In *Advances in neural information processing systems*, pp. 4967–4976, 2017.

Adam Santoro, Felix Hill, David Barrett, Ari Morcos, and Timothy Lillicrap. Measuring abstract reasoning in neural networks. In *International Conference on Machine Learning*, pp. 4477–4486, 2018.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. Computational capabilities of graph neural networks. *IEEE Transactions on Neural Networks*, 20 (1):81–102, 2009a.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009b.

Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep): 2539–2561, 2011.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.

Saurabh Verma and Zhi-Li Zhang. Graph capsule convolutional neural networks. *arXiv preprint arXiv:1805.08090*, 2018.

Boris Weisfeiler and AA Lehman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9):12–16, 1968.

Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. In *International Conference on Machine Learning (ICML)*, pp. 5453–5462, 2018.

Pinar Yanardag and SVN Vishwanathan. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1365–1374. ACM, 2015.

Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. In *Advances in Neural Information Processing Systems (NIPS)*, 2018.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan R Salakhutdinov, and Alexander J Smola. Deep sets. In *Advances in Neural Information Processing Systems*, pp. 3391–3401, 2017.

Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *AAAI Conference on Artificial Intelligence*, pp. 4438–4445, 2018.

A PROOF FOR LEMMA 2

Proof. Suppose after k iterations, a graph neural network $\mathcal{A}$ has $\mathcal{A}(G_1) \neq \mathcal{A}(G_2)$ but the WL test cannot decide G_1 and G_2 are non-isomorphic. It follows that from iteration 0 to k in the WL test, G_1 and G_2 always have the same collection of node labels. In particular, because G_1 and G_2 have the same WL node labels for iteration i and $i+1$ for any $i = 0, ..., k-1$, G_1 and G_2 have the same collection, i.e. multiset, of WL node labels $\left\{l_v^{(i)}\right\}$ as well as the same collection of node neighborhoods $\left\{\left(l_v^{(i)}, \left\{l_u^{(i)} : u \in \mathcal{N}(v)\right\}\right)\right\}$. Otherwise, the WL test would have obtained different collections of node labels at iteration $i+1$ for G_1 and G_2 as different multisets get unique new labels.

The WL test always relabels different multisets of neighboring nodes into different new labels. We show that on the same graph $G = G_1$ or G_2, if WL node labels $l_v^{(i)} = l_u^{(i)}$, we always have GNN node features $h_v^{(i)} = h_u^{(i)}$ for any iteration i. This apparently holds for $i = 0$ because WL and GNN starts with the same node features. Suppose this holds for iteration j, if for any u, v, $l_v^{(j+1)} = l_u^{(j+1)}$, then it must be the case that

$$\left(l_v^{(j)}, \left\{l_w^{(j)} : w \in \mathcal{N}(v)\right\}\right) = \left(l_u^{(j)}, \left\{l_w^{(j)} : w \in \mathcal{N}(u)\right\}\right)$$

By our assumption on iteration j, we must have

$$\left(h_v^{(j)}, \left\{h_w^{(j)} : w \in \mathcal{N}(v)\right\}\right) = \left(h_u^{(j)}, \left\{h_w^{(j)} : w \in \mathcal{N}(u)\right\}\right)$$

In the aggregation process of the GNN, the same AGGREGATE and COMBINE are applied. The same input, i.e. neighborhood features, generates the same output. Thus, $h_v^{(j+1)} = h_u^{(j+1)}$. By induction, if WL node labels $l_v^{(i)} = l_u^{(i)}$, we always have GNN node features $h_v^{(i)} = h_u^{(i)}$ for any iteration i. This creates a valid mapping ϕ such that $h_v^{(i)} = \phi(l_v^{(i)})$ for any $v \in G$. It follows from G_1 and G_2 have the same multiset of WL neighborhood labels that G_1 and G_2 also have the same collection of GNN neighborhood features

$$\left\{\left(h_v^{(i)}, \left\{h_u^{(i)} : u \in \mathcal{N}(v)\right\}\right)\right\} = \left\{\left(\phi(l_v^{(i)}), \left\{\phi(l_u^{(i)}) : u \in \mathcal{N}(v)\right\}\right)\right\}$$

Thus, $\left\{h_v^{(i+1)}\right\}$ are the same. In particular, we have the same collection of GNN node features $\left\{h_v^{(k)}\right\}$ for G_1 and G_2. Because the graph level readout function is permutation invariant with respect to the collection of node features, $\mathcal{A}(G_1) = \mathcal{A}(G_2)$. Hence we have reached a contradiction. □

B PROOF FOR THEOREM 3

Proof. Let $\mathcal{A}$ be a graph neural network where the condition holds. Let G_1, G_2 be any graphs which the WL test decides as non-isomorphic at iteration K. Because the graph-level readout function is injective, i.e., it maps distinct multiset of node features into unique embeddings, it sufficies to show that $\mathcal{A}$'s neighborhood aggregation process, with sufficient iterations, embeds G_1 and G_2 into different multisets of node features. Let us assume $\mathcal{A}$ updates node representations as

$$h_v^{(k)} = \phi\left(h_v^{(k-1)}, f\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)\right)$$

with injective funtions f and ϕ. The WL test applies a predetermined injective hash function g to update the WL node labels $l_v^{(k)}$:

$$l_v^{(k)} = g\left(l_v^{(k-1)}, \left\{l_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)$$

We will show, by induction, that for any iteration k, there always exists an injective function φ such that $h_v^{(k)} = \varphi\left(l_v^{(k)}\right)$. This apparently holds for $k = 0$ because the initial node features are the same

for WL and GNN $l_v^{(0)} = h_v^{(0)}$ for all $v \in G_1, G_2$. So φ could be the identity function for $k = 0$. Suppose this holds for iteration $k-1$, we show that it also holds for k. Substituting $h_v^{(k-1)}$ with $\varphi\left(l_v^{(k-1)}\right)$ gives us

$$h_v^{(k)} = \phi\left(\varphi\left(l_v^{(k-1)}\right), f\left(\left\{\varphi\left(l_u^{(k-1)}\right) : u \in \mathcal{N}(v)\right\}\right)\right).$$

Since the composition of injective functions is injective, there exists some injective function ψ so that

$$h_v^{(k)} = \psi\left(l_v^{(k-1)}, \left\{l_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right)$$

Then we have

$$h_v^{(k)} = \psi \circ g^{-1} g\left(l_v^{(k-1)}, \left\{l_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right) = \psi \circ g^{-1}\left(l_v^{(k)}\right)$$

$\varphi = \psi \circ g^{-1}$ is injective because the composition of injective functions is injective. Hence for any iteration k, there always exists an injective function φ such that $h_v^{(k)} = \varphi\left(l_v^{(k)}\right)$. At the K-th iteration, the WL test decides that G_1 and G_2 are non-isomorphic, that is the multisets $\left\{l_v^{(K)}\right\}$ are different for G_1 and G_2. The graph neural network $\mathcal{A}$'s node embeddings $\left\{h_v^{(K)}\right\} = \left\{\varphi\left(l_v^{(K)}\right)\right\}$ must also be different for G_1 and G_2 because of the injectivity of φ.

□

C Proof for Lemma 4

Proof. Before proving our lemma, we first show a well-known result that we will later reduce our problem to: $\mathbb{N}^k$ is countable for every $k \in \mathbb{N}$, i.e. finite Cartesian product of countable sets is countable. We observe that it suffices to show $\mathbb{N} \times \mathbb{N}$ is countable, because the proof then follows clearly from induction. To show $\mathbb{N} \times \mathbb{N}$ is countable, we construct a bijection ϕ from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$ as

$$\phi(m, n) = 2^{m-1} \cdot (2n - 1)$$

Now we go back to proving our lemma. If we can show that the range of any function g defined on multisets of bounded size from a countable set is also countable, then the lemma holds for any $g^{(k)}$ by induction. Thus, our goal is to show that the range of such g is countable. First, it is clear that the mapping from $g(X)$ to X is injective because g is a well-defined function. It follows that it suffices to show the set of all multisets $X \subset \mathcal{X}$ is countable.

Since the union of two countable sets is countable, the following set $\mathcal{X}'$ is also countable.

$$\mathcal{X}' = \mathcal{X} \cup \{e\}$$

where e is a dummy element that is not in $\mathcal{X}$. It follows from the result we showed above, *i.e.*, $\mathbb{N}^k$ is countable for every $k \in \mathbb{N}$, that $\mathcal{X}'^k$ is countable for every $k \in \mathbb{N}$. It remains to show there exists an injective mapping from the set of multisets in $\mathcal{X}$ to $\mathcal{X}'^k$ for some $k \in \mathbb{N}$.

We construct an injective mapping h from the set of multisets $X \subset \mathcal{X}$ to $\mathcal{X}'^k$ for some $k \in \mathbb{N}$ as follows. Because $\mathcal{X}$ is countable, there exists a mapping $Z : \mathcal{X} \to \mathbb{N}$ from $x \in \mathcal{X}$ to natural numbers. We can sort the elements $x \in X$ by $z(x)$ as $x_1, x_2, ..., x_n$, where $n = |X|$. Because the multisets X are of bounded size, there exists $k \in \mathbb{N}$ so that $|X| < k$ for all X. We can then define h as

$$h(X) = (x_1, x_2, ..., x_n, e, e, e...),$$

where the $k - n$ coordinates are filled with the dummy element e. It is clear that h is injective because for any multisets X and Y of bounded size, $h(X) = h(Y)$ only if X is equivalent to Y. Hence it follows that the range of g is countable as desired.

□

D PROOF FOR LEMMA 5

Proof. We first prove that there exists a mapping f so that $\sum_{x \in X} f(x)$ is unique for each multiset X of bounded size. Because $\mathcal{X}$ is countable, there exists a mapping $Z : \mathcal{X} \to \mathbb{N}$ from $x \in \mathcal{X}$ to natural numbers. Because the cardinality of multisets X is bounded, there exists a number $N \in \mathbb{N}$ so that $|X| < N$ for all X. Then an example of such f is $f(x) = N^{-Z(x)}$. This f can be viewed as a more compressed form of an one-hot vector or N-digit presentation. Thus, $h(X) = \sum_{x \in X} f(x)$ is an injective function of multisets.

$\phi\left(\sum_{x \in X} f(x)\right)$ is permutation invariant so it is a well-defined multiset function. For any multiset function g, we can construct such ϕ by letting $\phi\left(\sum_{x \in X} f(x)\right) = g(X)$. Note that such ϕ is well-defined because $h(X) = \sum_{x \in X} f(x)$ is injective.

□

E PROOF OF COROLLARY 6

Proof. Following the proof of Lemma 5, we consider $f(x) = N^{-Z(x)}$, where N and $Z : \mathcal{X} \to \mathbb{N}$ are the same as defined in Appendix D. Let $h(c, X) \equiv (1 + \epsilon) \cdot f(c) + \sum_{x \in X} f(x)$. Our goal is show that for any $(c', X') \neq (c, X)$ with $c, c' \in \mathcal{X}$ and $X, X' \subset \mathcal{X}$, $h(c, X) \neq h(c', X')$ holds, if ϵ is an irrational number. We prove by contradiction. For any (c, X), suppose there exists (c', X') such that $(c', X') \neq (c, X)$ but $h(c, X) = h(c', X')$ holds. Let us consider the following two cases: (1) $c' = c$ but $X' \neq X$, and (2) $c' \neq c$. For the first case, $h(c, X) = h(c, X')$ implies $\sum_{x \in X} f(x) = \sum_{x \in X'} f(x)$. It follows from Lemma 5 that the equality will not hold, because with $f(x) = N^{-Z(x)}$, $X' \neq X$ implies $\sum_{x \in X} f(x) \neq \sum_{x \in X'} f(x)$. Thus, we reach a contradiction. For the second case, we can similarly rewrite $h(c, X) = h(c', X')$ as

$$\epsilon \cdot (f(c) - f(c')) = \left(f(c') + \sum_{x \in X'} f(x)\right) - \left(f(c) + \sum_{x \in X} f(x)\right). \tag{E.1}$$

Because ϵ is an irrational number and $f(c) - f(c')$ is a non-zero rational number, L.H.S. of Eq. E.1 is irrational. On the other hand, R.H.S. of Eq. E.1, the sum of a finite number of rational numbers, is rational. Hence the equality in Eq. E.1 cannot hold, and we have reached a contradiction.

For any function g over the pairs (c, X), we can construct such φ for the desired decomposition by letting $\varphi\left((1 + \epsilon) \cdot f(c) + \sum_{x \in X} f(x)\right) = g(c, X)$. Note that such φ is well-defined because $h(c, X) = (1 + \epsilon) \cdot f(c) + \sum_{x \in X} f(x)$ is injective. □

F PROOF FOR LEMMA 7

Proof. Let us consider the example $X_1 = \{1, 1, 1, 1, 1\}$ and $X_2 = \{2, 3\}$, *i.e.* two different multisets of positive numbers that sum up to the same value. We will be using the homogeneity of ReLU.

Let W be an arbitrary linear transform that maps $x \in X_1, X_2$ into $\mathbb{R}^n$. It is clear that, at the same coordinates, Wx are either positive or negative for all x because all x in X_1 and X_2 are positive. It follows that $\text{ReLU}(Wx)$ are either positive or 0 at the same coordinate for all x in X_1, X_2. For the coordinates where $\text{ReLU}(Wx)$ are 0, we have $\sum_{x \in X_1} \text{ReLU}(Wx) = \sum_{x \in X_2} \text{ReLU}(Wx)$. For the coordinates where Wx are positive, linearity still holds. It follows from linearity that

$$\sum_{x \in X} \text{ReLU}(Wx) = \text{ReLU}\left(W \sum_{x \in X} x\right)$$

where X could be X_1 or X_2. Because $\sum_{x \in X_1} x = \sum_{x \in X_2} x$, we have the following as desired.

$$\sum_{x \in X_1} \text{ReLU}(Wx) = \sum_{x \in X_2} \text{ReLU}(Wx)$$

□

G PROOF FOR COROLLARY 8

Proof. Suppose multisets X_1 and X_2 have the same distribution, without loss of generality, let us assume $X_1 = (S, m)$ and $X_2 = (S, k \cdot m)$ for some $k \in \mathbb{N}_{\geq 1}$, i.e. X_1 and X_2 have the same underlying set and the multiplicity of each element in X_2 is k times of that in X_1. Then we have $|X_2| = k|X_1|$ and $\sum_{x \in X_2} f(x) = k \cdot \sum_{x \in X_1} f(x)$. Thus,

$$\frac{1}{|X_2|} \sum_{x \in X_2} f(x) = \frac{1}{k \cdot |X_1|} \cdot k \cdot \sum_{x \in X_1} f(x) = \frac{1}{|X_1|} \sum_{x \in X_1} f(x)$$

Now we show that there exists a function f so that $\frac{1}{|X|} \sum_{x \in X} f(x)$ is unique for distributionally equivalent X. Because $\mathcal{X}$ is countable, there exists a mapping $Z : \mathcal{X} \to \mathbb{N}$ from $x \in \mathcal{X}$ to natural numbers. Because the cardinality of multisets X is bounded, there exists a number $N \in \mathbb{N}$ so that $|X| < N$ for all X. Then an example of such f is $f(x) = N^{-2Z(x)}$. □

H PROOF FOR COROLLARY 9

Proof. Suppose multisets X_1 and X_2 have the same underlying set S, then we have

$$\max_{x \in X_1} f(x) = \max_{x \in S} f(x) = \max_{x \in X_2} f(x)$$

Now we show that there exists a mapping f so that $\max_{x \in X} f(x)$ is unique for Xs with the same underlying set. Because $\mathcal{X}$ is countable, there exists a mapping $Z : \mathcal{X} \to \mathbb{N}$ from $x \in \mathcal{X}$ to natural numbers. Then an example of such $f : \mathcal{X} \to \mathbb{R}^\infty$ is defined as $f_i(x) = 1$ for $i = Z(x)$ and $f_i(x) = 0$ otherwise, where $f_i(x)$ is the i-th coordinate of $f(x)$. Such an f essentially maps a multiset to its one-hot embedding. □

I DETAILS OF DATASETS

We give detailed descriptions of datasets used in our experiments. Further details can be found in (Yanardag & Vishwanathan, 2015).

Social networks datasets. IMDB-BINARY and IMDB-MULTI are movie collaboration datasets. Each graph corresponds to an ego-network for each actor/actress, where nodes correspond to actors/actresses and an edge is drawn betwen two actors/actresses if they appear in the same movie. Each graph is derived from a pre-specified genre of movies, and the task is to classify the genre graph it is derived from. REDDIT-BINARY and REDDIT-MULTI5K are balanced datasets where each graph corresponds to an online discussion thread and nodes correspond to users. An edge was drawn between two nodes if at least one of them responded to another's comment. The task is to classify each graph to a community or a subreddit it belongs to. COLLAB is a scientific collaboration dataset, derived from 3 public collaboration datasets, namely, High Energy Physics, Condensed Matter Physics and Astro Physics. Each graph corresponds to an ego-network of different researchers from each field. The task is to classify each graph to a field the corresponding researcher belongs to.

Bioinformatics datasets. MUTAG is a dataset of 188 mutagenic aromatic and heteroaromatic nitro compounds with 7 discrete labels. PROTEINS is a dataset where nodes are secondary structure elements (SSEs) and there is an edge between two nodes if they are neighbors in the amino-acid sequence or in 3D space. It has 3 discrete labels, representing helix, sheet or turn. PTC is a dataset of 344 chemical compounds that reports the carcinogenicity for male and female rats and it has 19 discrete labels. NCI1 is a dataset made publicly available by the National Cancer Institute (NCI) and is a subset of balanced datasets of chemical compounds screened for ability to suppress or inhibit the growth of a panel of human tumor cell lines, having 37 discrete labels.

Provably Powerful Graph Networks

Haggai Maron* **Heli Ben-Hamu*** **Hadar Serviansky*** **Yaron Lipman**
Weizmann Institute of Science
Rehovot, Israel

Abstract

Recently, the Weisfeiler-Lehman (WL) graph isomorphism test was used to measure the expressive power of graph neural networks (GNN). It was shown that the popular message passing GNN cannot distinguish between graphs that are indistinguishable by the 1-WL test (Morris et al., 2018; Xu et al., 2019). Unfortunately, many simple instances of graphs are indistinguishable by the 1-WL test.

In search for more expressive graph learning models we build upon the recent k-order invariant and equivariant graph neural networks (Maron et al., 2019a;b) and present two results:

First, we show that such k-order networks can distinguish between non-isomorphic graphs as good as the k-WL tests, which are provably stronger than the 1-WL test for $k > 2$. This makes these models strictly stronger than message passing models. Unfortunately, the higher expressiveness of these models comes with a computational cost of processing high order tensors.

Second, setting our goal at building a provably stronger, *simple* and *scalable* model we show that a reduced 2-order network containing just scaled identity operator, augmented with a single quadratic operation (matrix multiplication) has a provable 3-WL expressive power. Differently put, we suggest a simple model that interleaves applications of standard Multilayer-Perceptron (MLP) applied to the feature dimension and matrix multiplication. We validate this model by presenting state of the art results on popular graph classification and regression tasks. To the best of our knowledge, this is the first practical invariant/equivariant model with guaranteed 3-WL expressiveness, strictly stronger than message passing models.

1 Introduction

Graphs are an important data modality which is frequently used in many fields of science and engineering. Among other things, graphs are used to model social networks, chemical compounds, biological structures and high-level image content information. One of the major tasks in graph data analysis is learning from graph data. As classical approaches often use hand-crafted graph features that are not necessarily suitable to all datasets and/or tasks (e.g., Kriege et al. (2019)), a significant research effort in recent years is to develop deep models that are able to learn new graph representations from raw features (e.g., Gori et al. (2005); Duvenaud et al. (2015); Niepert et al. (2016); Kipf and Welling (2016); Veličković et al. (2017); Monti et al. (2017); Hamilton et al. (2017a); Morris et al. (2018); Xu et al. (2019)).

Currently, the most popular methods for deep learning on graphs are *message passing neural networks* in which the node features are propagated through the graph according to its connectivity structure (Gilmer et al., 2017). In a successful attempt to quantify the expressive power of message passing models, Morris et al. (2018); Xu et al. (2019) suggest to compare the model's ability to *distinguish* between two given graphs to that of the hierarchy of the Weisfeiler-Lehman (WL) graph isomorphism

*Equal contribution

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

tests (Grohe, 2017; Babai, 2016). Remarkably, they show that the class of message passing models has limited expressiveness and is not better than the first WL test (1-WL, a.k.a. color refinement). For example, Figure 1 depicts two graphs (i.e., in blue and in green) that 1-WL cannot distinguish, hence indistinguishable by any message passing algorithm.

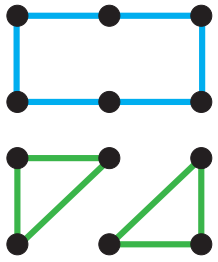


Figure 1: Two graphs not distinguished by 1-WL.

The goal of this work is to explore and develop GNN models that possess higher expressiveness while maintaining scalability, as much as possible. We present two main contributions. First, establishing a baseline for expressive GNNs, we prove that the recent k-order invariant GNNs (Maron et al., 2019a;b) offer a natural hierarchy of models that are as expressive as the k-WL tests, for $k \geq 2$. Second, as k-order GNNs are not practical for $k > 2$ we develop a simple, novel GNN model, that incorporates standard MLPs of the feature dimension and a matrix multiplication layer. This model, working only with $k = 2$ tensors (the same dimension as the graph input data), possesses the expressiveness of 3-WL. Since, in the WL hierarchy, 1-WL and 2-WL are equivalent, while 3-WL is strictly stronger, this model is provably more powerful than the message passing models. For example, it can distinguish the two graphs in Figure 1. As far as we know, this model is the first to offer both expressiveness (3-WL) and scalability ($k = 2$).

