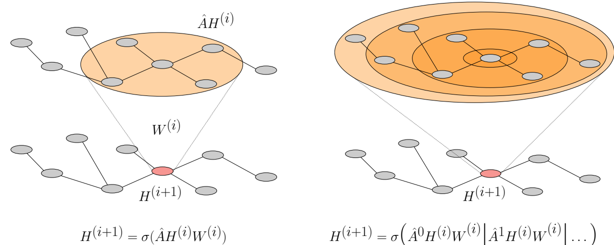
MixHop: Higher-Order Graph Convolutional Architectures via Sparsified Neighborhood Mixing

Sami Abu-El-Haija [1] **Bryan Perozzi** [2] **Amol Kapoor** [2] **Nazanin Alipourfard** [1] **Kristina Lerman** [1] **Hrayr Harutyunyan** [1] **Greg Ver Steeg** [1] **Aram Galstyan** [1]

Abstract

Existing popular methods for semi-supervised learning with Graph Neural Networks (such as the Graph Convolutional Network) provably cannot learn a general class of neighborhood mixing relationships. To address this weakness, we propose a new model, MixHop, that can learn these relationships, including difference operators, by repeatedly mixing feature representations of neighbors at various distances. MixHop requires no additional memory or computational complexity, and outperforms on challenging baselines. In addition, we propose sparsity regularization that allows us to visualize how the network prioritizes neighborhood information across different graph datasets. Our analysis of the learned architectures reveals that neighborhood mixing varies per datasets.



(a) Traditional graph convolution. (b) Our mixed feature model.

Figure 1: Feature propagation in traditional graph convolution methods (a), compared to ours (b). We show the latent feature for red node in layer $i+1$ given node features in layer i. The traditional graph convolution case only aggregates from immediate neighbors $\hat{A}H^{(i)}$. In our MixHop the feature vector $H^{(i+1)}$ is a learned combination of the node's neighbors $\hat{A}^j H^{(i)}$ at multiple distances j.

1. Introduction

Convolutional Neural Networks (CNNs) establish state-of-the-art performance for many Computer Vision applications (Krizhevsky et al., 2012; Szegedy et al., 2015). CNNs consist of a series of *convolutional layers*, each parameterized by a *filter* with pre-specified spatial dimensions. CNNs are powerful because they are able to learn a hierarchy of translation invariant feature detectors.

The success of CNNs in Computer Vision and other domains has motivated researchers (Bruna et al., 2014; Defferrard et al., 2016; Kipf & Welling, 2017) to extend the convolutional operator from regular grids, in which the structure is fixed and repeated everywhere, to graph-structured data, where nodes' neighborhoods can greatly vary in structure across the graph. Generalizing convolution to graph structures should allow models to learn location-invariant node and neighborhood features.

Early extensions of the graph convolution (GC) operator were theoretically motivated (Bruna et al., 2014), but (1) required quadratic computational complexity in number of nodes and therefore were not scalable to large graphs, and (2) required the graph to be completely observed during training, targeting only the *transductive* setting. Defferrard et al. (2016) and Kipf & Welling (2017) propose GC *approximations* that are computationally-efficient (linear complexity, in the number of edges), and can be applied in *inductive* settings, where the test graphs are not observed during training.

However, said approximations limit the representational capacity of the model. In particular, if we represent an image as a graph of pixel nodes, where edges connect adjacent pixels, GC approximations applied on the pixel graphs will be unable to learn Gabor-like[1] filters. Gabor filters are fundamental to the human visual cognitive system (Daugman, 1980; 1985). Further, these filters are automatically recovered by training CNNs on natural images (see Krizhevsky et al. (2012); Lee et al. (2009) for visualizations). Their

[1]Information Sciences Institute, University of Southern California [2]Google AI, New York. Correspondence to: Sami <sami@haija.org>, Bryan <bperozzi@acm.org>.

Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019. Copyright 2019 by the author(s).

[1]We use "*like*", as graph edges are not axis-aligned.

automatic recovery implies their usefulness for hierarchical object representations and scene understanding, as guided by the optimization (e.g. classification) objective. Since Graphs are generic data structures that can encode data from various domains (e.g. images, chemical compounds, social, and biological networks), realizing Gabor-like filters in Graph domains ought to yield a general advantage.

In this work, we address the limitations of the approximations that prevent these models from capturing the graph analogue of Gabor filters. Our proposed method, MixHop, allows full linear mixing of neighborhood information (as illustrated in Figure 1), at every message passing step. Specifically, our contributions are the following:

- We formalize *Delta Operators* and their generalization, *Neighborhood Mixing*, to analyze the expressiveness of graph convolution models. We show that popular graph convolution models (e.g. GCN of Kipf & Welling (2017)) cannot learn these representations.
- We propose MixHop, a new Graph Convolutional layer that mixes powers of the adjacency matrix. We prove that MixHop can learn a wider class of representations without increasing the memory footprint or computational complexity of previous GCN models.
- We provide a method of learning to divide modeling capacity among various widths and depths of a MixHop model, yielding powerful compact GCN architectures