The main challenge in achieving high-order WL expressiveness with GNN models stems from the difficulty to represent the multisets of neighborhoods required for the WL algorithms. We advocate a novel representation of multisets based on Power-sum Multi-symmetric Polynomials (PMP) which are a generalization of the well-known elementary symmetric polynomials. This representation provides a convenient theoretical tool to analyze models' ability to implement the WL tests.

A related work to ours that also tried to build graph learning methods that surpass the 1-WL expressiveness offered by message passing is Morris et al. (2018). They develop powerful deep models generalizing message passing to higher orders that are as expressive as higher order WL tests. Although making progress, their full model is still computationally prohibitive for 3-WL expressiveness and requires a relaxed local version compromising some of the theoretical guarantees.

Experimenting with our model on several real-world datasets that include classification and regression tasks on social networks, molecules, and chemical compounds, we found it to be on par or better than state of the art.

2 Previous work

Deep learning on graph data. The pioneering works that applied neural networks to graphs are Gori et al. (2005); Scarselli et al. (2009) that learn node representations using recurrent neural networks, which were also used in Li et al. (2015). Following the success of convolutional neural networks (Krizhevsky et al., 2012), many works have tried to generalize the notion of convolution to graphs and build networks that are based on this operation. Bruna et al. (2013) defined graph convolutions as operators that are diagonal in the graph laplacian eigenbasis. This paper resulted in multiple follow up works with more efficient and spatially localized convolutions (Henaff et al., 2015; Defferrard et al., 2016; Kipf and Welling, 2016; Levie et al., 2017). Other works define graph convolutions as local stationary functions that are applied to each node and its neighbours (e.g., Duvenaud et al. (2015); Atwood and Towsley (2016); Niepert et al. (2016); Hamilton et al. (2017b); Veličković et al. (2017); Monti et al. (2018)). Many of these works were shown to be instances of the family of message passing neural networks (Gilmer et al., 2017): methods that apply parametric functions to a node and its neighborhood and then apply some pooling operation in order to generate a new feature for each node. In a recent line of work, it was suggested to define graph neural networks using permutation equivariant operators on tensors describing k-order relations between the nodes. Kondor et al. (2018) identified several such linear and quadratic equivariant operators and showed that the resulting network can achieve excellent results on popular graph learning benchmarks. Maron et al. (2019a) provided a full characterization of linear equivariant operators between tensors of arbitrary order. In both cases, the resulting networks were shown to be at least as powerful as message passing neural networks. In another line of work, Murphy et al. (2019) suggest expressive invariant graph models defined using averaging over all permutations of an arbitrary base neural network.

Weisfeiler Lehman graph isomorphism test. The Weisfeiler Lehman tests is a hierarchy of increasingly powerful graph isomorphism tests (Grohe, 2017). The WL tests have found many applications in machine learning: in addition to Xu et al. (2019); Morris et al. (2018), this idea was used in Shervashidze et al. (2011) to construct a graph kernel method, which was further generalized to higher order WL tests in Morris et al. (2017). Lei et al. (2017) showed that their suggested GNN has a theoretical connection to the WL test. WL tests were also used in Zhang and Chen (2017) for link prediction tasks. In a concurrent work, Morris and Mutzel (2019) suggest constructing graph features based on an equivalent sparse version of high-order WL achieving great speedup and expressiveness guarantees for sparsely connected graphs.

3 Preliminaries

We denote a set by $\{a, b, \ldots, c\}$, an ordered set (tuple) by $(a, b, \ldots, c)$ and a multiset (i.e., a set with possibly repeating elements) by $\{\!\{a, b, \ldots, c\}\!\}$. We denote $[n] = \{1, 2, \ldots, n\}$, and $(a_i \mid i \in [n]) = (a_1, a_2, \ldots, a_n)$. Let S_n denote the permutation group on n elements. We use multi-index $\boldsymbol{i} \in [n]^k$ to denote a k-tuple of indices, $\boldsymbol{i} = (i_1, i_2, \ldots, i_k)$. $g \in S_n$ acts on multi-indices $\boldsymbol{i} \in [n]^k$ entrywise by $g(\boldsymbol{i}) = (g(i_1), g(i_2), \ldots, g(i_k))$. S_n acts on k-tensors $\mathbf{X} \in \mathbb{R}^{n^k \times a}$ by $(g \cdot \mathbf{X})_{\boldsymbol{i},j} = \mathbf{X}_{g^{-1}(\boldsymbol{i}),j}$, where $\boldsymbol{i} \in [n]^k$, $j \in [a]$.

3.1 k-order graph networks

Maron et al. (2019a) have suggested a family of permutation-invariant deep neural network models for graphs. Their main idea is to construct networks by concatenating maximally expressive linear equivariant layers. More formally, a k-order invariant graph network is a composition $F = m \circ h \circ L_d \circ \sigma \circ \cdots \circ \sigma \circ L_1$, where $L_i : \mathbb{R}^{n^{k_i} \times a_i} \to \mathbb{R}^{n^{k_{i+1}} \times a_{i+1}}$, $\max_{i \in [d+1]} k_i = k$, are *equivariant linear layers*, namely satisfy

$$L_i(g \cdot \mathbf{X}) = g \cdot L_i(\mathbf{X}), \qquad \forall g \in S_n, \quad \forall \mathbf{X} \in \mathbb{R}^{n^{k_i} \times a_i},$$

σ is an entrywise non-linear activation, $\sigma(\mathbf{X})_{\boldsymbol{i},j} = \sigma(\mathbf{X}_{\boldsymbol{i},j})$, $h : \mathbb{R}^{n^{k_{d+1}} \times a_{d+1}} \to \mathbb{R}^{a_{d+2}}$ is an *invariant linear layer*, namely satisfies

$$h(g \cdot \mathbf{X}) = h(\mathbf{X}), \qquad \forall g \in S_n, \quad \forall \mathbf{X} \in \mathbb{R}^{n^{k_{d+1}} \times a_{d+1}},$$

and m is a Multilayer Perceptron (MLP). The invariance of F is achieved by construction (by propagating g through the layers using the definitions of equivariance and invariance):

$$F(g \cdot \mathbf{X}) = m(\cdots(L_1(g \cdot \mathbf{X}))\cdots) = m(\cdots(g \cdot L_1(\mathbf{X}))\cdots) = \cdots = m(h(g \cdot L_d(\cdots))) = F(\mathbf{X}).$$

When $k = 2$, Maron et al. (2019a) proved that this construction gives rise to a model that can approximate any message passing neural network (Gilmer et al., 2017) to an arbitrary precision; Maron et al. (2019b) proved these models are universal for a very high tensor order of $k = poly(n)$, which is of little practical value (an alternative proof was recently suggested in Keriven and Peyré (2019)).

3.2 The Weisfeiler-Lehman graph isomorphism test

Let $G = (V, E, d)$ be a colored graph where $|V| = n$ and $d : V \to \Sigma$ defines the color attached to each vertex in V, Σ is a set of colors. The Weisfeiler-Lehman (WL) test is a family of algorithms used to test graph isomorphism. Two graphs G, G' are called isomorphic if there exists an edge and color preserving bijection $\phi : V \to V'$.

There are two families of WL algorithms: k-WL and k-FWL (Folklore WL), both parameterized by $k = 1, 2, \ldots, n$. k-WL and k-FWL both construct a coloring of k-tuples of vertices, that is $c : V^k \to \Sigma$. Testing isomorphism of two graphs G, G' is then performed by comparing the histograms of colors produced by the k-WL (or k-FWL) algorithms.

We will represent coloring of k-tuples using a tensor $\mathbf{C} \in \Sigma^{n^k}$, where $\mathbf{C}_{\boldsymbol{i}} \in \Sigma$, $\boldsymbol{i} \in [n]^k$ denotes the color of the k-tuple $v_{\boldsymbol{i}} = (v_{i_1}, \ldots, v_{i_k}) \in V^k$. In both algorithms, the initial coloring $\mathbf{C}^0$ is defined using the *isomorphism type* of each k-tuple. That is, two k-tuples $\boldsymbol{i}, \boldsymbol{i}'$ have the same isomorphism type (i.e., get the same color, $\mathbf{C}_{\boldsymbol{i}} = \mathbf{C}_{\boldsymbol{i}'}$) if for all $q, r \in [k]$: (i) $v_{i_q} = v_{i_r} \iff v_{i'_q} = v_{i'_r}$; (ii) $d(v_{i_q}) = d(v_{i'_q})$; and (iii) $(v_{i_r}, v_{i_q}) \in E \iff (v_{i'_r}, v_{i'_q}) \in E$. Clearly, if G, G' are two isomorphic graphs then there exists $g \in S_n$ so that $g \cdot \mathbf{C}'^0 = \mathbf{C}^0$.

In the next steps, the algorithms refine the colorings $\mathbf{C}^l$, $l = 1, 2, \ldots$ until the coloring does not change further, that is, the subsets of k-tuples with same colors do not get further split to different color groups. It is guaranteed that no more than $l = poly(n)$ iterations are required (Douglas, 2011).

The construction of $\mathbf{C}^l$ from $\mathbf{C}^{l-1}$ differs in the WL and FWL versions. The difference is in how the colors are aggregated from neighboring k-tuples. We define two notions of neighborhoods of a k-tuple $\boldsymbol{i} \in [n]^k$:

$$N_j(\boldsymbol{i}) = \left\{ (i_1, \ldots, i_{j-1}, i', i_{j+1}, \ldots, i_k) \;\middle|\; i' \in [n] \right\} \tag{1}$$

$$N_j^F(\boldsymbol{i}) = \Big((j, i_2, \ldots, i_k), (i_1, j, \ldots, i_k), \ldots, (i_1, \ldots, i_{k-1}, j) \Big) \tag{2}$$

$N_j(\boldsymbol{i})$, $j \in [k]$ is the j-th neighborhood of the tuple $\boldsymbol{i}$ used by the WL algorithm, while $N_j^F(\boldsymbol{i})$, $j \in [n]$ is the j-th neighborhood used by the FWL algorithm. Note that $N_j(\boldsymbol{i})$ is a set of n k-tuples, while $N_j^F(\boldsymbol{i})$ is an ordered set of k k-tuples. The inset to the right illustrates these notions of neighborhoods for the case $k = 2$: the top figure shows $N_1(3, 2)$ in purple and $N_2(3, 2)$ in orange. The bottom figure shows $N_j^F(3, 2)$ for all $j = 1, \ldots, n$ with different colors for different j.

The coloring update rules are:

$$\text{WL:} \quad \mathbf{C}_{\boldsymbol{i}}^l = \text{enc}\Big(\mathbf{C}_{\boldsymbol{i}}^{l-1}, \Big(\{\!\{ \mathbf{C}_{\boldsymbol{j}}^{l-1} \mid \boldsymbol{j} \in N_j(\boldsymbol{i}) \}\!\} \;\Big|\; j \in [k] \Big) \Big) \tag{3}$$

$$\text{FWL:} \quad \mathbf{C}_{\boldsymbol{i}}^l = \text{enc}\Big(\mathbf{C}_{\boldsymbol{i}}^{l-1}, \Big\{\!\!\Big\{ \big(\mathbf{C}_{\boldsymbol{j}}^{l-1} \mid \boldsymbol{j} \in N_j^F(\boldsymbol{i}) \big) \;\Big|\; j \in [n] \Big\}\!\!\Big\} \Big) \tag{4}$$

where enc is a bijective map from the collection of all possible tuples in the r.h.s. of Equations (3)-(4) to Σ.

When $k = 1$ both rules, (3)-(4), degenerate to $\mathbf{C}_i^l = \text{enc}\Big(\mathbf{C}_i^{l-1}, \{\!\{ \mathbf{C}_j^{l-1} \mid j \in [n] \}\!\} \Big)$, which will not refine any initial color. Traditionally, the first algorithm in the WL hierarchy is called WL, 1-WL, or the *color refinement algorithm*. In color refinement, one starts with the coloring prescribed with d. Then, in each iteration, the color at each vertex is refined by a new color representing its current color and the multiset of its neighbors' colors.

Several known results of WL and FWL algorithms (Cai et al., 1992; Grohe, 2017; Morris et al., 2018; Grohe and Otto, 2015) are:

1. 1-WL and 2-WL have equivalent discrimination power.
2. k-FWL is equivalent to $(k+1)$-WL for $k \geq 2$.
3. For each $k \geq 2$ there is a pair of non-isomorphic graphs distinguishable by $(k+1)$-WL but not by k-WL.

4 Colors and multisets in networks

Before we get to the two main contributions of this paper we address three challenges that arise when analyzing networks' ability to implement WL-like algorithms: (i) Representing the colors Σ in the network; (ii) implementing a multiset representation; and (iii) implementing the encoding function.

Color representation. We will represent colors as vectors. That is, we will use tensors $\mathbf{C} \in \mathbb{R}^{n^k \times a}$ to encode a color per k-tuple; that is, the color of the tuple $\boldsymbol{i} \in [n]^k$ is a vector $\mathbf{C}_{\boldsymbol{i}} \in \mathbb{R}^a$. This effectively replaces the color tensors Σ^{n^k} in the WL algorithm with $\mathbb{R}^{n^k \times a}$.

Multiset representation. A key technical part of our method is the way we encode multisets in networks. Since colors are represented as vectors in $\mathbb{R}^a$, an n-tuple of colors is represented by a matrix $\boldsymbol{X} = [x_1, x_2, \ldots, x_n]^T \in \mathbb{R}^{n \times a}$, where $x_j \in \mathbb{R}^a$, $j \in [n]$ are the rows of $\boldsymbol{X}$. Thinking about $\boldsymbol{X}$ as a multiset forces us to be indifferent to the order of rows. That is, the color representing $g \cdot \boldsymbol{X}$ should be the same as the color representing $\boldsymbol{X}$, for all $g \in S_n$. One possible approach is to perform some sort (e.g., lexicographic) to the rows of $\boldsymbol{X}$. Unfortunately, this seems challenging to implement with equivariant layers.

Instead, we suggest to encode a multiset $\boldsymbol{X}$ using a set of S_n-invariant functions called the *Power-sum Multi-symmetric Polynomials* (PMP) (Briand, 2004; Rydh, 2007). The PMP are the multivariate

analog to the more widely known *Power-sum Symmetric Polynomials*, $p_j(y) = \sum_{i=1}^{n} y_i^j$, $j \in [n]$, where $y \in \mathbb{R}^n$. They are defined next. Let $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_a) \in [n]^a$ be a multi-index and for $y \in \mathbb{R}^a$ we set $y^{\boldsymbol{\alpha}} = y_1^{\alpha_1} \cdot y_2^{\alpha_2} \cdots y_a^{\alpha_a}$. Furthermore, $|\boldsymbol{\alpha}| = \sum_{j=1}^{a} \alpha_j$. The PMP of degree $\boldsymbol{\alpha} \in [n]^a$ is

$$p_{\boldsymbol{\alpha}}(\boldsymbol{X}) = \sum_{i=1}^{n} x_i^{\boldsymbol{\alpha}}, \quad \boldsymbol{X} \in \mathbb{R}^{n \times a}.$$

A key property of the PMP is that the finite subset $p_{\boldsymbol{\alpha}}$, for $|\boldsymbol{\alpha}| \leq n$ generates the ring of *Multi-symmetric Polynomials* (MP), the set of polynomials q so that $q(g \cdot \boldsymbol{X}) = q(\boldsymbol{X})$ for all $g \in S_n$, $\boldsymbol{X} \in \mathbb{R}^{n \times a}$ (see, e.g., (Rydh, 2007) corollary 8.4). The PMP generates the ring of MP in the sense that for an arbitrary MP q, there exists a polynomial r so that $q(\boldsymbol{X}) = r\left(u(\boldsymbol{X})\right)$, where

$$u(\boldsymbol{X}) := \left(p_{\boldsymbol{\alpha}}(\boldsymbol{X}) \;\middle|\; |\boldsymbol{\alpha}| \leq n\right). \tag{5}$$

As the following proposition shows, a useful consequence of this property is that the vector $u(\boldsymbol{X})$ is a unique representation of the multi-set $\boldsymbol{X} \in \mathbb{R}^{n \times a}$.

Proposition 1. *For arbitrary $\boldsymbol{X}, \boldsymbol{X}' \in \mathbb{R}^{n \times a}$: $\exists g \in S_n$ so that $\boldsymbol{X}' = g \cdot \boldsymbol{X}$ if and only if $u(\boldsymbol{X}) = u(\boldsymbol{X}')$.*

We note that Proposition 1 is a generalization of lemma 6 in Zaheer et al. (2017) to the case of multisets of vectors. This generalization was possible since the PMP provide a continuous way to encode *vector* multisets (as opposed to scalar multisets in previous works). The full proof is provided in the supplementary material.

Encoding function. One of the benefits in the vector representation of colors is that the encoding function can be implemented as a simple concatenation: Given two color tensors $\mathbf{C} \in \mathbb{R}^{n^k \times a}$, $\mathbf{C}' \in \mathbb{R}^{n^k \times b}$, the tensor that represents for each k-tuple $\boldsymbol{i}$ the color pair $(\mathbf{C}_{\boldsymbol{i}}, \mathbf{C}'_{\boldsymbol{i}})$ is simply $(\mathbf{C}, \mathbf{C}') \in \mathbb{R}^{n^k \times (a+b)}$.

5 k-order graph networks are as powerful as k-WL

Our goal in this section is to show that, for every $2 \leq k \leq n$, k-order graph networks (Maron et al., 2019a) are at least as powerful as the k-WL graph isomorphism test in terms of distinguishing non-isomorphic graphs. This result is shown by constructing a k-order network model and learnable weight assignment that implements the k-WL test.

To motivate this construction we note that the WL update step, Equation 3, is equivariant (see proof in the supplementary material). Namely, plugging in $g \cdot \mathbf{C}^{l-1}$ the WL update step would yield $g \cdot \mathbf{C}^{l}$. Therefore, it is plausible to try to implement the WL update step using linear equivariant layers and non-linear pointwise activations.

Theorem 1. *Given two graphs $G = (V, E, d)$, $G' = (V', E', d')$ that can be distinguished by the k-WL graph isomorphism test, there exists a k-order network F so that $F(G) \neq F(G')$. On the other direction for every two isomorphic graphs $G \cong G'$ and k-order network F, $F(G) = F(G')$.*

The full proof is provided in the supplementary material. Here we outline the basic idea for the proof. First, an input graph $G = (V, E, d)$ is represented using a tensor of the form $\mathbf{B} \in \mathbb{R}^{n^2 \times (e+1)}$, as follows. The last channel of $\mathbf{B}$, namely $\mathbf{B}_{:,:,e+1}$ (':' stands for all possible values $[n]$) encodes the adjacency matrix of G according to E. The first e channels $\mathbf{B}_{:,:,1:e}$ are zero outside the diagonal, and $\mathbf{B}_{i,i,1:e} = d(v_i) \in \mathbb{R}^e$ is the color of vertex $v_i \in V$.

Now, the second statement in Theorem 1 is clear since two isomorphic graphs G, G' will have tensor representations satisfying $\mathbf{B}' = g \cdot \mathbf{B}$ and therefore, as explained in Section 3.1, $F(\mathbf{B}) = F(\mathbf{B}')$.

More challenging is showing the other direction, namely that for non-isomorphic graphs G, G' that can be distinguished by the k-WL test, there exists a k-network distinguishing G and G'. The key idea is to show that a k-order network can encode the multisets $\{\!\{\mathbf{B}_{\boldsymbol{j}} \mid \boldsymbol{j} \in N_j(\boldsymbol{i})\}\!\}$ for a given tensor $\mathbf{B} \in \mathbb{R}^{n^k \times a}$. These multisets are the only non-trivial component in the WL update rule, Equation 3. Note that the rows of the matrix $\boldsymbol{X} = \mathbf{B}_{i_1,\ldots,i_{j-1},:,i_{j+1},\ldots,i_k,:} \in \mathbb{R}^{n \times a}$ are the colors (i.e., vectors)

that define the multiset $\{\mathbf{B}_{\boldsymbol{j}} \mid \boldsymbol{j} \in N_j(\boldsymbol{i})\}$. Following our multiset representation (Section 4) we would like the network to compute $u(\boldsymbol{X})$ and plug the result at the $\boldsymbol{i}$-th entry of an output tensor $\mathbf{C}$.

This can be done in two steps: First, applying the polynomial function $\tau : \mathbb{R}^a \to \mathbb{R}^b$, $b = \binom{n+a-1}{a-1}$ entrywise to $\mathbf{B}$, where τ is defined by $\tau(x) = (x^{\boldsymbol{\alpha}} \mid |\boldsymbol{\alpha}| \leq n)$ (note that b is the number of multi-indices $\boldsymbol{\alpha}$ such that $|\boldsymbol{\alpha}| \leq n$). Denote the output of this step $\mathbf{Y}$. Second, apply a linear equivariant operator summing over the j-the coordinate of $\mathbf{Y}$ to get $\mathbf{C}$, that is

$$\mathbf{C}_{\boldsymbol{i},:} := L_j(\mathbf{Y})_{\boldsymbol{i},:} = \sum_{i'=1}^{n} \mathbf{Y}_{i_1,\cdots,i_{j-1},i',i_{j+1},\ldots,i_k,:} = \sum_{\boldsymbol{j} \in N_j(\boldsymbol{i})} \tau(\mathbf{B}_{\boldsymbol{j},:}) = u(\boldsymbol{X}), \quad \boldsymbol{i} \in [n]^k,$$

where $\boldsymbol{X} = \mathbf{B}_{i_1,\ldots,i_{j-1},:,i_{j+1},\ldots,i_k,:}$ as desired. Lastly, we use the universal approximation theorem (Cybenko, 1989; Hornik, 1991) to replace the polynomial function τ with an approximating MLP $m : \mathbb{R}^a \to \mathbb{R}^b$ to get a k-order network (details are in the supplementary material). Applying m feature-wise, that is $m(\mathbf{B})_{\boldsymbol{i},:} = m(\mathbf{B}_{\boldsymbol{i},:})$, is in particular a k-order network in the sense of Section 3.1.

6 A simple network with 3-WL discrimination power

In this section we describe a simple GNN model that has 3-WL discrimination power. The model has the form

$$F = m \circ h \circ B_d \circ B_{d-1} \cdots \circ B_1, \tag{6}$$

where as in k-order networks (see Section 3.1) h is an invariant layer and m is an MLP. $B_1, \ldots, B_d$ are blocks with the following structure (see figure 2 for an illustration). Let $\mathbf{X} \in \mathbb{R}^{n \times n \times a}$ denote the input tensor to the block. First, we apply three MLPs $m_1, m_2 : \mathbb{R}^a \to \mathbb{R}^b$, $m_3 : \mathbb{R}^a \to \mathbb{R}^{b'}$ to the input tensor, $m_l(\mathbf{X})$, $l \in [3]$. This means applying the MLP to each feature of the input tensor independently, i.e., $m_l(\mathbf{X})_{i_1,i_2,:} := m_l(\mathbf{X}_{i_1,i_2,:})$, $l \in [3]$. Second, matrix multiplication is performed between matching features, i.e., $\mathbf{W}_{:,:,j} := m_1(\mathbf{X})_{:,:,j} \cdot m_2(\mathbf{X})_{:,:,j}$, $j \in [b]$. The output of the block is the tensor $(m_3(\mathbf{X}), \mathbf{W})$.

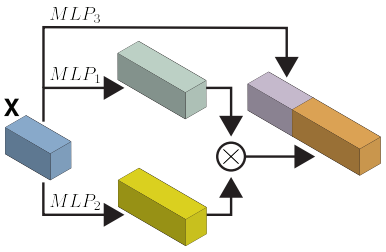


Figure 2: Block structure.

We start with showing our basic requirement from GNN, namely invariance:

Lemma 1. *The model F described above is invariant, i.e., $F(g \cdot \mathbf{B}) = F(\mathbf{B})$, for all $g \in S_n$, and $\mathbf{B}$.*

Proof. Note that matrix multiplication is equivariant: for two matrices $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{n \times n}$ and $g \in S_n$ one has $(g \cdot \boldsymbol{A}) \cdot (g \cdot \boldsymbol{B}) = g \cdot (\boldsymbol{A} \cdot \boldsymbol{B})$. This makes the basic building block B_i equivariant, and consequently the model F invariant, i.e., $F(g \cdot \mathbf{B}) = F(\mathbf{B})$. □

Before we prove the 3-WL power for this model, let us provide some intuition as to why matrix multiplication improves expressiveness. Let us show matrix multiplication allows this model to distinguish between the two graphs in Figure 1, which are 1-WL indistinguishable. The input tensor $\mathbf{B}$ representing a graph G holds the adjacency matrix at the last channel $\boldsymbol{A} := \mathbf{B}_{:,:,e+1}$. We can build a network with 2 blocks computing $\boldsymbol{A}^3$ and then take the trace of this matrix (using the invariant layer h). Remember that the d-th power of the adjacency matrix computes the number of d-paths between vertices; in particular $\mathrm{tr}(\boldsymbol{A}^3)$ computes the number of cycles of length 3. Counting shows the upper graph in Figure 1 has 0 such cycles while the bottom graph has 12. The main result of this section is:

Theorem 2. *Given two graphs $G = (V, E, d)$, $G' = (V', E', d')$ that can be distinguished by the 3-WL graph isomorphism test, there exists a network F (equation 6) so that $F(G) \neq F(G')$. On the other direction for every two isomorphic graphs $G \cong G'$ and F (Equation 6), $F(G) = F(G')$.*

The full proof is provided in the supplementary material. Here we outline the main idea of the proof. The second part of this theorem is already shown in Lemma 1. To prove the first part, namely that the model in Equation 6 has 3-WL expressiveness, we show it can implement the 2-FWL algorithm, that is known to be equivalent to 3-WL (see Section 3.2). As before, the challenge is in implementing the neighborhood multisets as used in the 2-FWL algorithm. That is, given an input tensor $\mathbf{B} \in \mathbb{R}^{n^2 \times a}$ we would like to compute an output tensor $\mathbf{C} \in \mathbb{R}^{n^2 \times b}$ where $\mathbf{C}_{i_1,i_2,:} \in \mathbb{R}^b$ represents a color matching

the multiset $\{\!\{(\mathbf{B}_{j,i_2,:}, \mathbf{B}_{i_1,j,:}) \mid j \in [n]\}\!\}$. As before, we use the multiset representation introduced in section 4. Consider the matrix $\boldsymbol{X} \in \mathbb{R}^{n \times 2a}$ defined by

$$\boldsymbol{X}_{j,:} = (\mathbf{B}_{j,i_2,:}, \mathbf{B}_{i_1,j,:}), \quad j \in [n]. \tag{7}$$

Our goal is to compute an output tensor $\mathbf{W} \in \mathbb{R}^{n^2 \times b}$, where $\mathbf{W}_{i_1,i_2,:} = u(\boldsymbol{X})$.

Consider the multi-index set $\left\{\boldsymbol{\alpha} \mid \boldsymbol{\alpha} \in [n]^{2a}, |\boldsymbol{\alpha}| \leq n\right\}$ of cardinality $b = \binom{n+2a-1}{2a-1}$, and write it in the form $\{(\boldsymbol{\beta}_l, \boldsymbol{\gamma}_l) \mid \boldsymbol{\beta}, \boldsymbol{\gamma} \in [n]^a, |\boldsymbol{\beta}_l| + |\boldsymbol{\gamma}_l| \leq n, l \in b\}$.

Now define polynomial maps $\tau_1, \tau_2 : \mathbb{R}^a \to \mathbb{R}^b$ by $\tau_1(x) = (x^{\boldsymbol{\beta}_l} \mid l \in [b])$, and $\tau_2(x) = (x^{\boldsymbol{\gamma}_l} \mid l \in [b])$. We apply τ_1 to the features of $\mathbf{B}$, namely $\mathbf{Y}_{i_1,i_2,l} := \tau_1(\mathbf{B})_{i_1,i_2,l} = (\mathbf{B}_{i_1,i_2,:})^{\boldsymbol{\beta}_l}$; similarly, $\mathbf{Z}_{i_1,i_2,l} := \tau_2(\mathbf{B})_{i_1,i_2,l} = (\mathbf{B}_{i_1,i_2,:})^{\boldsymbol{\gamma}_l}$. Now,

$$\mathbf{W}_{i_1,i_2,l} := (\mathbf{Z}_{:,:,l} \cdot \mathbf{Y}_{:,:,l})_{i_1,i_2} = \sum_{j=1}^{n} \mathbf{Z}_{i_1,j,l} \mathbf{Y}_{j,i_2,l} = \sum_{j=1}^{n} \mathbf{B}^{\boldsymbol{\beta}_l}_{j,i_2,:} \mathbf{B}^{\boldsymbol{\gamma}_l}_{i_1,j,:} = \sum_{j=1}^{n} (\mathbf{B}_{j,i_2,:}, \mathbf{B}_{i_1,j,:})^{(\boldsymbol{\beta}_l, \boldsymbol{\gamma}_l)},$$

hence $\mathbf{W}_{i_1,i_2,:} = u(\boldsymbol{X})$, where $\boldsymbol{X}$ is defined in Equation 7. To get an implementation with the model in Equation 6 we need to replace τ_1, τ_2 with MLPs. We use the universal approximation theorem to that end (details are in the supplementary material).

To conclude, each update step of the 2-FWL algorithm is implemented in the form of a block B_i applying m_1, m_2 to the input tensor $\mathbf{B}$, followed by matrix multiplication of matching features, $\mathbf{W} = m_1(\mathbf{B}) \cdot m_2(\mathbf{B})$. Since Equation 4 requires pairing the multiset with the input color of each k-tuple, we take m_3 to be identity and get $(\mathbf{B}, \mathbf{W})$ as the block output.

Generalization to k-FWL. One possible extension is to add a generalized matrix multiplication to k-order networks to make them as expressive as k-FWL and hence $(k+1)$-WL. Generalized matrix multiplication is defined as follows. Given $\mathbf{A}^1, \ldots, \mathbf{A}^k \in \mathbb{R}^{n^k}$, then $(\odot_{i=1}^k \mathbf{A}^i)_{\boldsymbol{i}} = \sum_{j=1}^n \mathbf{A}^1_{j,i_2,\ldots,i_k} \mathbf{A}^2_{i_1,j,\ldots,i_k} \cdots \mathbf{A}^k_{i_1,\ldots,i_{k-1},j}$.

Relation to (Morris et al., 2018). Our model offers two benefits over the 1-2-3-GNN suggested in the work of Morris et al. (2018), a recently suggested GNN that also surpasses the expressiveness of message passing networks. First, it has lower space complexity (see details below). This allows us to work with a provably 3-WL expressive model while Morris et al. (2018) resorted to a local 3-GNN version, hindering their 3-WL expressive power. Second, from a practical point of view our model is arguably simpler to implement as it only consists of fully connected layers and matrix multiplication (without having to account for all subsets of size 3).

Complexity analysis of a single block. Assuming a graph with n nodes, dense edge data and a constant feature depth, the layer proposed in Morris et al. (2018) has $O(n^3)$ space complexity (number of subsets) and $O(n^4)$ time complexity ($O(n^3)$ subsets with $O(n)$ neighbors each). Our layer (block), however, has $O(n^2)$ space complexity as only second order tensors are stored (i.e., linear in the size of the graph data), and time complexity of $O(n^3)$ due to the matrix multiplication. We note that the time complexity of Morris et al. (2018) can probably be improved to $O(n^3)$ while our time complexity can be improved to $O(n^{2.x})$ due to more advanced matrix multiplication algorithms.

7 Experiments

Implementation details. We implemented the GNN model as described in Section 6 (see Equation 6) using the TensorFlow framework (Abadi et al., 2016). We used three identical blocks B_1, B_2, B_3, where in each block $B_i : \mathbb{R}^{n^2 \times a} \to \mathbb{R}^{n^2 \times b}$ we took $m_3(x) = x$ to be the identity (i.e., m_3 acts as a skip connection, similar to its role in the proof of Theorem 2); $m_1, m_2 : \mathbb{R}^a \to \mathbb{R}^b$ are chosen as d layer MLP with hidden layers of b features. After each block B_i we also added a single layer MLP $m_4 : \mathbb{R}^{b+a} \to \mathbb{R}^b$. Note that although this fourth MLP is not described in the model in Section 6 it clearly does not decrease (nor increase) the theoretical expressiveness of the model; we found it efficient for coding as it reduces the parameters of the model. For the first block, B_1, $a = e + 1$, where for the other blocks $b = a$. The MLPs are implemented with 1×1 convolutions.

Table 1: Graph Classification Results on the datasets from Yanardag and Vishwanathan (2015)

dataset	MUTAG	PTC	PROTEINS	NCI1	NCI109	COLLAB	IMDB-B	IMDB-M
size	188	344	1113	4110	4127	5000	1000	1500
classes	2	2	2	2	2	3	2	3
avg node #	17.9	25.5	39.1	29.8	29.6	74.4	19.7	13
Results								
GK (Shervashidze et al., 2009)	81.39±1.7	55.65±0.5	71.39±0.3	62.49±0.3	62.35±0.3	NA	NA	NA
RW (Vishwanathan et al., 2010)	79.17±2.1	55.91±0.3	59.57±0.1	> 3 days	NA	NA	NA	NA
PK (Neumann et al., 2016)	76±2.7	59.5±2.4	73.68±0.7	82.54±0.5	NA	NA	NA	NA
WL (Shervashidze et al., 2011)	84.11±1.9	57.97±2.5	**74.68±0.5**	**84.46±0.5**	**85.12±0.3**	NA	NA	NA
FGSD (Verma and Zhang, 2017)	**92.12**	**62.80**	73.42	79.80	78.84	**80.02**	73.62	**52.41**
AWE-DD (Ivanov and Burnaev, 2018)	NA	NA	NA	NA	NA	73.93±1.9	**74.45 ± 5.8**	51.54 ±3.6
AWE-FB (Ivanov and Burnaev, 2018)	87.87±9.7	NA	NA	NA	NA	70.99 ± 1.4	73.13 ±3.2	51.58 ± 4.6
DGCNN (Zhang et al., 2018)	85.83±1.7	58.59±2.5	75.54±0.9	74.44±0.5	NA	73.76±0.5	70.03±0.9	47.83±0.9
PSCN (Niepert et al., 2016) (k=10)	88.95±4.4	62.29±5.7	75±2.5	76.34±1.7	NA	72.6±2.2	71±2.3	45.23±2.8
DCNN (Atwood and Towsley, 2016)	NA	NA	61.29±1.6	56.61± 1.0	NA	52.11±0.7	49.06±1.4	33.49±1.4
ECC (Simonovsky and Komodakis, 2017)	76.11	NA	NA	76.82	75.03	NA	NA	NA
DGK (Yanardag and Vishwanathan, 2015)	87.44±2.7	60.08±2.6	75.68±0.5	80.31±0.5	80.32±0.3	73.09±0.3	66.96±0.6	44.55±0.5
DiffPool (Ying et al., 2018)	NA	NA	**78.1**	NA	NA	75.5	NA	NA
CCN (Kondor et al., 2018)	**91.64±7.2**	**70.62±7.0**	NA	76.27±4.1	75.54±3.4	NA	NA	NA
Invariant Graph Networks (Maron et al., 2019a)	83.89±12.95	58.53±6.86	76.58±5.49	74.33±2.71	72.82±1.45	78.36±2.47	72.0±5.54	48.73±3.41
GIN (Xu et al., 2019)	89.4±5.6	64.6±7.0	76.2±2.8	82.7±1.7	NA	80.2±1.9	**75.1±5.1**	**52.3±2.8**
1-2-3 GNN (Morris et al., 2018)	86.1±	60.9±	75.5±	76.2±	NA	NA	74.2±	49.5±
Ours 1	90.55±8.7	66.17±6.54	77.2±4.73	**83.19±1.11**	**81.84±1.85**	80.16±1.11	72.6±4.9	50±3.15
Ours 2	88.88±7.4	64.7±7.46	76.39±5.03	81.21±2.14	**81.77±1.26**	**81.38±1.42**	72.2±4.26	44.73±7.89
Ours 3	89.44±8.05	62.94±6.96	76.66±5.59	80.97±1.91	**82.23±1.42**	**80.68±1.71**	73±5.77	50.46±3.59
Rank	3^{rd}	2^{nd}	2^{nd}	2^{nd}	2^{nd}	1^{st}	6^{th}	5^{th}

Parameter search was conducted on learning rate and learning rate decay, as detailed below. We have experimented with two network suffixes adopted from previous papers: (i) The suffix used in Maron et al. (2019a) that consists of an invariant max pooling (diagonal and off-diagonal) followed by a three Fully Connected (FC) with hidden units' sizes of $(512, 256, \#\text{classes})$; (ii) the suffix used in Xu et al. (2019) adapted to our network: we apply the invariant max layer from Maron et al. (2019a) to the output of every block followed by a single fully connected layer to #classes. These outputs are then summed together and used as the network output on which the loss function is defined.

Table 2: Regression, the QM9 dataset.

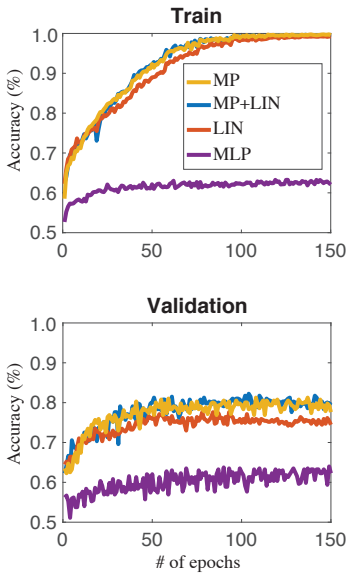
Target	DTNN	MPNN	123-gnn	Ours 1	Ours 2
μ	0.244	0.358	0.476	**0.231**	**0.0934**
α	0.95	0.89	**0.27**	0.382	0.318
ϵ_{homo}	0.00388	0.00541	0.00337	**0.00276**	**0.00174**
ϵ_{lumo}	0.00512	0.00623	0.00351	**0.00287**	**0.0021**
Δ_ϵ	0.0112	0.0066	0.0048	**0.00406**	**0.0029**
$\langle R^2 \rangle$	17	28.5	22.9	**16.07**	**3.78**
$ZPVE$	0.00172	0.00216	**0.00019**	0.00064	0.000399
U_0	-	-	0.0427	0.234	**0.022**
U	-	-	0.111	0.234	**0.0504**
H	-	-	0.0419	0.229	**0.0294**
G	-	-	0.0469	0.238	**0.024**
C_v	0.27	0.42	**0.0944**	0.184	0.144

Datasets. We evaluated our network on two different tasks: Graph classification and graph regression. For classification, we tested our method on eight real-world graph datasets from (Yanardag and Vishwanathan, 2015): three datasets consist of social network graphs, and the other five datasets come from bioinformatics and represent chemical compounds or protein structures. Each graph is represented by an adjacency matrix and possibly categorical node features (for the bioinformatics datasets). For the regression task, we conducted an experiment on a standard graph learning benchmark called the QM9 dataset (Ramakrishnan et al., 2014; Wu et al., 2018). It is composed of 134K small organic molecules (sizes vary from 4 to 29 atoms). Each molecule is represented by an adjacency matrix, a distance matrix (between atoms), categorical data on the edges, and node features; the data was obtained from the pytorch-geometric library (Fey and Lenssen, 2019). The task is to predict 12 real valued physical quantities for each molecule.

Graph classification results. We follow the standard 10-fold cross validation protocol and splits from Zhang et al. (2018) and report our results according to the protocol described in Xu et al. (2019), namely the best averaged accuracy across the 10-folds. Parameter search was conducted on a fixed random 90%-10% split: learning rate in $\left\{5 \cdot 10^{-5}, 10^{-4}, 5 \cdot 10^{-4}, 10^{-3}\right\}$; learning rate decay in $[0.5, 1]$ every 20 epochs. We have tested three architectures: (1) $b = 400$, $d = 2$, and suffix (ii); (2) $b = 400$, $d = 2$, and suffix (i); and (3) $b = 256$, $d = 3$, and suffix (ii). (See above for definitions of b, d and suffix). Table 1 presents a summary of the results (top part - non deep learning methods). The last row presents our ranking compared to all previous methods; note that we have scored in the top 3 methods in 6 out of 8 datasets.

Graph regression results. The data is randomly split into 80% train, 10% validation and 10% test. We have conducted the same parameter search as in the previous experiment on the validation set. We have used the network (2) from classification experiment, i.e., $b = 400$, $d = 2$, and suffix (i), with an absolute error loss adapted to the regression task. Test results are according to the best validation error. We have tried two different settings: (1) training a single network to predict all the output quantities together and (2) training a different network for each quantity. Table 2 compares the mean absolute error of our method with three other methods: 123-gnn (Morris et al., 2018) and (Wu et al., 2018); results of all previous work were taken from (Morris et al., 2018). Note that our method achieves the lowest error on 5 out of the 12 quantities when using a single network, and the lowest error on 9 out of the 12 quantities in case each quantity is predicted by an independent network.

Equivariant layer evaluation. The model in Section 6 does not incorporate all equivariant linear layers as characterized in (Maron et al., 2019a). It is therefore of interest to compare this model to models richer in linear equivariant layers, as well as a simple MLP baseline (i.e., without matrix multiplication). We performed such an experiment on the NCI1 dataset (Yanardag and Vishwanathan, 2015) comparing: (i) our suggested model, denoted Matrix Product (MP); (ii) matrix product + full linear basis from (Maron et al., 2019a) (MP+LIN); (iii) only full linear basis (LIN); and (iv) MLP applied to the feature dimension.



Due to the memory limitation in (Maron et al., 2019a) we used the same feature depths of $b_1 = 32, b_2 = 64, b_3 = 256$, and $d = 2$. The inset shows the performance of all methods on both training and validation sets, where we performed a parameter search on the learning rate (as above) for a fixed decay rate of 0.75 every 20 epochs. Although all methods (excluding MLP) are able to achieve a zero training error, the (MP) and (MP+LIN) enjoy better generalization than the linear basis of Maron et al. (2019a). Note that (MP) and (MP+LIN) are comparable, however (MP) is considerably more efficient.

8 Conclusions

We explored two models for graph neural networks that possess superior graph distinction abilities compared to existing models. First, we proved that k-order invariant networks offer a hierarchy of neural networks that parallels the distinction power of the k-WL tests. This model has lesser practical interest due to the high dimensional tensors it uses. Second, we suggested a simple GNN model consisting of only MLPs augmented with matrix multiplication and proved it achieves 3-WL expressiveness. This model operates on input tensors of size n^2 and therefore useful for problems with dense edge data. The downside is that its complexity is still quadratic, worse than message passing type methods. An interesting future work is to search for more efficient GNN models with high expressiveness. Another interesting research venue is quantifying the generalization ability of these models.

Acknowledgments

This research was supported in part by the European Research Council (ERC Consolidator Grant, "LiftMatch" 771136) and the Israel Science Foundation (Grant No. 1830/17).

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283.

Atwood, J. and Towsley, D. (2016). Diffusion-convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1993–2001.

Babai, L. (2016). Graph isomorphism in quasipolynomial time. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 684–697. ACM.

Briand, E. (2004). When is the algebra of multisymmetric polynomials generated by the elementary multisymmetric polynomials. *Contributions to Algebra and Geometry*, 45(2):353–368.

Bruna, J., Zaremba, W., Szlam, A., and LeCun, Y. (2013). Spectral Networks and Locally Connected Networks on Graphs. pages 1–14.

Cai, J.-Y., Fürer, M., and Immerman, N. (1992). An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314.

Defferrard, M., Bresson, X., and Vandergheynst, P. (2016). Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pages 3844–3852.

Douglas, B. L. (2011). The weisfeiler-lehman method and graph isomorphism testing. *arXiv preprint arXiv:1101.5211*.

Duvenaud, D. K., Maclaurin, D., Iparraguirre, J., Bombarell, R., Hirzel, T., Aspuru-Guzik, A., and Adams, R. P. (2015). Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, pages 2224–2232.

Fey, M. and Lenssen, J. E. (2019). Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. (2017). Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pages 1263–1272.

Gori, M., Monfardini, G., and Scarselli, F. (2005). A new model for earning in raph domains. *Proceedings of the International Joint Conference on Neural Networks*, 2(January):729–734.

Grohe, M. (2017). *Descriptive complexity, canonisation, and definable graph structure theory*, volume 47. Cambridge University Press.

Grohe, M. and Otto, M. (2015). Pebble games and linear equations. *The Journal of Symbolic Logic*, 80(3):797–844.

Hamilton, W., Ying, Z., and Leskovec, J. (2017a). Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034.

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

Henaff, M., Bruna, J., and LeCun, Y. (2015). Deep Convolutional Networks on Graph-Structured Data. (June).

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257.

Ivanov, S. and Burnaev, E. (2018). Anonymous walk embeddings. *arXiv preprint arXiv:1805.11921*.

Keriven, N. and Peyré, G. (2019). Universal invariant and equivariant graph neural networks. *CoRR*, abs/1905.04943.

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

Kondor, R., Son, H. T., Pan, H., Anderson, B., and Trivedi, S. (2018). Covariant compositional networks for learning graphs. *arXiv preprint arXiv:1801.02144*.

Kriege, N. M., Johansson, F. D., and Morris, C. (2019). A survey on graph kernels.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Lei, T., Jin, W., Barzilay, R., and Jaakkola, T. (2017). Deriving neural architectures from sequence and graph kernels. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2024–2033. JMLR. org.

Levie, R., Monti, F., Bresson, X., and Bronstein, M. M. (2017). CayleyNets: Graph Convolutional Neural Networks with Complex Rational Spectral Filters. pages 1–12.

Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. (2015). Gated Graph Sequence Neural Networks. (1):1–20.

Maron, H., Ben-Hamu, H., Shamir, N., and Lipman, Y. (2019a). Invariant and equivariant graph networks. In *International Conference on Learning Representations*.

Maron, H., Fetaya, E., Segol, N., and Lipman, Y. (2019b). On the universality of invariant networks. In *International conference on machine learning*.

Monti, F., Boscaini, D., Masci, J., Rodola, E., Svoboda, J., and Bronstein, M. M. (2017). Geometric deep learning on graphs and manifolds using mixture model cnns. In *Proc. CVPR*, volume 1, page 3.

Monti, F., Shchur, O., Bojchevski, A., Litany, O., Günnemann, S., and Bronstein, M. M. (2018). Dual-Primal Graph Convolutional Networks. pages 1–11.

Morris, C., Kersting, K., and Mutzel, P. (2017). Glocalized weisfeiler-lehman graph kernels: Global-local feature maps of graphs. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 327–336. IEEE.

Morris, C. and Mutzel, P. (2019). Towards a practical k-dimensional weisfeiler-leman algorithm. *arXiv preprint arXiv:1904.01543*.

Morris, C., Ritzert, M., Fey, M., Hamilton, W. L., Lenssen, J. E., Rattan, G., and Grohe, M. (2018). Weisfeiler and leman go neural: Higher-order graph neural networks. *arXiv preprint arXiv:1810.02244*.

Murphy, R. L., Srinivasan, B., Rao, V., and Ribeiro, B. (2019). Relational pooling for graph representations. *arXiv preprint arXiv:1903.02541*.

Neumann, M., Garnett, R., Bauckhage, C., and Kersting, K. (2016). Propagation kernels: efficient graph kernels from propagated information. *Machine Learning*, 102(2):209–245.

Niepert, M., Ahmed, M., and Kutzkov, K. (2016). Learning Convolutional Neural Networks for Graphs.

Ramakrishnan, R., Dral, P. O., Rupp, M., and Von Lilienfeld, O. A. (2014). Quantum chemistry structures and properties of 134 kilo molecules. *Scientific data*, 1:140022.

Rydh, D. (2007). A minimal set of generators for the ring of multisymmetric functions. In *Annales de l'institut Fourier*, volume 57, pages 1741–1769.

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

Shervashidze, N., Schweitzer, P., Leeuwen, E. J. v., Mehlhorn, K., and Borgwardt, K. M. (2011). Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep):2539–2561.

Shervashidze, N., Vishwanathan, S., Petri, T., Mehlhorn, K., and Borgwardt, K. (2009). Efficient graphlet kernels for large graph comparison. In *Artificial Intelligence and Statistics*, pages 488–495.

Simonovsky, M. and Komodakis, N. (2017). Dynamic edge-conditioned filters in convolutional neural networks on graphs. In *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. (2017). Graph Attention Networks. pages 1–12.

Verma, S. and Zhang, Z.-L. (2017). Hunt for the unique, stable, sparse and fast feature learning on graphs. In *Advances in Neural Information Processing Systems*, pages 88–98.

Vishwanathan, S. V. N., Schraudolph, N. N., Kondor, R., and Borgwardt, K. M. (2010). Graph kernels. *Journal of Machine Learning Research*, 11(Apr):1201–1242.

Wu, Z., Ramsundar, B., Feinberg, E. N., Gomes, J., Geniesse, C., Pappu, A. S., Leswing, K., and Pande, V. (2018). Moleculenet: a benchmark for molecular machine learning. *Chemical science*, 9(2):513–530.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. (2019). How powerful are graph neural networks? In *International Conference on Learning Representations*.

Yanardag, P. and Vishwanathan, S. (2015). Deep Graph Kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '15*.

Ying, R., You, J., Morris, C., Ren, X., , W. L., and Leskovec, J. (2018). Hierarchical Graph Representation Learning with Differentiable Pooling.

Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R. R., and Smola, A. J. (2017). Deep sets. In *Advances in Neural Information Processing Systems*, pages 3391–3401.

Zhang, M. and Chen, Y. (2017). Weisfeiler-lehman neural machine for link prediction. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 575–583. ACM.

Zhang, M., Cui, Z., Neumann, M., and Chen, Y. (2018). An end-to-end deep learning architecture for graph classification. In *Proceedings of AAAI Conference on Artificial Inteligence*.

PREDICT THEN PROPAGATE: GRAPH NEURAL NETWORKS MEET PERSONALIZED PAGERANK

Johannes Gasteiger, Aleksandar Bojchevski & Stephan Günnemann
Technical University of Munich, Germany
{j.gasteiger,a.bojchevski,guennemann}@in.tum.de

ABSTRACT

Neural message passing algorithms for semi-supervised classification on graphs have recently achieved great success. However, for classifying a node these methods only consider nodes that are a few propagation steps away and the size of this utilized neighborhood is hard to extend. In this paper, we use the relationship between graph convolutional networks (GCN) and PageRank to derive an improved propagation scheme based on personalized PageRank. We utilize this propagation procedure to construct a simple model, personalized propagation of neural predictions (PPNP), and its fast approximation, APPNP. Our model's training time is on par or faster and its number of parameters on par or lower than previous models. It leverages a large, adjustable neighborhood for classification and can be easily combined with any neural network. We show that this model outperforms several recently proposed methods for semi-supervised classification in the most thorough study done so far for GCN-like models. Our implementation is available online. [1]

1 INTRODUCTION

Graphs are ubiquitous in the real world and its description through scientific models. They are used to study the spread of information, to optimize delivery, to recommend new books, to suggest friends, or to find a party's potential voters. Deep learning approaches have achieved great success on many important graph problems such as link prediction (Grover & Leskovec, 2016; Bojchevski et al., 2018), graph classification (Duvenaud et al., 2015; Niepert et al., 2016; Gilmer et al., 2017) and semi-supervised node classification (Yang et al., 2016; Kipf & Welling, 2017).

There are many approaches for leveraging deep learning algorithms on graphs. Node embedding methods use random walks or matrix factorization to directly train individual node embeddings, often without using node features and usually in an unsupervised manner, i.e. without leveraging node classes (Perozzi et al., 2014; Tang et al., 2015; Nandanwar & Murty, 2016; Grover & Leskovec, 2016; Qiu et al., 2018). Many other approaches use both graph structure and node features in a supervised setting. Examples for these include spectral graph convolutional neural networks (Bruna et al., 2014; Defferrard et al., 2016), message passing (or neighbor aggregation) algorithms (Kearnes et al., 2016; Kipf & Welling, 2017; Hamilton et al., 2017; Pham et al., 2017; Monti et al., 2017; Gilmer et al., 2017), and neighbor aggregation via recurrent neural networks (Scarselli et al., 2009; Li et al., 2016; Dai et al., 2018). Among these categories, the class of message passing algorithms has garnered particular attention recently due to its flexibility and good performance.

Several works have been aimed at improving the basic neighborhood aggregation scheme by using attention mechanisms (Kearnes et al., 2016; Hamilton et al., 2017; Veličković et al., 2018), random walks (Abu-El-Haija et al., 2018a; Ying et al., 2018; Li et al., 2018), edge features (Kearnes et al., 2016; Gilmer et al., 2017; Schlichtkrull et al., 2018) and making it more scalable on large graphs (Chen et al., 2018; Ying et al., 2018). However, all of these methods only use the information of a very limited neighborhood for each node. A larger neighborhood would be desirable to provide the model with more information, especially for nodes in the periphery or in a sparsely labelled setting.

Increasing the size of the neighborhood used by these algorithms, i.e. their range, is not trivial since neighborhood aggregation in this scheme is essentially a type of Laplacian smoothing and too

[1] https://www.kdd.in.tum.de/ppnp

many layers lead to oversmoothing (Li et al., 2018). Xu et al. (2018) highlighted the same problem by establishing a relationship between the message passing algorithm termed Graph Convolutional Network (GCN) by Kipf & Welling (2017) and a random walk. Using this relationship we see that GCN converges to this random walk's limit distribution as the number of layers increases. The limit distribution is a property of the graph as a whole and does not take the random walk's starting (root) node into account. As such it is unsuited to describe the root node's neighborhood. Hence, GCN's performance necessarily deteriorates for a high number of layers (or aggregation/propagation steps).

To solve this issue, in this paper, we first highlight the inherent connection between the limit distribution and PageRank (Page et al., 1998). We then propose an algorithm that utilizes a propagation scheme derived from *personalized* PageRank instead. This algorithm adds a chance of teleporting back to the root node, which ensures that the PageRank score encodes the local neighborhood for every root node (Page et al., 1998). The teleport probability allows us to balance the needs of preserving locality (i.e. staying close to the root node to avoid oversmoothing) and leveraging the information from a large neighborhood. We show that this propagation scheme permits the use of far more (in fact, infinitely many) propagation steps without leading to oversmoothing.

Moreover, while propagation and classification are inherently intertwined in message passing, our proposed algorithm *separates* the neural network from the propagation scheme. This allows us to achieve a much higher range without changing the neural network, whereas in the message passing scheme every additional propagation step would require an additional layer. It also permits the independent development of the propagation algorithm and the neural network generating predictions from node features. That is, we can combine any state-of-the-art prediction method with our propagation scheme. We even found that adding our propagation scheme during inference significantly improves the accuracy of networks that were trained without using any graph information.

Our model achieves state-of-the-art results while requiring fewer parameters and less training time compared to most competing models, with a computational complexity that is linear in the number of edges. We show these results in the most thorough study (including significance testing) of message passing models using graphs with text-based features that has been done so far.

2 Graph convolutional networks and their limited range

We first introduce our notation and explain the problem our model solves. Let $G = (V, E)$ be a graph with nodes V and edges E. Let n denote the number of nodes and m the number of edges. The nodes are described by the feature matrix $\boldsymbol{X} \in \mathbb{R}^{n \times f}$, with the number of features f per node, and the class (or label) matrix $\boldsymbol{Y} \in \mathbb{R}^{n \times c}$, with the number of classes c. The graph G is described by the adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$. $\tilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}_n$ denotes the adjacency matrix with added self-loops.

One simple and widely used message passing algorithm for semi-supervised classification is the Graph Convolutional Network (GCN). In the case of two message passing layers its equation is

$$\boldsymbol{Z}_{\text{GCN}} = \text{softmax}\left(\hat{\tilde{\boldsymbol{A}}}\ \text{ReLU}\left(\hat{\tilde{\boldsymbol{A}}}\boldsymbol{X}\boldsymbol{W}_0\right)\boldsymbol{W}_1\right), \tag{1}$$

where $\boldsymbol{Z} \in \mathbb{R}^{n \times c}$ are the predicted node labels, $\hat{\tilde{\boldsymbol{A}}} = \tilde{\boldsymbol{D}}^{-1/2}\tilde{\boldsymbol{A}}\tilde{\boldsymbol{D}}^{-1/2}$ is the symmetrically normalized adjacency matrix with self-loops, with the diagonal degree matrix $\tilde{\boldsymbol{D}}_{ij} = \sum_k \tilde{\boldsymbol{A}}_{ik}\delta_{ij}$, and $\boldsymbol{W}_0$ and $\boldsymbol{W}_1$ are trainable weight matrices (Kipf & Welling, 2017).

With two GCN-layers, only neighbors in the two-hop neighborhood are considered. There are essentially two reasons why a message passing algorithm like GCN cannot be trivially expanded to use a larger neighborhood. First, aggregation by averaging causes oversmoothing if too many layers are used. It, therefore, loses its focus on the local neighborhood (Li et al., 2018). Second, most common aggregation schemes use learnable weight matrices in each layer. Therefore, using a larger neighborhood necessarily increases the depth and number of learnable parameters of the neural network (the second aspect can be circumvented by using weight sharing, which is typically not the case, though). However, the required neighborhood size and neural network depth are two completely orthogonal aspects. This fixed relationship is a strong limitation and leads to bad compromises.

We will start by concentrating on the first issue. Xu et al. (2018) have shown that for a k-layer GCN the influence score of node x on y, $I(x, y) = \sum_i \sum_j \frac{\partial \boldsymbol{Z}_{yi}}{\partial \boldsymbol{X}_{xj}}$, is proportional in

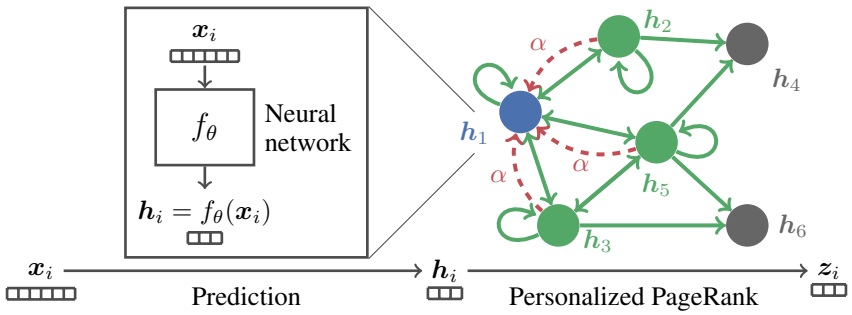


Figure 1: Illustration of (approximate) personalized propagation of neural predictions (PPNP, APPNP). Predictions are first generated from each node's own features by a neural network and then propagated using an adaptation of personalized PageRank. The model is trained end-to-end.

expectation to a slightly modified k-step random walk distribution starting at the root node x, $P_{\text{rw}'}(x \to y, k)$. Hence, the information of node x spreads to node y in a random walk-like manner. If we take the limit $k \to \infty$ and the graph is irreducible and aperiodic, this random walk probability distribution $P_{\text{rw}'}(x \to y, k)$ converges to the limit (or stationary) distribution $P_{\text{lim}}(\to y)$. This distribution can be obtained by solving the equation $\boldsymbol{\pi}_{\text{lim}} = \hat{\tilde{\boldsymbol{A}}}\boldsymbol{\pi}_{\text{lim}}$. Obviously, the result only depends on the graph as a whole and is independent of the random walk's starting (root) node x. This global property is therefore unsuitable for describing the root node's neighborhood.

3 Personalized propagation of neural predictions

From message passing to personalized PageRank. We can solve the problem of lost focus by recognizing the connection between the limit distribution and PageRank (Page et al., 1998). The only differences between these two are the added self-loops and the adjacency matrix normalization in $\hat{\tilde{\boldsymbol{A}}}$. Original PageRank is calculated via $\boldsymbol{\pi}_{\text{pr}} = \boldsymbol{A}_{\text{rw}}\boldsymbol{\pi}_{\text{pr}}$, with $\boldsymbol{A}_{\text{rw}} = \boldsymbol{A}\boldsymbol{D}^{-1}$. Having made this connection we can now consider using a variant of PageRank that takes the root node into account – personalized PageRank (Page et al., 1998). We define the root node x via the teleport vector $\boldsymbol{i}_x$, which is a one-hot indicator vector. Our adaptation of personalized PageRank can be obtained for node x using the recurrent equation $\boldsymbol{\pi}_{\text{ppr}}(\boldsymbol{i}_x) = (1-\alpha)\hat{\tilde{\boldsymbol{A}}}\boldsymbol{\pi}_{\text{ppr}}(\boldsymbol{i}_x) + \alpha\boldsymbol{i}_x$, with the teleport (or restart) probability $\alpha \in (0, 1]$. By solving this equation, we obtain

$$\boldsymbol{\pi}_{\text{ppr}}(\boldsymbol{i}_x) = \alpha\left(\boldsymbol{I}_n - (1-\alpha)\hat{\tilde{\boldsymbol{A}}}\right)^{-1}\boldsymbol{i}_x. \tag{2}$$

Introducing the teleport vector $\boldsymbol{i}_x$ allows us to preserve the node's local neighborhood even in the limit distribution. In this model the influence score of root node x on node y, $I(x, y)$, is proportional to the y-th element of our personalized PageRank $\boldsymbol{\pi}_{\text{ppr}}(\boldsymbol{i}_x)$. This value is different for every root node. How fast it decreases as we move away from the root node can be adjusted via α. By substituting the indicator vector $\boldsymbol{i}_x$ with the unit matrix $\boldsymbol{I}_n$ we obtain our fully personalized PageRank matrix $\boldsymbol{\Pi}_{\text{ppr}} = \alpha(\boldsymbol{I}_n - (1-\alpha)\hat{\tilde{\boldsymbol{A}}})^{-1}$, whose element (yx) specifies the influence score of node x on node y, $I(x, y) \propto \boldsymbol{\Pi}_{\text{ppr}}^{(yx)}$. Note that due to symmetry $\boldsymbol{\Pi}_{\text{ppr}}^{(yx)} = \boldsymbol{\Pi}_{\text{ppr}}^{(xy)}$, i.e. the influence of x on y is equal to the influence of y on x. This inverse always exists since $\frac{1}{1-\alpha} > 1$ and therefore cannot be an eigenvalue of $\hat{\tilde{\boldsymbol{A}}}$ (see Appendix A).

Personalized propagation of neural predictions (PPNP). To utilize the above influence scores for semi-supervised classification we generate predictions for each node based on its own features and then propagate them via our fully personalized PageRank scheme to generate the final predictions. This is the foundation of personalized propagation of neural predictions. PPNP's model equation is

$$\boldsymbol{Z}_{\text{PPNP}} = \text{softmax}\left(\alpha\left(\boldsymbol{I}_n - (1-\alpha)\hat{\tilde{\boldsymbol{A}}}\right)^{-1}\boldsymbol{H}\right), \qquad \boldsymbol{H}_{i,:} = f_\theta(\boldsymbol{X}_{i,:}), \tag{3}$$

where $\boldsymbol{X}$ is the feature matrix and f_θ a neural network with parameter set θ generating the predictions $\boldsymbol{H} \in \mathbb{R}^{n \times c}$. Note that f_θ operates on each node's features independently, allowing for parallelization. Furthermore, one could substitute $\hat{\tilde{\boldsymbol{A}}}$ with any propagation matrix, such as $\boldsymbol{A}_{\text{rw}}$.

As a consequence, PPNP separates the neural network used for generating predictions from the propagation scheme. This separation additionally solves the second issue mentioned above: the depth of the neural network is now fully independent of the propagation algorithm. As we saw when connecting GCN to PageRank, personalized PageRank can effectively use even infinitely many neighborhood aggregation layers, which is clearly not possible in the classical message passing framework. Furthermore, the separation gives us the flexibility to use any method for generating predictions, e.g. deep convolutional neural networks for graphs of images.

While generating predictions and propagating them happen consecutively during inference, it is important to note that **the model is trained end-to-end**. That is, the gradient flows through the propagation scheme during backpropagation (implicitly considering infinitely many neighborhood aggregation layers). Adding these propagation effects significantly improves the model's accuracy.

Efficiency analysis. Directly calculating the fully personalized PageRank matrix $\boldsymbol{\Pi}_{\text{ppr}}$, is computationally inefficient and results in a dense $\mathbb{R}^{n \times n}$ matrix. Using this matrix would lead to a computational complexity and memory requirement of $\mathcal{O}(n^2)$ for training and inference.

To solve this issue, reconsider the equation $\boldsymbol{Z} = \alpha(\boldsymbol{I}_n - (1-\alpha)\hat{\tilde{\boldsymbol{A}}})^{-1}\boldsymbol{H}$. Instead of viewing this equation as a combination of a dense fully *personalized* PageRank matrix with the prediction matrix, we can also view it as a variant of *topic-sensitive* PageRank, with each class corresponding to one topic (Haveliwala, 2002). In this view every *column* of $\boldsymbol{H}$ defines an (unnormalized) distribution over nodes that acts as a teleport set. Hence, we can approximate PPNP via an approximate computation of topic-sensitive PageRank.

Approximate personalized propagation of neural predictions (APPNP). More precisely, APPNP achieves linear computational complexity by approximating topic-sensitive PageRank via power iteration. While PageRank's power iteration is connected to the regular random walk, the power iteration of topic-sensitive PageRank is related to a random walk with restarts. Each power iteration (random walk/propagation) step of our topic-sensitive PageRank variant is, thus, calculated via

$$\begin{aligned} \boldsymbol{Z}^{(0)} &= \boldsymbol{H} = f_\theta(\boldsymbol{X}), \\ \boldsymbol{Z}^{(k+1)} &= (1-\alpha)\hat{\tilde{\boldsymbol{A}}}\boldsymbol{Z}^{(k)} + \alpha\boldsymbol{H}, \\ \boldsymbol{Z}^{(K)} &= \text{softmax}\left((1-\alpha)\hat{\tilde{\boldsymbol{A}}}\boldsymbol{Z}^{(K-1)} + \alpha\boldsymbol{H}\right), \end{aligned} \tag{4}$$

where the prediction matrix $\boldsymbol{H}$ acts as both the starting vector and the teleport set, K defines the number of power iteration steps and $k \in [0, K-2]$. Note that this method retains the graph's sparsity and never constructs an $\mathbb{R}^{n \times n}$ matrix. The convergence of this iterative scheme can be shown by investigating the resulting series (see Appendix B).

Note that the propagation scheme of this model does not require any additional parameters to train – as opposed to models like GCN, which typically require more parameters for each additional propagation layer. We can therefore propagate very far with very few parameters. Our experiments show that this ability is indeed very beneficial (see Section 6). A similar model expressed in the message passing framework would therefore not be able to achieve the same level of performance.

The reformulation of PPNP via fixed-point iterations illustrates a connection to the original graph neural network (GNN) model (Scarselli et al., 2009). While the latter uses a learned fixed-point iteration, our approach uses a predetermined iteration (adapted personalized PageRank) and applies a learned feature transformation *before* propagation.

In both PPNP and APPNP, the size of the neighborhood influencing each node can be adjusted via the teleport probability α. The freedom to choose α allows us to adjust the model for different types of networks, since varying graph types require the consideration of different neighborhood sizes, as shown in Section 6 and described by Grover & Leskovec (2016) and Abu-El-Haija et al. (2018b).

Table 1: Dataset statistics. Shortest path length is denoted by SP.

Dataset	Type	Classes	Features	Nodes	Edges	Label rate	Avg. SP
CITESEER	Citation	6	3703	2110	3668	0.036	9.31
CORA-ML	Citation	7	2879	2810	7981	0.047	5.27
PUBMED	Citation	3	500	19 717	44 324	0.003	6.34
MS ACADEMIC	Co-author	15	6805	18 333	81 894	0.016	5.43

4 RELATED WORK

Several works have tried to improve the training of message passing algorithms and increase the neighborhood available at each node by adding skip connections (Li et al., 2016; Pham et al., 2017; Hamilton et al., 2017; Ying et al., 2018). One recent approach combined skip connection with aggregation schemes (Xu et al., 2018). However, the range of these models is still limited, as apparent in the low number of message passing layers used. While it is possible to add skip connections in the neural network used by our algorithm, this would not influence the propagation scheme. Our approach to solving the range problem is therefore unrelated to these models.

Li et al. (2018) facilitated training by combining message passing with co- and self-training. The improvements achieved by this combination are similar to results reported with other semi-supervised classification models (Buchnik & Cohen, 2018). Note that most algorithms, including ours, can be improved using self- and co-training. However, each additional step used by these methods corresponds to a full training cycle and therefore significantly increases the training time.

Deep GNNs that avoid the oversmoothing issue have been proposed in recent works by combining residual (skip) connections with batch normalization (Kawamoto et al., 2018; Chen et al., 2019). However, our model solves this issue by simplifying the architecture via decoupling prediction and propagation and does not rely on ad-hoc techniques that further complicate the model and introduce additional hyperparameters. Furthermore, since PPNP increases the range without introducing additional layers and parameters it is easier and faster to train compared to a deep GNN.

5 EXPERIMENTAL SETUP

Recently, many experimental evaluations have suffered from superficial statistical evaluation and experimental bias from using varying training setups and overfitting. The latter is caused by experiments using a single training/validation/test split, by not distinguishing clearly between the validation and test set, and by finetuning hyperparameters to each dataset or even data split separately. Message-passing algorithms are very sensitive to both data splits and weight initialization (as clearly shown by our evaluation). Thus, a carefully designed evaluation protocol is extremely important. Our work aims to establish such a thorough evaluation protocol. First, we run each experiment 100 times on multiple random splits and initializations. Second, we split the data into a visible and a test set, which do not change. The test set was only used *once* to report the final performance; and in particular, has never been used to perform hyperparameter and model selection. To further prevent overfitting we use the *same* number of layers and hidden units, dropout rate d, L_2 regularization parameter λ, and learning rate l across datasets, since all datasets use bag-of-words as features. To prevent experimental bias we optimized the hyperparameters of *all* models individually using a grid search on CITESEER and CORA-ML and use the *same* early stopping criterion across models.

Finally, to ensure the statistical robustness of our experimental setup, we calculate confidence intervals via bootstrapping and report the p-values of a paired t-test for our main claims. To our knowledge, this is the most rigorous study on GCN-like models which has been done so far. More details about the experimental setup are provided in Appendix C.

Datasets. We use four text-classification datasets for evaluation. CITESEER (Sen et al., 2008), CORA-ML (McCallum et al., 2000; Bojchevski & Günnemann, 2018) and PUBMED (Namata et al., 2012) are citation graphs, where each node represents a paper and the edges represent citations between them. In the MICROSOFT ACADEMIC graph (Shchur et al., 2018) edges represent co-authorship. We use the largest connected component of each graph. All graphs use a bag-of-words representation of the papers' abstracts as features. While large graphs do not necessarily have a

Table 2: Average accuracy with uncertainties showing the 95 % confidence level calculated by bootstrapping. Previously reported improvements vanish on our rigorous experimental setup, while PPNP and APPNP significantly outperform the compared models on all datasets.

Model	CITESEER	CORA-ML	PUBMED	MS ACADEMIC
V. GCN	73.51 ± 0.48	82.30 ± 0.34	77.65 ± 0.40	91.65 ± 0.09
GCN	75.40 ± 0.30	83.41 ± 0.39	78.68 ± 0.38	92.10 ± 0.08
N-GCN	74.25 ± 0.40	82.25 ± 0.30	77.43 ± 0.42	92.86 ± 0.11
GAT	75.39 ± 0.27	84.37 ± 0.24	77.76 ± 0.44	91.22 ± 0.07
JK	73.03 ± 0.47	82.69 ± 0.35	77.88 ± 0.38	91.71 ± 0.10
Bt. FP	73.55 ± 0.57	80.84 ± 0.97	72.94 ± 1.00	91.61 ± 0.24
PPNP*	$\mathbf{75.83 \pm 0.27}$	$\mathbf{85.29 \pm 0.25}$	-	-
APPNP	75.73 ± 0.30	85.09 ± 0.25	$\mathbf{79.73 \pm 0.31}$	$\mathbf{93.27 \pm 0.08}$

*out of memory on PUBMED, MS ACADEMIC (see efficiency analysis in Section 3)

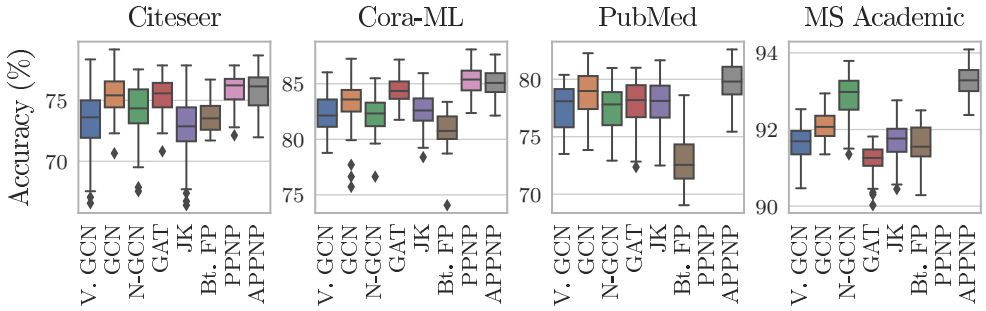


Figure 2: Accuracy distributions of different models. The high standard deviation between data splits and initializations shows the importance of a rigorous evaluation, which is often omitted.

larger diameter (Leskovec et al., 2005), note that these graphs indeed have average shortest path lengths between 5 and 10 and therefore a regular two-layer GCN cannot cover the entire graph. Table 1 reports the dataset statistics.

Baseline models. We compare to five state-of-the-art models: GCN (Kipf & Welling, 2017), network of GCNs (N-GCN) (Abu-El-Haija et al., 2018a), graph attention networks (GAT) (Veličković et al., 2018), bootstrapped feature propagation (bt. FP) (Buchnik & Cohen, 2018) and jumping knowledge networks with concatenation (JK) (Xu et al., 2018). For GCN we also show the results of the (unoptimized) vanilla version (V. GCN) to demonstrate the strong impact of early stopping and hyperparameter optimization. The hyperparameters of all models are listed in Appendix D.

Model hyperparameters. To ensure a fair model comparison we used a neural network for PPNP that is structurally very similar to GCN and has the same number of parameters. We use two layers with $h = 64$ hidden units. We apply L_2 regularization with $\lambda = 0.005$ on the weights of the first layer and use dropout with dropout rate $d = 0.5$ on both layers and the adjacency matrix. For APPNP, adjacency dropout is resampled for each power iteration step. For propagation we use the teleport probability $\alpha = 0.1$ and $K = 10$ power iteration steps for APPNP. We use $\alpha = 0.2$ on the MICROSOFT ACADEMIC graph due to its structural difference (see Figure 5 and its discussion). The combination of this shallow neural network with a comparatively high number of power iteration steps achieved the best results during hyperparameter optimization (see Appendix G).

6 RESULTS

Overall accuracy. The results for the accuracy (micro F1-score) are summarized in Table 2. Similar trends are observed for the macro F1-score (see Appendix E). Both models significantly outperform the state-of-the-art baseline models on all datasets. Our rigorous setup might understate the improvements achieved by PPNP and APPNP – this result is statistically significant $p < 0.05$, as tested via a paired t-test (see Appendix F). This thorough setup furthermore shows that the advantages reported by recent works practically vanish when training is harmonized, hyperparameters are properly op-

Table 3: Average training time per epoch. PPNP and APPNP are only slightly slower than GCN and much faster than more sophisticated methods like GAT.

Graph	V. GCN	GCN	N-GCN	GAT	JK	Bt. FP*	PPNP**	APPNP
CITESEER	37.6 ms	35.3 ms	115.9 ms	187.0 ms	57.5 ms	-	49.2 ms	43.3 ms
CORA-ML	32.4 ms	36.5 ms	118.9 ms	217.4 ms	43.6 ms	-	55.3 ms	42.7 ms
PUBMED	48.6 ms	48.3 ms	342.6 ms	1029.8 ms	77.8 ms	-	-	64.1 ms
MS ACADEMIC	45.5 ms	39.2 ms	328.5 ms	772.2 ms	61.9 ms	-	-	59.8 ms

*not applicable, since core method not trainable **out of memory on PUBMED, MS ACADEMIC (see efficiency analysis in Section 3)

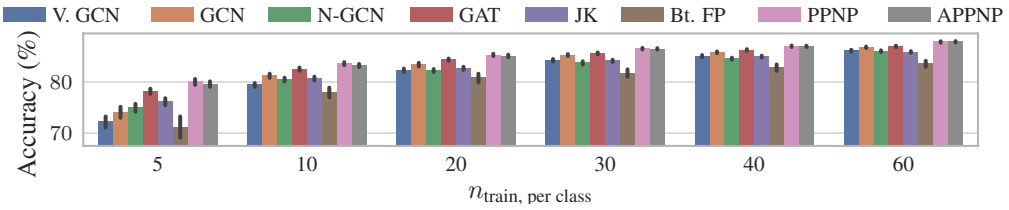


Figure 3: Accuracy for different training set sizes (number of labeled nodes per class) on CORA-ML. PPNP's dominance increases further for smaller training set sizes.

timized and multiple data splits are considered. A simple GCN with optimized hyperparameters outperforms several recently proposed models on our setup.

Figure 2 shows how broad the accuracy distribution of each model is. This is caused by both random initialization and different data splits (train / early stopping / test). This demonstrates how crucial a statistically rigorous evaluation is for a conclusive model comparison. Moreover, it shows the sensitivity (robustness) of each method, e.g. PPNP, APPNP and GAT typically have lower variance.

Training time per epoch. We report the average training time per epoch in Table 3. We decided to only compare the training time per epoch since all hyperparameters were solely optimized for accuracy and the used early stopping criterion is very generous. Obviously, (exact) PPNP can only be applied to moderately sized graphs, while APPNP scales to large data. On average, APPNP is around 25 % slower than GCN due to its higher number of matrix multiplications. It scales similarly with graph size as GCN and is therefore significantly faster than other more sophisticated models like GAT. This is observed even though our implementation improved GAT's training time roughly by a factor of 2 compared to the reference implementation.

Training set size. Since the labeling rate is often very small for real world datasets, investigating how the models perform with a small number of training samples is very important. Figure 3 shows how the number of training nodes per class $n_{\text{train, per class}}$ impacts the accuracy on CORA-ML (for other datasets see Appendix H). The dominance of PPNP and APPNP increases further in this sparsely labelled setting. This can be attributed to their higher range, which allows them to better propagate the information further away from the (few) training nodes. We see further evidence for this when comparing the accuracy of APPNP and GCN depending on the distance between a node and the training set (in terms of shortest path). Appendix I shows that the performance gap between APPNP and GCN tends to increase for nodes that are far away from the training nodes. That is, nodes further away from the training set benefit more from the increase in range.

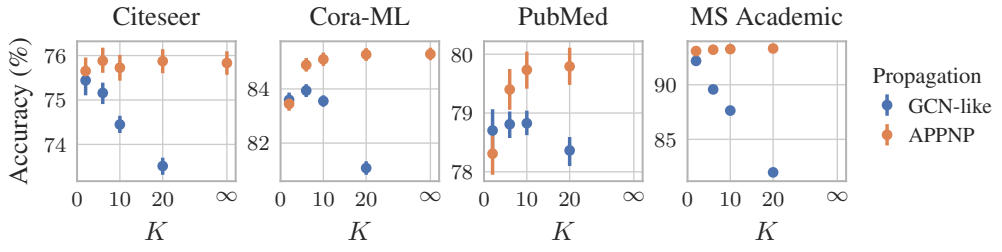


Figure 4: Accuracy depending on the number of propagation steps K. The accuracy breaks down for the GCN-like propagation ($\alpha = 0$), while it increases and stabilizes when using APPNP ($\alpha = 0.1$).

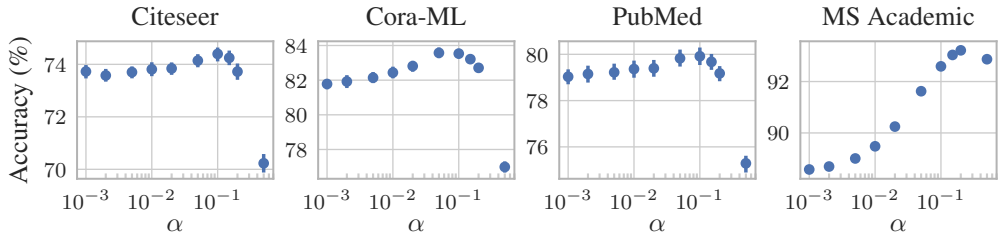


Figure 5: Accuracy depending on teleport probability α. The optimum typically lies within $\alpha \in [0.05, 0.2]$, but changes for different types of datasets.

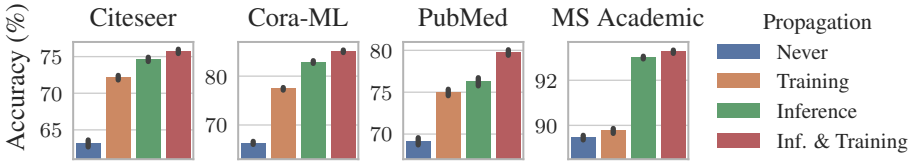


Figure 6: Accuracy of APPNP with propagation used only during training/inference. Best results are achieved with full propagation, but propagating only during inference also achieves good results.

Number of power iteration steps. Figure 4 shows how the accuracy depends on the number of power iterations for two different propagation schemes. The first mimics the standard propagation as known from GCNs (i.e. $\alpha = 0$ in APPNP). As clearly shown the performance breaks down as we increase the number of power iterations K (since we approach the global PageRank solution). However, when using personalized propagation (with $\alpha = 0.1$) the accuracy *increases* and converges to exact PPNP with infinitely many propagation steps, thus demonstrating the personalized propagation principle is indeed beneficial. As also shown in the figure, it is enough to use a moderate number of power iterations (e.g. $K = 10$) to effectively approximate exact PPNP. Interestingly, we've found that this number coincides with the highest shortest path distance of any node to the training set.

Teleport probability α. Figure 5 shows the effect of the hyperparameter α on the accuracy on the validation set. While the optimum differs slightly for every dataset, we consistently found a teleport probability of around $\alpha \in [0.05, 0.2]$ to perform best. This probability should be adjusted for the dataset under investigation, since different graphs exhibit different neighborhood structures (Grover & Leskovec, 2016; Abu-El-Haija et al., 2018b). Note that a higher α improves convergence speed.

Neural network without propagation. PPNP and APPNP are trained end-to-end, with the propagation scheme affecting (i) the neural network f_θ during training, and (ii) the classification decision during inference. Investigating how the model performs without propagation shows if and how valuable this addition is. Figure 6 shows how propagation affects both training and inference. "Never" denotes the case where no propagation is used; essentially we train and apply a standard multilayer perceptron (MLP) f_θ using the features only. "Training" denotes the case where we use APPNP during training to learn f_θ; at inference time, however, only f_θ is used to predict the class labels. "Inference", in contrast, denotes the case where f_θ is trained without APPNP (i.e. standard MLP on features). This pretrained network with fixed weights is then used with APPNP's propagation for inference. Finally, "Inf. & Training" denotes the regular APPNP, which always uses propagation.

The best results are achieved with regular APPNP, which validates our approach. However, on most datasets the accuracy decreases surprisingly little when propagating only during inference. Skipping propagation during training can significantly reduce training time for large graphs as all nodes can be handled independently. This also shows that our model can be combined with pretrained neural networks that do not incorporate any graph information and still significantly improve their accuracy. Moreover, Figure 6 shows that just propagating during training can also lead to large improvements. This indicates that our model can also be applied to online/inductive learning where only the features and not the neighborhood information of an incoming (previously unobserved) node are available.

7 CONCLUSION

In this paper we have introduced personalized propagation of neural predictions (PPNP) and its fast approximation, APPNP. We derived this model by considering the relationship between GCN and PageRank and extending it to personalized PageRank. This simple model decouples prediction and propagation and solves the limited range problem inherent in many message passing models without introducing any additional parameters. It uses the information from a large, adjustable (via the teleport probability α) neighborhood for classifying each node. The model is computationally efficient and outperforms several state-of-the-art methods for semi-supervised classification on multiple graphs in the most thorough study which has been done for GCN-like models so far.

For future work it would be interesting to combine PPNP with more complex neural networks used e.g. in computer vision or natural language processing. Furthermore, faster or incremental approximations of personalized PageRank (Bahmani et al., 2010; 2011; Lofgren et al., 2014) and more sophisticated propagation schemes would also benefit the method.

ACKNOWLEDGEMENTS

This research was supported by the German Research Foundation, grant GU 1409/2-1.

REFERENCES

Sami Abu-El-Haija, Amol Kapoor, Bryan Perozzi, and Joonseok Lee. N-GCN: Multi-scale Graph Convolution for Semi-supervised Node Classification. In *International Workshop on Mining and Learning with Graphs (MLG)*, 2018a.

Sami Abu-El-Haija, Bryan Perozzi, Rami Al-Rfou, and Alex Alemi. Watch Your Step: Learning Node Embeddings via Graph Attention. In *NeurIPS*, 2018b.

Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. Fast Incremental and Personalized PageRank. *VLDB*, 2010.

Bahman Bahmani, Kaushik Chakrabarti, and Dong Xin. Fast Personalized PageRank on MapReduce. In *SIGMOD*, 2011.

Aleksandar Bojchevski and Stephan Günnemann. Deep Gaussian Embedding of Graphs: Unsupervised Inductive Learning via Ranking. *ICLR*, 2018.

Aleksandar Bojchevski, Oleksandr Shchur, Daniel Zügner, and Stephan Günnemann. NetGAN: Generating Graphs via Random Walks. In *ICML*, 2018.

Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral Networks and Deep Locally Connected Networks on Graphs. *ICLR*, 2014.

Eliav Buchnik and Edith Cohen. Bootstrapped Graph Diffusions: Exposing the Power of Nonlinearity. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, 2 (1):1–19, April 2018.

Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. *ICLR*, 2018.

Zhengdao Chen, Lisha Li, and Joan Bruna. Supervised Community Detection with Line Graph Neural Networks. In *ICLR*, 2019.

Hanjun Dai, Zornitsa Kozareva, Bo Dai, Alexander J. Smola, and Le Song. Learning Steady-States of Iterative Algorithms over Graphs. In *ICML*, 2018.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *NIPS*, 2016.

David K. Duvenaud, Dougal Maclaurin, Jorge Aguilera-Iparraguirre, Rafael Gómez-Bombarelli, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P. Adams. Convolutional Networks on Graphs for Learning Molecular Fingerprints. In *NIPS*, 2015.

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

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010.

Aditya Grover and Jure Leskovec. node2vec: Scalable Feature Learning for Networks. In *KDD*, 2016.

William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *NIPS*, 2017.

Taher H. Haveliwala. Topic-sensitive PageRank. In *WWW*, 2002.

Tatsuro Kawamoto, Masashi Tsubaki, and Tomoyuki Obuchi. Mean-field theory of graph neural networks in graph partitioning. In *NeurIPS*, 2018.

Steven M. Kearnes, Kevin McCloskey, Marc Berndl, Vijay S. Pande, and Patrick Riley. Molecular graph convolutions: moving beyond fingerprints. *Journal of Computer-Aided Molecular Design*, 30(8):595–608, 2016.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *ICLR*, 2015.

Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. *ICLR*, 2017.

Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *KDD*, 2005.

Qimai Li, Zhichao Han, and Xiao-Ming Wu. Deeper Insights Into Graph Convolutional Networks for Semi-Supervised Learning. In *AAAI*, 2018.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated Graph Sequence Neural Networks. In *ICLR*, 2016.

Peter Lofgren, Siddhartha Banerjee, Ashish Goel, and Seshadhri Comandur. FAST-PPR: scaling personalized pagerank estimation for large graphs. In *KDD*, 2014.

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, Dandelion 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. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015.

Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval*, 3(2):127–163, 2000.

Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodola, Jan Svoboda, and Michael M. Bronstein. Geometric Deep Learning on Graphs and Manifolds Using Mixture Model CNNs. In *CVPR*, 2017.

Galileo Namata, Ben London, Lise Getoor, and Bert Huang. Query-driven Active Surveying for Collective Classification. In *International Workshop on Mining and Learning with Graphs (MLG)*, 2012.

Sharad Nandanwar and M. N. Murty. Structural Neighborhood Based Classification of Nodes in a Network. In *KDD*, 2016.

Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning Convolutional Neural Networks for Graphs. In *ICML*, 2016.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. *Technical report, Stanford InfoLab*, 1998.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk: online learning of social representations. In *KDD*, 2014.

Trang Pham, Truyen Tran, Dinh Q. Phung, and Svetha Venkatesh. Column Networks for Collective Classification. In *AAAI*, 2017.

Jiezhong Qiu, Yuxiao Dong, Hao Ma, Jian Li, Kuansan Wang, and Jie Tang. Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec. In *ACM International Conference on Web Search and Data Mining (WSDM)*, 2018.

F. Scarselli, M. Gori, Ah Chung Tsoi, M. Hagenbuchner, and G. Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80, January 2009.

Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling Relational Data with Graph Convolutional Networks. In *Extended Semantic Web Conference (ESWC)*, 2018.

Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective Classification in Network Data. *AI Magazine*, 29(3):93–106, 2008.

Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of Graph Neural Network Evaluation. In *Relational Representation Learning Workshop (R2L 2018), NeurIPS*, 2018.

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

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *ICLR*, 2018.

Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation Learning on Graphs with Jumping Knowledge Networks. In *ICML*, 2018.

Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. Revisiting Semi-Supervised Learning with Graph Embeddings. In *ICML*, 2016.

Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. *KDD*, 2018.

A EXISTENCE OF $\boldsymbol{\Pi}_{\text{PPR}}$

The matrix

$$\boldsymbol{\Pi}_{\text{ppr}} = \alpha \left(\boldsymbol{I}_n - (1-\alpha)\hat{\tilde{\boldsymbol{A}}} \right)^{-1} \tag{5}$$

exists iff the determinant $\det(\boldsymbol{I}_n - (1-\alpha)\hat{\tilde{\boldsymbol{A}}}) \neq 0$, which is the case iff $\det(\hat{\tilde{\boldsymbol{A}}} - \frac{1}{1-\alpha}\boldsymbol{I}_n) \neq 0$, i.e. iff $\frac{1}{1-\alpha}$ is not an eigenvalue of $\hat{\tilde{\boldsymbol{A}}}$. This value is always larger than 1 since the teleport probability $\alpha \in (0, 1]$. Furthermore, the symmetrically normalized matrix $\hat{\tilde{\boldsymbol{A}}}$ has the same eigenvalues as the row-stochastic matrix $\tilde{\boldsymbol{A}}_{\text{rw}}$. This can be shown by multiplying the eigenvalue equation $\hat{\tilde{\boldsymbol{A}}}\boldsymbol{v} = \lambda\boldsymbol{v}$ with $\tilde{\boldsymbol{D}}^{-1/2}$ from left and substituting $\boldsymbol{w} = \tilde{\boldsymbol{D}}^{-1/2}\boldsymbol{v}$. This also shows that the eigenvectors of $\hat{\tilde{\boldsymbol{A}}}$ are the eigenvectors of $\tilde{\boldsymbol{A}}_{\text{rw}}$ scaled by $\tilde{\boldsymbol{D}}^{1/2}$. The largest eigenvalue of a row-stochastic matrix is 1, as can be proven using the Gershgorin circle theorem. Hence, $\frac{1}{1-\alpha}$ cannot be an eigenvalue and $\boldsymbol{\Pi}_{\text{ppr}}$ always exists.

B CONVERGENCE OF APPNP

APPNP uses the iterative equation

$$\boldsymbol{Z}^{(k+1)} = (1-\alpha)\hat{\tilde{\boldsymbol{A}}}\boldsymbol{Z}^{(k)} + \alpha\boldsymbol{H}. \tag{6}$$

After the k-th propagation step, the resulting predictions are

$$\boldsymbol{Z}^{(k)} = \left((1-\alpha)^k \hat{\tilde{\boldsymbol{A}}}^k + \alpha\sum_{i=0}^{k-1}(1-\alpha)^i \hat{\tilde{\boldsymbol{A}}}^i\right)\boldsymbol{H}. \tag{7}$$

If we take the limit $k \to \infty$ the left term tends to 0 and the right term becomes a geometric series. The series converges since $\alpha \in (0, 1]$ and $\hat{\tilde{\boldsymbol{A}}}$ is symmetrically normalized and therefore $\det(\hat{\tilde{\boldsymbol{A}}}) \leq 1$, resulting in

$$\boldsymbol{Z}^{(\infty)} = \alpha\left(\boldsymbol{I}_n - (1-\alpha)\hat{\tilde{\boldsymbol{A}}}\right)^{-1}\boldsymbol{H}, \tag{8}$$

which is the equation for calculating (exact) PPNP.

C EXPERIMENTAL DETAILS

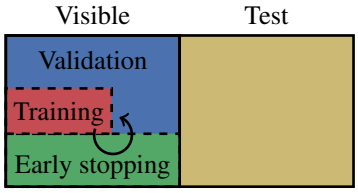


Figure 7: Illustration of the node sampling procedure.

The sampling procedure is illustrated in Figure 7. The data is first split into a visible and a test set. For the visible set 1500 nodes were sampled for the citation graphs and 5000 for MICROSOFT ACADEMIC. The test set contains all remaining nodes. We use three different label sets in each experiment: A training set of 20 nodes per class, an early stopping set of 500 nodes and either a validation or test set. The validation set contains the remaining nodes of the visible set. We use 20 random seeds for determining the splits. These seeds are drawn once and fixed across runs to facilitate comparisons. We use one set of seeds for the validation splits and a different set for the test splits. Each experiment is run with 5 random initializations on each data split, leading to a total of 100 runs per experiment.

The early stopping criterion uses a patience of $p = 100$ and an (unreachably high) maximum of $n = 10\,000$ epochs. The patience is reset whenever the accuracy increases or the loss decreases on the early stopping set. We choose the parameter set achieving the highest accuracy and break ties by selecting the lowest loss on this set. This criterion was inspired by GAT (Veličković et al., 2018).

We used TensorFlow (Martín Abadi et al., 2015) for all experiments except bootstrapped feature propagation. All uncertainties and confidence intervals correspond to a confidence level of $95\,\%$ and were calculated by bootstrapping with 1000 samples.

We use the Adam optimizer with a learning rate of $l = 0.01$ and cross-entropy loss for all models (Kingma & Ba, 2015). Weights are initialized as described in Glorot & Bengio (2010). The feature matrix is L_1 normalized per row.

D BASELINE HYPERPARAMETERS

Vanilla GCN uses the original settings of two layers with $h = 16$ hidden units, no dropout on the adjacency matrix, L_2 regularization parameter $\lambda = 5 \times 10^{-4}$ and the original early stopping with a maximum of 200 steps and a patience of 10 steps based on the loss.

The optimized GCN uses two layers with $h = 64$ hidden units, dropout on the adjacency matrix with $d = 0.5$ and L_2 regularization parameter $\lambda = 0.02$.

N-GCN uses $h = 16$ hidden units, $R = 4$ heads per random walk length and random walks of up to $K - 1 = 4$ steps. It uses L_2 regularization on all layers with $\lambda = 1 \times 10^{-5}$ and the attention variant for merging the predictions (Abu-El-Haija et al., 2018a). Note that this model effectively uses $RKh = 320$ hidden units, which is 5 times as many units compared to GCN, GAT, and PPNP.

For GAT we use the (well optimized) original hyperparameters, except the L_2 regularization parameter $\lambda = 0.001$ and learning rate $l = 0.01$. As opposed to the original paper, we do not use different hyperparameters on PUBMED, as described in our experimental setup.

Bootstrapped feature propagation uses a return probability of $\alpha = 0.2$, 10 propagation steps, 10 bootstrapping (self-training) steps with $r = 0.1n$ training nodes added per step. We add the training nodes with the lowest entropy on the predictions. The number of nodes added per class is based on the class proportions estimated using the predictions. Note that this model does not include any stochasticity in its initialization. We therefore only run it once per train/early stopping/test split.

For the jumping knowledge networks we use the concatenation variant with three layers and $h = 64$ hidden units per layer. We apply L_2 regularization with $\lambda = 0.001$ on all layers and perform dropout with $d = 0.5$ on all layers but not on the adjacency matrix.

E F1 SCORE

Table 4: Average macro F1 score with uncertainties showing the 95 % confidence level calculated by bootstrapping. PPNP achieves the highest F1 score on all datasets investigated.

Model	CITESEER	CORA-ML	PUBMED	MS ACADEMIC
V. GCN	0.7002 ± 0.0043	0.8205 ± 0.0027	0.7801 ± 0.0038	0.9000 ± 0.0008
GCN	0.7065 ± 0.0037	0.8289 ± 0.0030	0.7883 ± 0.0032	0.9045 ± 0.0008
N-GCN	0.7021 ± 0.0035	0.8183 ± 0.0024	0.7773 ± 0.0040	0.9144 ± 0.0012
GAT	0.7062 ± 0.0029	0.8359 ± 0.0025	0.7777 ± 0.0040	0.8917 ± 0.0007
JK	0.6914 ± 0.0043	0.8202 ± 0.0026	0.7799 ± 0.0039	0.8985 ± 0.0012
Bt. FP	0.6789 ± 0.0055	0.8026 ± 0.0082	0.7448 ± 0.0079	0.8997 ± 0.0018
PPNP*	0.7102 ± 0.0041	$\mathbf{0.8454 \pm 0.0021}$	-	-
APPNP	$\mathbf{0.7105 \pm 0.0038}$	0.8429 ± 0.0022	$\mathbf{0.7966 \pm 0.0031}$	$\mathbf{0.9184 \pm 0.0009}$

* out of memory on PUBMED, MS ACADEMIC (see efficiency analysis in Section 3)

F PAIRED t-TEST

Table 5: p-value of the paired t-test with respect to accuracy.

Model	CITESEER	CORA-ML	PUBMED	MS ACADEMIC
PPNP	1.02×10^{-3}	5.96×10^{-14}	-	-
APPNP	1.77×10^{-2}	4.27×10^{-9}	2.19×10^{-15}	5.93×10^{-13}

Table 6: p-value of the paired t-test with respect to F1 score.

Model	CITESEER	CORA-ML	PUBMED	MS ACADEMIC
PPNP	4.49×10^{-2}	4.50×10^{-14}	-	-
APPNP	2.32×10^{-2}	1.07×10^{-8}	8.70×10^{-14}	1.99×10^{-8}

G NUMBER OF NEURAL NETWORK LAYERS

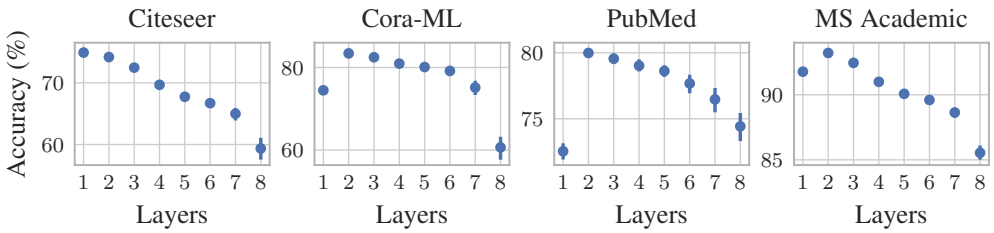


Figure 8: Validation accuracy of APPNP for varying numbers of neural network (NN) layers. Deep NNs do not improve the accuracy, which is probably due to the simple bag-of-words features and the small training set size.

H TRAINING SET SIZE

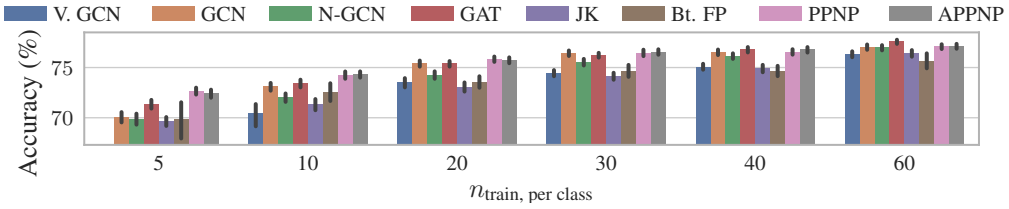


Figure 9: Accuracy for different training set sizes on CITESEER.

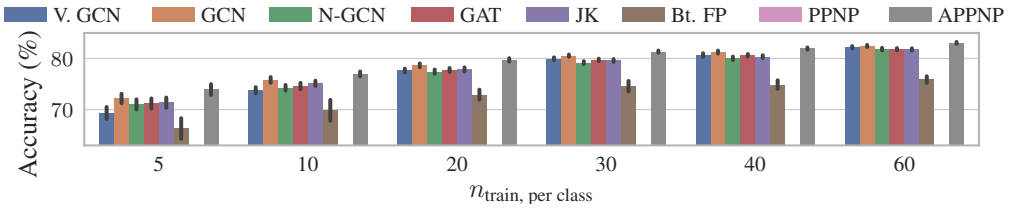


Figure 10: Accuracy for different training set sizes on PUBMED.

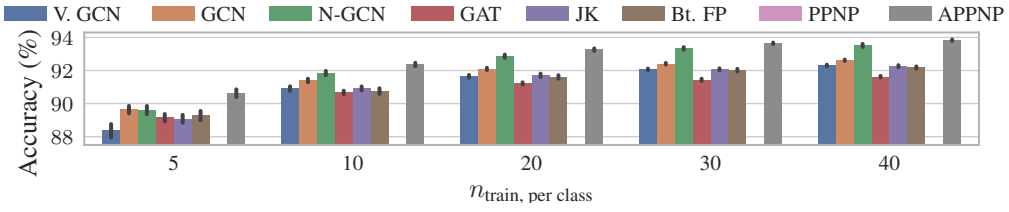


Figure 11: Accuracy for different training set sizes on MICROSOFT ACADEMIC.

I ACCURACY DEPENDING ON DISTANCE FROM TRAINING NODES

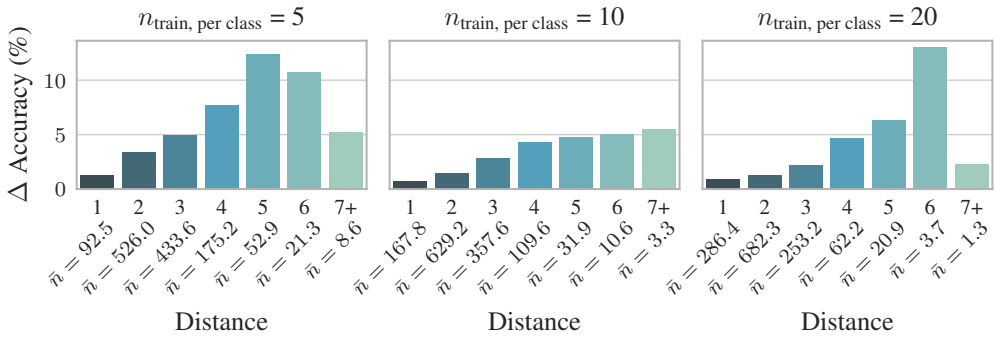


Figure 12: Δ Accuracy (%) denotes the average improvement in percentage points of APPNP over GCN depending on the distance (number of hops) from the training nodes on CORA-ML. $\bar{n}$ denotes the average number of nodes at each distance. The improvement increases with distance.

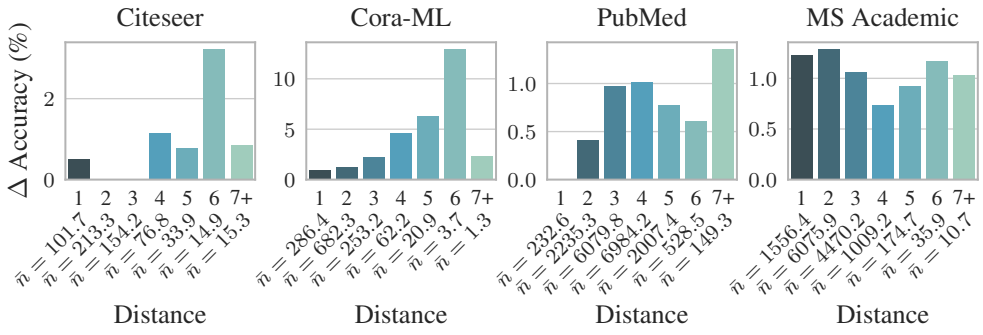


Figure 13: Δ Accuracy (%) denotes the average improvement in percentage points of APPNP over GCN depending on the distance (number of hops) from the training nodes on different graphs. $\bar{n}$ denotes the average number of nodes at each distance over different splits.

Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks

Christopher Morris,[1] **Martin Ritzert,**[2] **Matthias Fey,**[1] **William L. Hamilton,**[3]
Jan Eric Lenssen,[1] **Gaurav Rattan,**[2] **Martin Grohe**[2]
[1]TU Dortmund University
[2]RWTH Aachen University
[3]McGill University and MILA
{christopher.morris, matthias.fey, janeric.lenssen}@tu-dortmund.de,
{ritzert, rattan, grohe}@informatik.rwth-aachen.de,
wlh@cs.mcgill.ca

Abstract

In recent years, graph neural networks (GNNs) have emerged as a powerful neural architecture to learn vector representations of nodes and graphs in a supervised, end-to-end fashion. Up to now, GNNs have only been evaluated empirically—showing promising results. The following work investigates GNNs from a theoretical point of view and relates them to the 1-dimensional Weisfeiler-Leman graph isomorphism heuristic (1-WL). We show that GNNs have the same expressiveness as the 1-WL in terms of distinguishing non-isomorphic (sub-)graphs. Hence, both algorithms also have the same shortcomings. Based on this, we propose a generalization of GNNs, so-called k-dimensional GNNs (k-GNNs), which can take higher-order graph structures at multiple scales into account. These higher-order structures play an essential role in the characterization of social networks and molecule graphs. Our experimental evaluation confirms our theoretical findings as well as confirms that higher-order information is useful in the task of graph classification and regression.

Introduction

Graph-structured data is ubiquitous across application domains ranging from chemo- and bioinformatics to image and social network analysis. To develop successful machine learning models in these domains, we need techniques that can exploit the rich information inherent in graph structure, as well as the feature information contained within a graph's nodes and edges. In recent years, numerous approaches have been proposed for machine learning graphs—most notably, approaches based on graph kernels (Vishwanathan et al. 2010) or, alternatively, using graph neural network algorithms (Hamilton, Ying, and Leskovec 2017b).

Kernel approaches typically fix a set of features in advance—e.g., indicator features over subgraph structures or features of local node neighborhoods. For example, one of the most successful kernel approaches, the *Weisfeiler-Lehman subtree kernel* (Shervashidze et al. 2011), which is based on the 1-dimensional Weisfeiler-Leman graph isomorphism heuristic (Grohe 2017, pp. 79 ff.), generates node features through an iterative relabeling, or *coloring*, scheme: First, all nodes are assigned a common initial color; the algorithm then iteratively recolors a node by aggregating over the multiset of colors in its neighborhood, and the final feature representation of a graph is the histogram of the resulting node colors. By iteratively aggregating over local node neighborhoods in this way, the WL subtree kernel is able to effectively summarize the neighborhood substructures present in a graph. However, while powerful, the WL subtree kernel—like other kernel methods—is limited because this feature construction scheme is fixed (i.e., it does not adapt to the given data distribution). Moreover, this approach—like the majority of kernel methods—focuses only on the graph structure and cannot interpret continuous node and edge labels, such as real-valued vectors which play an important role in applications such as bio- and chemoinformatics.

Graph neural networks (GNNs) have emerged as a machine learning framework addressing the above challenges. Standard GNNs can be viewed as a neural version of the 1-WL algorithm, where colors are replaced by continuous feature vectors and neural networks are used to aggregate over node neighborhoods (Hamilton, Ying, and Leskovec 2017a; Kipf and Welling 2017). In effect, the GNN framework can be viewed as implementing a continuous form of graph-based "message passing", where local neighborhood information is aggregated and passed on to the neighbors (Gilmer et al. 2017). By deploying a trainable neural network to aggregate information in local node neighborhoods, GNNs can be trained in an end-to-end fashion together with the parameters of the classification or regression algorithm, possibly allowing for greater adaptability and better generalization compared to the kernel counterpart of the classical 1-WL algorithm.

Up to now, the evaluation and analysis of GNNs has been largely empirical, showing promising results compared to kernel approaches, see, e.g., (Ying et al. 2018b). However, it remains unclear how GNNs are actually encoding graph structure information into their vector representations, and

whether there are theoretical advantages of GNNs compared to kernel based approaches.

Present Work. We offer a theoretical exploration of the relationship between GNNs and kernels that are based on the 1-WL algorithm. We show that GNNs cannot be more powerful than the 1-WL in terms of distinguishing non-isomorphic (sub-)graphs, e.g., the properties of subgraphs around each node. This result holds for a broad class of GNN architectures and all possible choices of parameters for them. On the positive side, we show that given the right parameter initialization GNNs have the same expressiveness as the 1-WL algorithm, completing the equivalence. Since the power of the 1-WL has been completely characterized, see, e.g., (Arvind et al. 2015; Kiefer, Schweitzer, and Selman 2015), we can transfer these results to the case of GNNs, showing that both approaches have the same shortcomings.

Going further, we leverage these theoretical relationships to propose a generalization of GNNs, called k-GNNs, which are neural architectures based on the k-dimensional WL algorithm (k-WL), which are strictly more powerful than GNNs. The key insight in these higher-dimensional variants is that they perform message passing directly between subgraph structures, rather than individual nodes. This higher-order form of message passing can capture structural information that is not visible at the node-level.

Graph kernels based on the k-WL have been proposed in the past (Morris, Kersting, and Mutzel 2017). However, a key advantage of implementing higher-order message passing in GNNs—which we demonstrate here—is that we can design hierarchical variants of k-GNNs, which combine graph representations learned at different granularities in an end-to-end trainable framework. Concretely, in the presented hierarchical approach the initial messages in a k-GNN are based on the output of lower-dimensional k'-GNN (with $k' < k$), which allows the model to effectively capture graph structures of varying granularity. Many real-world graphs inherit a hierarchical structure—e.g., in a social network we must model both the ego-networks around individual nodes, as well as the coarse-grained relationships between entire communities, see, e.g., (Newman 2003)—and our experimental results demonstrate that these hierarchical k-GNNs are able to consistently outperform traditional GNNs on a variety of graph classification and regression tasks. Across twelve graph regression tasks from the QM9 benchmark, we find that our hierarchical model reduces the mean absolute error by 54.45% on average. For graph classification, we find that our hierarchical models leads to slight performance gains.

Key Contributions. Our key contributions are summarized as follows:

1. We show that GNNs are not more powerful than the 1-WL in terms of distinguishing non-isomorphic (sub-)graphs. Moreover, we show that, assuming a suitable parameter initialization, GNNs have the same power as the 1-WL.
2. We propose k-GNNs, which are strictly more powerful than GNNs. Moreover, we propose a hierarchical version of k-GNNs, so-called 1-k-GNNs, which are able to work with the fine- and coarse-grained structures of a given graph, and relationships between those.
3. Our theoretical findings are backed-up by an experimental study, showing that higher-order graph properties are important for successful graph classification and regression.

Related Work

Our study builds upon a wealth of work at the intersection of supervised learning on graphs, kernel methods, and graph neural networks.

Historically, kernel methods—which implicitly or explicitly map graphs to elements of a Hilbert space—have been the dominant approach for supervised learning on graphs. Important early work in this area includes random-walk based kernels (Gärtner, Flach, and Wrobel 2003; Kashima, Tsuda, and Inokuchi 2003)) and kernels based on shortest paths (Borgwardt and Kriegel 2005). More recently, developments in graph kernels have emphasized scalability, focusing on techniques that bypass expensive Gram matrix computations by using explicit feature maps. Prominent examples of this trend include kernels based on graphlet counting (Shervashidze et al. 2009), and, most notably, the Weisfeiler-Lehman subtree kernel (Shervashidze et al. 2011) as well as its higher-order variants (Morris, Kersting, and Mutzel 2017). Graphlet and Weisfeiler-Leman kernels have been successfully employed within frameworks for smoothed and deep graph kernels (Yanardag and Vishwanathan 2015a; 2015b). Recent works focus on assignment-based approaches (Kriege, Giscard, and Wilson 2016; Nikolentzos, Meladianos, and Vazirgiannis 2017; Johansson and Dubhashi 2015), spectral approaches (Kondor and Pan 2016), and graph decomposition approaches (Nikolentzos et al. 2018). Graph kernels were dominant in graph classification for several years, leading to new state-of-the-art results on many classification tasks. However, they are limited by the fact that they cannot effectively adapt their feature representations to a given data distribution, since they generally rely on a fixed set of features. More recently, a number of approaches to graph classification based upon neural networks have been proposed. Most of the neural approaches fit into the graph neural network framework proposed by (Gilmer et al. 2017). Notable instances of this model include *Neural Fingerprints* (Duvenaud et al. 2015), *Gated Graph Neural Networks* (Li et al. 2016), *GraphSAGE* (Hamilton, Ying, and Leskovec 2017a), *SplineCNN* (Fey et al. 2018), and the spectral approaches proposed in (Bruna et al. 2014; Defferrard, X., and Vandergheynst 2016; Kipf and Welling 2017)—all of which descend from early work in (Merkwirth and Lengauer 2005) and (Scarselli et al. 2009b). Recent extensions and improvements to the GNN framework include approaches to incorporate different local structures around subgraphs (Xu et al. 2018) and novel techniques for pooling node representations in order perform graph classification (Zhang et al. 2018; Ying et al. 2018b). GNNs have achieved state-of-the-art performance on several graph classification benchmarks in recent years, see, e.g., (Ying et al. 2018b)—as well as applications such as protein-protein interaction prediction (Fout et al. 2017), recommender systems (Ying et al. 2018a), and the analysis of quantum interactions in molecules (Schütt et al. 2017). A survey of recent advancements in GNN techniques can be found in (Hamilton, Ying, and Leskovec 2017b).

Up to this point (and despite their empirical success) there has been very little theoretical work on GNNs—with the no-

table exceptions of Li et al.'s (Li, Han, and Wu 2018) work connecting GNNs to a special form Laplacian smoothing and Lei et al.'s (Lei et al. 2017) work showing that the feature maps generated by GNNs lie in the same Hilbert space as some popular graph kernels. Moreover, Scarselli et al. (Scarselli et al. 2009a) investigates the approximation capabilities of GNNs.

Preliminaries

We start by fixing notation, and then outline the Weisfeiler-Leman algorithm and the standard graph neural network framework.

Notation and Background

A *graph* G is a pair (V, E) with a finite set of *nodes* V and a set of *edges* $E \subseteq \{\{u, v\} \subseteq V \mid u \neq v\}$. We denote the set of nodes and the set of edges of G by $V(G)$ and $E(G)$, respectively. For ease of notation we denote the edge $\{u, v\}$ in $E(G)$ by (u, v) or (v, u). Moreover, $N(v)$ denotes the *neighborhood* of v in $V(G)$, i.e., $N(v) = \{u \in V(G) \mid (v, u) \in E(G)\}$. We say that two graphs G and H are *isomorphic* if there exists an edge preserving bijection $\varphi : V(G) \to V(H)$, i.e., (u, v) is in $E(G)$ if and only if $(\varphi(u), \varphi(v))$ is in $E(H)$. We write $G \simeq H$ and call the equivalence classes induced by $\simeq$ *isomorphism types*. Let $S \subseteq V(G)$ then $G[S] = (S, E_S)$ is the *subgraph induced* by S with $E_S = \{(u, v) \in E(G) \mid u, v \in S\}$. A *node coloring* is a function $V(G) \to \Sigma$ with arbitrary codomain Σ. Then a *node colored* or *labeled graph* (G, l) is a graph G endowed with a node coloring $l \colon V(G) \to \Sigma$. We say that $l(v)$ is a *label* or *color* of $v \in V(G)$. We say that a node coloring c *refines* a node coloring d, written $c \sqsubseteq d$, if $c(v) = c(w)$ implies $d(v) = d(w)$ for every v, w in $V(G)$. Two colorings are *equivalent* if $c \sqsubseteq d$ and $d \sqsubseteq c$, and we write $c \equiv d$. A *color class* $Q \subseteq V(G)$ of a node coloring c is a maximal set of nodes with $c(v) = c(w)$ for every v, w in Q. Moreover, let $[1 : n] = \{1, \ldots, n\} \subset \mathbb{N}$ for $n > 1$, let S be a set then the set of *k-sets* $[S]^k = \{U \subseteq S \mid |U| = k\}$ for $k \geq 2$, which is the set of all subsets with cardinality k, and let $\{\!\{\ldots\}\!\}$ denote a multiset.

Weisfeiler-Leman Algorithm

We now describe the 1-WL algorithm for labeled graphs. Let (G, l) be a labeled graph. In each iteration, $t \geq 0$, the 1-WL computes a node coloring $c_l^{(t)} \colon V(G) \to \Sigma$, which depends on the coloring from the previous iteration. In iteration 0, we set $c_l^{(0)} = l$. Now in iteration $t > 0$, we set

$$c_l^{(t)}(v) = \textsc{hash}\Big(\big(c_l^{(t-1)}(v), \{\!\{c_l^{(t-1)}(u) \mid u \in N(v)\}\!\}\big)\Big) \quad (1)$$

where HASH bijectively maps the above pair to a unique value in Σ, which has not been used in previous iterations. To test two graph G and H for isomorphism, we run the above algorithm in "parallel" on both graphs. Now if the two graphs have a different number of nodes colored σ in Σ, the 1-WL concludes that the graphs are not isomorphic. Moreover, if the number of colors between two iterations does not change, i.e., the cardinalities of the images of $c_l^{(t-1)}$ and $c_l^{(t)}$ are equal, the algorithm terminates. Termination is guaranteed after at most $\max\{|V(G)|, |V(H)|\}$ iterations. It is easy to see that the algorithm is not able to distinguish all non-isomorphic graphs, e.g., see (Cai, Fürer, and Immerman 1992). Nonetheless, it is a powerful heuristic, which can successfully test isomorphism for a broad class of graphs (Babai and Kucera 1979).

The k-dimensional Weisfeiler-Leman algorithm (k-WL), for $k \geq 2$, is a generalization of the 1-WL which colors tuples from $V(G)^k$ instead of nodes. That is, the algorithm computes a coloring $c_{l,k}^{(t)} \colon V(G)^k \to \Sigma$. In order to describe the algorithm, we define the j-th neighborhood

$$N_j(s) = \{(s_1, \ldots, s_{j-1}, r, s_{j+1}, \ldots, s_k) \mid r \in V(G)\} \quad (2)$$

of a k-tuple $s = (s_1, \ldots, s_k)$ in $V(G)^k$. That is, the j-th neighborhood $N_j(t)$ of s is obtained by replacing the j-th component of s by every node from $V(G)$. In iteration 0, the algorithm labels each k-tuple with its *atomic type*, i.e., two k-tuples s and s' in $V(G)^k$ get the same color if the map $s_i \mapsto s'_i$ induces a (labeled) isomorphism between the subgraphs induced from the nodes from s and s', respectively. For iteration $t > 0$, we define

$$C_j^{(t)}(s) = \textsc{hash}\big(\{\!\{c_{l,k}^{(t-1)}(s') \mid s' \in N_j(s)\}\!\}\big), \quad (3)$$

and set

$$c_{k,l}^{(t)}(s) = \textsc{hash}\Big(\big(c_{k,l}^{(t-1)}(s), \big(C_1^{(t)}(s), \ldots, C_k^{(t)}(s)\big)\big)\Big). \quad (4)$$

Hence, two tuples s and s' with $c_{k,l}^{(t-1)}(s) = c_{k,l}^{(t-1)}(s')$ get different colors in iteration t if there exists j in $[1 : k]$ such that the number of j-neighbors of s and s', respectively, colored with a certain color is different. The algorithm then proceeds analogously to the 1-WL. By increasing k, the algorithm gets more powerful in terms of distinguishing non-isomorphic graphs, i.e., for each $k \geq 2$, there are non-isomorphic graphs which can be distinguished by the $(k + 1)$-WL but not by the k-WL (Cai, Fürer, and Immerman 1992). We note here that the above variant is not equal to the *folklore* variant of k-WL described in (Cai, Fürer, and Immerman 1992), which differs slightly in its update rule. However, it holds that the k-WL using Equation (4) is as powerful as the folklore $(k-1)$-WL (Grohe and Otto 2015).

WL Kernels. After running the WL algorithm, the concatenation of the histogram of colors in each iteration can be used as a feature vector in a kernel computation. Specifically, in the histogram for every color σ in Σ there is an entry containing the number of nodes or k-tuples that are colored with σ.

Graph Neural Networks

Let (G, l) be a labeled graph with an initial node coloring $f^{(0)} : V(G) \to \mathbb{R}^{1\times d}$ that is *consistent* with l. This means that each node v is annotated with a feature $f^{(0)}(v)$ in $\mathbb{R}^{1\times d}$ such that $f^{(0)}(u) = f^{(0)}(v)$ if and only if $l(u) = l(v)$. Alternatively, $f^{(0)}(v)$ can be an arbitrary real-valued feature vector associated with v. Examples include continuous atomic properties in chemoinformatic applications where nodes correspond to atoms, or vector representations of text in social network applications. A GNN model consists of a stack of neural network layers, where each layer aggregates local neighborhood information, i.e., features of neighbors, around each node and then passes this aggregated information on to the next layer.

A basic GNN model can be implemented as follows (Hamilton, Ying, and Leskovec 2017b). In each layer $t > 0$, we compute a new feature

$$f^{(t)}(v) = \sigma\Big(f^{(t-1)}(v)\cdot W_1^{(t)} + \sum_{w\in N(v)} f^{(t-1)}(w)\cdot W_2^{(t)}\Big) \quad (5)$$

in $\mathbb{R}^{1\times e}$ for v, where $W_1^{(t)}$ and $W_2^{(t)}$ are parameter matrices from $\mathbb{R}^{d\times e}$, and σ denotes a component-wise non-linear function, e.g., a sigmoid or a ReLU.[1]

Following (Gilmer et al. 2017), one may also replace the sum defined over the neighborhood in the above equation by a permutation-invariant, differentiable function, and one may substitute the outer sum, e.g., by a column-wise vector concatenation or LSTM-style update step. Thus, in full generality a new feature $f^{(t)}(v)$ is computed as

$$f_{\text{merge}}^{W_1}\Big(f^{(t-1)}(v), f_{\text{aggr}}^{W_2}\big(\{\!\{f^{(t-1)}(w) \mid w \in N(v)\}\!\}\big)\Big), \quad (6)$$

where $f_{\text{aggr}}^{W_1}$ aggregates over the set of neighborhood features and $f_{\text{merge}}^{W_2}$ merges the node's representations from step $(t-1)$ with the computed neighborhood features. Both $f_{\text{aggr}}^{W_1}$ and $f_{\text{merge}}^{W_2}$ may be arbitrary differentiable, permutation-invariant functions (e.g., neural networks), and, by analogy to Equation 5, we denote their parameters as W_1 and W_2, respectively. In the rest of this paper, we refer to neural architectures implementing Equation (6) as 1-*dimensional GNN architectures* (1-GNNs).

A vector representation f_{GNN} over the whole graph can be computed by summing over the vector representations computed for all nodes, i.e.,

$$f_{\text{GNN}}(G) = \sum_{v\in V(G)} f^{(T)}(v),$$

where $T > 0$ denotes the last layer. More refined approaches use differential pooling operators based on sorting (Zhang et al. 2018) and soft assignments (Ying et al. 2018b).

In order to adapt the parameters W_1 and W_2 of Equations (5) and (6), to a given data distribution, they are optimized in an end-to-end fashion (usually via stochastic gradient descent) together with the parameters of a neural network used for classification or regression.

Relationship Between 1-WL and 1-GNNs

In the following we explore the relationship between the 1-WL and 1-GNNs. Let (G, l) be a labeled graph, and let $\mathbf{W}^{(t)} = \big(W_1^{(t')}, W_2^{(t')}\big)_{t'\leq t}$ denote the GNN parameters given by Equation (5) or Equation (6) up to iteration t. We encode the initial labels $l(v)$ by vectors $f^{(0)}(v) \in \mathbb{R}^{1\times d}$, e.g., using a 1-hot encoding.

Our first theoretical result shows that the 1-GNN architectures do not have more power in terms of distinguishing between non-isomorphic (sub-)graphs than the 1-WL algorithm. More formally, let $f_{\text{merge}}^{W_1}$ and $f_{\text{aggr}}^{W_2}$ be any two functions chosen in (6). For every encoding of the labels $l(v)$ as vectors $f^{(0)}(v)$, and for every choice of $\mathbf{W}^{(t)}$, we have that the coloring $c_l^{(t)}$ of 1-WL always refines the coloring $f^{(t)}$ induced by a 1-GNN parameterized by $\mathbf{W}^{(t)}$.

Theorem 1. Let (G, l) be a labeled graph. Then for all $t \geq 0$ and for all choices of initial colorings $f^{(0)}$ consistent with l, and weights $\mathbf{W}^{(t)}$,

$$c_l^{(t)} \sqsubseteq f^{(t)}\,.$$

Our second result states that there exist a sequence of parameter matrices $\mathbf{W}^{(t)}$ such that 1-GNNs have exactly the same power in terms of distinguishing non-isomorphic (sub-)graphs as the 1-WL algorithm. This even holds for the simple architecture (5), provided we choose the encoding of the initial labeling l in such a way that different labels are encoded by linearly independent vectors.

Theorem 2. Let (G, l) be a labeled graph. Then for all $t \geq 0$ there exists a sequence of weights $\mathbf{W}^{(t)}$, and a 1-GNN architecture such that

$$c_l^{(t)} \equiv f^{(t)}\,.$$

Hence, in the light of the above results, 1-GNNs may viewed as an extension of the 1-WL which in principle have the same power but are more flexible in their ability to adapt to the learning task at hand and are able to handle continuous node features.

Shortcomings of Both Approaches

The power of 1-WL has been completely characterized, see, e.g., (Arvind et al. 2015). Hence, by using Theorems 1 and 2, this characterization is also applicable to 1-GNNs. On the other hand, 1-GNNs have the same shortcomings as the 1-WL. For example, both methods will give the same color to every node in a graph consisting of a triangle and a 4-cycle, although vertices from the triangle and the vertices from the 4-cycle are clearly different. Moreover, they are not capable of capturing simple graph theoretic properties, e.g., triangle counts, which are an important measure in social network analysis (Milo et al. 2002; Newman 2003).

k-dimensional Graph Neural Networks

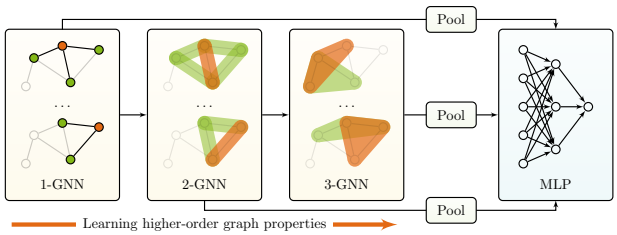
In the following, we propose a generalization of 1-GNNs, so-called k-GNNs, which are based on the k-WL. Due to scalability and limited GPU memory, we consider a set-based version of the k-WL. For a given k, we consider all k-element subsets $[V(G)]^k$ over $V(G)$. Let $s = \{s_1, \ldots, s_k\}$ be a k-set in $[V(G)]^k$, then we define the *neighborhood* of s as

$$N(s) = \{t \in [V(G)]^k \mid |s \cap t| = k - 1\}\,.$$

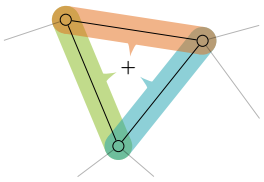
The *local neighborhood* $N_L(s)$ consists of all $t \in N(s)$ such that $(v, w) \in E(G)$ for the unique $v \in s \setminus t$ and the unique $w \in t \setminus s$. The *global neighborhood* $N_G(s)$ then is defined as $N(s) \setminus N_{\text{L}}(s)$.[2]

[1] For clarity of presentation we omit biases.

[2] Note that the definition of the local neighborhood is different from the the one defined in (Morris, Kersting, and Mutzel 2017) which is a superset of our definition. Our computations therefore involve sparser graphs.



(a) Hierarchical 1-2-3-GNN network architecture



(b) Pooling from 2- to 3-GNN.

Figure 1: Illustration of the proposed hierarchical variant of the k-GNN layer. For each subgraph S on k nodes a feature f is learned, which is initialized with the learned features of all $(k-1)$-element subgraphs of S. Hence, a hierarchical representation of the input graph is learned.

The set based k-WL works analogously to the k-WL, i.e., it computes a coloring $c^{(t)}_{\mathrm{s},k,l} : [V(G)]^k \to \Sigma$ as in Equation (1) based on the above neighborhood. Initially, $c^{(0)}_{\mathrm{s},k,l}$ colors each element s in $[V(G)]^k$ with the isomorphism type of $G[s]$.

Let (G, l) be a labeled graph. In each k-GNN layer $t \geq 0$, we compute a feature vector $f^{(t)}_k(s)$ for each k-set s in $[V(G)]^k$. For $t = 0$, we set $f^{(0)}_k(s)$ to $f^{\mathrm{iso}}(s)$, a one-hot encoding of the isomorphism type of $G[s]$ labeled by l. In each layer $t > 0$, we compute new features by

$$f^{(t)}_k(s) = \sigma\Big(f^{(t-1)}_k(s) \cdot W^{(t)}_1 + \sum_{u \in N_L(s) \cup N_G(s)} f^{(t-1)}_k(u) \cdot W^{(t)}_2\Big).$$

Moreover, one could split the sum into two sums ranging over $N_L(s)$ and $N_G(s)$ respectively, using distinct parameter matrices to enable the model to learn the importance of local and global neighborhoods. To scale k-GNNs to larger datasets and to prevent overfitting, we propose *local* k-GNNs, where we omit the global neighborhood of s, i.e.,

$$f^{(t)}_{k,\mathrm{L}}(s) = \sigma\Big(f^{(t-1)}_{k,\mathrm{L}}(s) \cdot W^{(t)}_1 + \sum_{u \in N_L(s)} f^{(t-1)}_{k,\mathrm{L}}(u) \cdot W^{(t)}_2\Big).$$

The running time for evaluation of the above depends on $|V|$, k and the sparsity of the graph (each iteration can be bounded by the number of subsets of size k times the maximum degree). Note that we can scale our method to larger datasets by using sampling strategies introduced in, e.g., (Morris, Kersting, and Mutzel 2017; Hamilton, Ying, and Leskovec 2017a). We can now lift the results of the previous section to the k-dimensional case.

Proposition 3. Let (G, l) be a labeled graph and let $k \geq 2$. Then for all $t \geq 0$, for all choices of initial colorings $f^{(0)}_k$ consistent with l and for all weights $\mathbf{W}^{(t)}$,

$$c^{(t)}_{\mathrm{s},k,l} \sqsubseteq f^{(t)}_k.$$

Again the second result states that there exists a suitable initialization of the parameter matrices $\mathbf{W}^{(t)}$ such that k-GNNs have exactly the same power in terms of distinguishing non-isomorphic (sub-)graphs as the set-based k-WL.

Proposition 4. Let (G, l) be a labeled graph and let $k \geq 2$. Then for all $t \geq 0$ there exists a sequence of weights $\mathbf{W}^{(t)}$, and a k-GNN architecture such that

$$c^{(t)}_{\mathrm{s},k,l} \equiv f^{(t)}_k.$$

Hierarchical Variant

One key benefit of the end-to-end trainable k-GNN framework—compared to the discrete k-WL algorithm—is that we can hierarchically combine representations learned at different granularities. Concretely, rather than simply using one-hot indicator vectors as initial feature inputs in a k-GNN, we propose a *hierarchical* variant of k-GNN that uses the features learned by a $(k-1)$-dimensional GNN, in addition to the (labeled) isomorphism type, as the initial features, i.e.,

$$f^{(0)}_k(s) = \sigma\Big(\Big[f^{\mathrm{iso}}(s), \sum_{u \subset s} f^{(T_{k-1})}_{k-1}(u)\Big] \cdot W_{k-1}\Big),$$

for some $T_{k-1} > 0$, where W_{k-1} is a matrix of appropriate size, and square brackets denote matrix concatenation.

Hence, the features are recursively learned from dimensions 1 to k in an end-to-end fashion. This hierarchical model also satisfies Propositions 3 and 4, so its representational capacity is theoretically equivalent to a standard k-GNN (in terms of its relationship to k-WL). Nonetheless, hierarchy is a natural inductive bias for graph modeling, since many real-world graphs incorporate hierarchical structure, so we expect this hierarchical formulation to offer empirical utility.

Experimental Study

In the following, we want to investigate potential benefits of GNNs over graph kernels as well as the benefits of our proposed k-GNN architectures over 1-GNN architectures. More precisely, we address the following questions:

Q1 How do the (hierarchical) k-GNNs perform in comparison to state-of-the-art graph kernels?

Q2 How do the (hierarchical) k-GNNs perform in comparison to the 1-GNN in graph classification and regression tasks?

Q3 How much (if any) improvement is provided by optimizing the parameters of the GNN aggregation function, compared to just using random GNN parameters while optimizing the parameters of the downstream classification/regression algorithm?

Datasets

To compare our k-GNN architectures to kernel approaches we use well-established benchmark datasets from the graph kernel literature (Kersting et al. 2016). The nodes of each graph in these dataset is annotated with (discrete) labels or no labels.

Table 1: Classification accuracies in percent on various graph benchmark datasets.

	Method	Dataset						
		PRO	IMDB-BIN	IMDB-MUL	PTC-FM	NCI1	MUTAG	PTC-MR
Kernel	GRAPHLET	72.9	59.4	40.8	58.3	72.1	87.7	54.7
	SHORTEST-PATH	**76.4**	59.2	40.5	62.1	74.5	81.7	58.9
	1-WL	73.8	72.5	**51.5**	62.9	83.1	78.3	61.3
	2-WL	75.2	72.6	50.6	**64.7**	77.0	77.0	61.9
	3-WL	74.7	73.5	49.7	61.5	83.1	83.2	62.5
	WL-OA	75.3	73.1	50.4	62.7	**86.1**	84.5	**63.6**
GNN	DCNN	61.3	49.1	33.5	—	62.6	67.0	56.6
	PATCHYSAN	75.9	71.0	45.2	—	78.6	**92.6**	60.0
	DGCNN	75.5	70.0	47.8	—	74.4	85.8	58.6
	1-GNN NO TUNING	70.7	69.4	47.3	59.0	58.6	82.7	51.2
	1-GNN	72.2	71.2	47.7	59.3	74.3	82.2	59.0
	1-2-3-GNN NO TUNING	75.9	70.3	48.8	60.0	67.4	84.4	59.3
	1-2-3-GNN	75.5	**74.2**	49.5	62.8	76.2	86.1	60.9

Table 2: Mean absolute errors on the QM9 dataset. The far-right column shows the improvement of the best k-GNN model in comparison to the 1-GNN baseline.

Target	Method						
	DTNN (Wu et al. 2018)	MPNN (Wu et al. 2018)	1-GNN	1-2-GNN	1-3-GNN	1-2-3-GNN	Gain
μ	**0.244**	0.358	0.493	0.493	**0.473**	0.476	4.0%
α	0.95	0.89	0.78	**0.27**	0.46	**0.27**	65.3%
$\varepsilon_{\text{HOMO}}$	0.00388	0.00541	**0.00321**	0.00331	0.00328	0.00337	–
$\varepsilon_{\text{LUMO}}$	0.00512	0.00623	0.00355	**0.00350**	0.00354	0.00351	1.4%
$\Delta\varepsilon$	0.0112	0.0066	0.0049	0.0047	**0.0046**	0.0048	6.1%
$\langle R^2 \rangle$	**17.0**	28.5	34.1	21.5	25.8	22.9	37.0%
ZPVE	0.00172	0.00216	0.00124	**0.00018**	0.00064	0.00019	85.5%
U_0	2.43	2.05	2.32	**0.0357**	0.6855	0.0427	98.5%
U	2.43	2.00	2.08	**0.107**	0.686	0.111	94.9%
H	2.43	2.02	2.23	0.070	0.794	**0.0419**	98.1%
G	2.43	2.02	1.94	0.140	0.587	**0.0469**	97.6%
C_{v}	0.27	0.42	0.27	0.0989	0.158	**0.0944**	65.0%

To demonstrate that our architectures scale to larger datasets and offer benefits on real-world applications, we conduct experiments on the QM9 dataset (Ramakrishnan et al. 2014; Ruddigkeit et al. 2012; Wu et al. 2018), which consists of 133 385 small molecules. The aim here is to perform regression on twelve targets representing energetic, electronic, geometric, and thermodynamic properties, which were computed using density functional theory.

Baselines

We use the following kernel and GNN methods as baselines for our experiments.

Kernel Baselines. We use the Graphlet kernel (Shervashidze et al. 2009), the shortest-path kernel (Borgwardt and Kriegel 2005), the Weisfeiler-Lehman subtree kernel (WL) (Shervashidze et al. 2011), the Weisfeiler-Lehman Optimal Assignment kernel (WL-OA) (Kriege, Giscard, and Wilson 2016), and the global-local k-WL (Morris, Kersting, and Mutzel 2017) with k in $\{2, 3\}$ as kernel baselines. For each kernel, we computed the normalized Gram matrix. We used the C-SVM implementation of LIBSVM (Chang and Lin 2011) to compute the classification accuracies using 10-fold cross validation. The parameter C was selected from $\{10^{-3}, 10^{-2}, \ldots, 10^{2}, 10^{3}\}$ by 10-fold cross validation on the training folds.

Neural Baselines. To compare GNNs to kernels we used the basic 1-GNN layer of Equation (5), DCNN (Wang et al. 2018), PatchySan (Niepert, Ahmed, and Kutzkov 2016), DGCNN (Zhang et al. 2018). For the QM9 dataset we used a 1-GNN layer similar to (Gilmer et al. 2017), where we replaced the inner sum of Equation (5) with a 2-layer MLP in order incorporate edge features (bond type and distance information). Moreover, we compare against the numbers provided in (Wu et al. 2018).

Model Configuration

We always used three layers for 1-GNN, and two layers for (local) 2-GNN and 3-GNN, all with a hidden-dimension size of 64. For the hierarchical variant we used architectures that use features computed by 1-GNN as initial features for the 2-GNN (1-2-GNN) and 3-GNN (1-3-GNN), respectively. Moreover, us-

ing the combination of the former we componentwise concatenated the computed features of the 1-2-GNN and the 1-3-GNN (1-2-3-GNN). For the final classification and regression steps, we used a three layer MLP, with binary cross entropy and mean squared error for the optimization, respectively. For classification we used a dropout layer with $p = 0.5$ after the first layer of the MLP. We applied global average pooling to generate a vector representation of the graph from the computed node features for each k. The resulting vectors are concatenated column-wise before feeding them into the MLP. Moreover, we used the Adam optimizer with an initial learning rate of 10^{-2} and applied an adaptive learning rate decay based on validation results to a minimum of 10^{-5}. We trained the classification networks for 100 epochs and the regression networks for 200 epochs.

Experimental Protocol

For the smaller datasets, which we use for comparison against the kernel methods, we performed a 10-fold cross validation where we randomly sampled 10% of each training fold to act as a validation set. For the QM9 dataset, we follow the dataset splits described in (Wu et al. 2018). We randomly sampled 10% of the examples for validation, another 10% for testing, and used the remaining for training. We used the same initial node features as described in (Gilmer et al. 2017). Moreover, in order to illustrate the benefits of our hierarchical k-GNN architecture, we did not use a complete graph, where edges are annotated with pairwise distances, as input. Instead, we only used pairwise Euclidean distances for connected nodes, computed from the provided node coordinates. The code was built upon the work of (Fey et al. 2018) and is provided at https://github.com/chrsmrrs/k-gnn.

Results and Discussion

In the following we answer questions **Q1** to **Q3**. Table 1 shows the results for comparison with the kernel methods on the graph classification benchmark datasets. Here, the hierarchical k-GNN is on par with the kernels despite the small dataset sizes (answering question **Q1**). We also find that the 1-2-3-GNN significantly outperforms the 1-GNN on all seven datasets (answering **Q2**), with the 1-GNN being the overall weakest method across all tasks.[3] We can further see that optimizing the parameters of the aggregation function only leads to slight performance gains on two out of three datasets, and that no optimization even achieves better results on the PROTEINS benchmark dataset (answering **Q3**). We contribute this effect to the one-hot encoded node labels, which allow the GNN to gather enough information out of the neighborhood of a node, even when this aggregation is not learned.

Table 2 shows the results for the QM9 dataset. On eleven out of twelve targets all of our hierarchical variants beat the 1-GNN baseline, providing further evidence for **Q2**. For example, on the target H we achieve a large improvement of 98.1% in MAE compared to the baseline. Moreover, on ten out of twelve datasets, the hierarchical k-GNNs beat the baselines from (Wu et al. 2018). However, the additional structural information extracted by the k-GNN layers does not serve all tasks equally, leading to huge differences in gains across the targets.

It should be noted that our k-GNN models have more parameters than the 1-GNN model, since we stack two additional GNN layers for each k. However, extending the 1-GNN model by additional layers to match the number of parameters of the k-GNN did not lead to better results in any experiment.

[3] Note that in very recent work, GNNs have shown superior results over kernels when using advanced pooling techniques (Ying et al. 2018b). Note that our layers can be combined with these pooling layers. However, we opted to use standard global pooling in order to compare a typical GNN implementation with standard off-the-shelf kernels.

Conclusion

We presented a theoretical investigation of GNNs, showing that a wide class of GNN architectures cannot be stronger than the 1-WL. On the positive side, we showed that, in principle, GNNs possess the same power in terms of distinguishing between non-isomorphic (sub-)graphs, while having the added benefit of adapting to the given data distribution. Based on this insight, we proposed k-GNNs which are a generalization of GNNs based on the k-WL. This new model is strictly stronger then GNNs in terms of distinguishing non-isomorphic (sub-)graphs and is capable of distinguishing more graph properties. Moreover, we devised a hierarchical variant of k-GNNs, which can exploit the hierarchical organization of most real-world graphs. Our experimental study shows that k-GNNs consistently outperform 1-GNNs and beat state-of-the-art neural architectures on large-scale molecule learning tasks. Future work includes designing task-specific k-GNNs, e.g., devising k-GNNs layers that exploit expert-knowledge in bio- and chemoinformatic settings.

Acknowledgments

This work is supported by the German research council (DFG) within the Research Training Group 2236 *UnRAVeL* and the Collaborative Research Center SFB 876, *Providing Information by Resource-Constrained Analysis*, projects A6 and B2.

References

Arvind, V.; Köbler, J.; Rattan, G.; and Verbitsky, O. 2015. On the power of color refinement. In *Symposium on Fundamentals of Computation Theory*, 339–350.

Babai, L., and Kucera, L. 1979. Canonical labelling of graphs in linear average time. In *Symposium on Foundations of Computer Science*, 39–46.

Borgwardt, K. M., and Kriegel, H.-P. 2005. Shortest-path kernels on graphs. In *ICDM*, 74–81.

Bruna, J.; Zaremba, W.; Szlam, A.; and LeCun, Y. 2014. Spectral networks and deep locally connected networks on graphs. In *ICLR*.

Cai, J.; Fürer, M.; and Immerman, N. 1992. An optimal lower bound on the number of variables for graph identifications. *Combinatorica* 12(4):389–410.

Chang, C.-C., and Lin, C.-J. 2011. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* 2:27:1–27:27.

Defferrard, M.; X., B.; and Vandergheynst, P. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, 3844–3852.

Duvenaud, D. K.; Maclaurin, D.; Iparraguirre, J.; Bombarell, R.; Hirzel, T.; Aspuru-Guzik, A.; and Adams, R. P. 2015. Convolutional networks on graphs for learning molecular fingerprints. In *NIPS*, 2224–2232.

Fey, M.; Lenssen, J. E.; Weichert, F.; and Müller, H. 2018. SplineCNN: Fast geometric deep learning with continuous B-spline kernels. In *CVPR*.

Fout, A.; Byrd, J.; Shariat, B.; and Ben-Hur, A. 2017. Protein interface prediction using graph convolutional networks. In *NIPS*, 6533–6542.

Gärtner, T.; Flach, P.; and Wrobel, S. 2003. On graph kernels: Hardness results and efficient alternatives. In *Learning Theory and Kernel Machines*. 129–143.

Gilmer, J.; Schoenholz, S. S.; Riley, P. F.; Vinyals, O.; and Dahl, G. E. 2017. Neural message passing for quantum chemistry. In *ICML*.

Grohe, M., and Otto, M. 2015. Pebble games and linear equations. *Journal of Symbolic Logic* 80(3):797–844.

Grohe, M. 2017. *Descriptive Complexity, Canonisation, and Definable Graph Structure Theory*. Lecture Notes in Logic. Cambridge University Press.

Hamilton, W. L.; Ying, R.; and Leskovec, J. 2017a. Inductive representation learning on large graphs. In *NIPS*, 1025–1035.

Hamilton, W. L.; Ying, R.; and Leskovec, J. 2017b. Representation learning on graphs: Methods and applications. *IEEE Data Engineering Bulletin* 40(3):52–74.

Johansson, F. D., and Dubhashi, D. 2015. Learning with similarity functions on graphs using matchings of geometric embeddings. In *KDD*, 467–476.

Kashima, H.; Tsuda, K.; and Inokuchi, A. 2003. Marginalized kernels between labeled graphs. In *ICML*, 321–328.

Kersting, K.; Kriege, N. M.; Morris, C.; Mutzel, P.; and Neumann, M. 2016. Benchmark data sets for graph kernels.

Kiefer, S.; Schweitzer, P.; and Selman, E. 2015. Graphs identified by logics with counting. In *MFCS*, 319–330. Springer.

Kipf, T. N., and Welling, M. 2017. Semi-supervised classification with graph convolutional networks. In *ICLR*.

Kondor, R., and Pan, H. 2016. The multiscale laplacian graph kernel. In *NIPS*, 2982–2990.

Kriege, N. M.; Giscard, P.-L.; and Wilson, R. C. 2016. On valid optimal assignment kernels and applications to graph classification. In *NIPS*, 1615–1623.

Lei, T.; Jin, W.; Barzilay, R.; and Jaakkola, T. S. 2017. Deriving neural architectures from sequence and graph kernels. In *ICML*, 2024–2033.

Li, W.; Saidi, H.; Sanchez, H.; Schäf, M.; and Schweitzer, P. 2016. Detecting similar programs via the Weisfeiler-Leman graph kernel. In *International Conference on Software Reuse*, 315–330.

Li, Q.; Han, Z.; and Wu, X.-M. 2018. Deeper insights into graph convolutional networks for semi-supervised learning. In *AAAI*, 3538–3545.

Merkwirth, C., and Lengauer, T. 2005. Automatic generation of complementary descriptors with molecular graph networks. *Journal of Chemical Information and Modeling* 45(5):1159–1168.

Milo, R.; Shen-Orr, S.; Itzkovitz, S.; Kashtan, N.; Chklovskii, D.; and Alon, U. 2002. Network motifs: simple building blocks of complex networks. *Science* 298(5594):824–827.

Morris, C.; Kersting, K.; and Mutzel, P. 2017. Glocalized Weisfeiler-Lehman kernels: Global-local feature maps of graphs. In *ICDM*, 327–336.

Newman, M. E. J. 2003. The structure and function of complex networks. *SIAM review* 45(2):167–256.

Niepert, M.; Ahmed, M.; and Kutzkov, K. 2016. Learning convolutional neural networks for graphs. In *ICML*, 2014–2023.

Nikolentzos, G.; Meladianos, P.; Limnios, S.; and Vazirgiannis, M. 2018. A degeneracy framework for graph similarity. In *IJCAI*, 2595–2601.

Nikolentzos, G.; Meladianos, P.; and Vazirgiannis, M. 2017. Matching node embeddings for graph similarity. In *AAAI*, 2429–2435.

Ramakrishnan, R.; Dral, P., O.; Rupp, M.; and von Lilienfeld, O. A. 2014. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific Data* 1.

Ruddigkeit, L.; van Deursen, R.; Blum, L. C.; and Reymond, J.-L. 2012. Enumeration of 166 billion organic small molecules in the chemical universe database gdb-17. *Journal of Chemical Information and Modeling* 52 11:2864–75.

Scarselli, F.; Gori, M.; Tsoi, A. C.; Hagenbuchner, M.; and Monfardini, G. 2009a. Computational capabilities of graph neural networks. *IEEE Transactions on Neural Networks* 20(1):81–102.

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

Schütt, K.; Kindermans, P. J.; Sauceda, H. E.; Chmiela, S.; Tkatchenko, A.; and Müller, K. R. 2017. SchNet: A continuous-filter convolutional neural network for modeling quantum interactions. In *NIPS*, 992–1002.

Shervashidze, N.; Vishwanathan, S. V. N.; Petri, T. H.; Mehlhorn, K.; and Borgwardt, K. M. 2009. Efficient graphlet kernels for large graph comparison. In *AISTATS*, 488–495.

Shervashidze, N.; Schweitzer, P.; van Leeuwen, E. J.; Mehlhorn, K.; and Borgwardt, K. M. 2011. Weisfeiler-Lehman graph kernels. *JMLR* 12:2539–2561.

Vishwanathan, S. V. N.; Schraudolph, N. N.; Kondor, R.; and Borgwardt, K. M. 2010. Graph kernels. *JMLR* 11:1201–1242.

Wang, Y.; Sun, Y.; Liu, Z.; Sarma, S. E.; Bronstein, M. M.; and Solomon, J. M. 2018. Dynamic graph CNN for learning on point clouds. *CoRR* abs/1801.07829.

Wu, Z.; Ramsundar, B.; Feinberg, E. N.; Gomes, J.; Geniesse, C.; Pappu, A. S.; Leswing, K.; and Pande, V. 2018. Moleculenet: a benchmark for molecular machine learning. *Chemical Science* 9:513–530.

Xu, K.; Li, C.; Tian, Y.; Sonobe, T.; Kawarabayashi, K.-i.; and Jegelka, S. 2018. Representation learning on graphs with jumping knowledge networks. In *ICML*, 5453–5462.

Yanardag, P., and Vishwanathan, S. V. N. 2015a. Deep graph kernels. In *KDD*, 1365–1374.

Yanardag, P., and Vishwanathan, S. V. N. 2015b. A structural smoothing framework for robust graph comparison. In *NIPS*, 2134–2142.

Ying, R.; He, R.; Chen, K.; Eksombatchai, P.; Hamilton, W. L.; and Leskovec, J. 2018a. Graph convolutional neural networks for web-scale recommender systems. *KDD*.

Ying, R.; You, J.; Morris, C.; Ren, X.; Hamilton, W. L.; and Leskovec, J. 2018b. Hierarchical graph representation learning with differentiable pooling. In *NIPS*.

Zhang, M.; Cui, Z.; Neumann, M.; and Yixin, C. 2018. An end-to-end deep learning architecture for graph classification. In *AAAI*, 4428–4435.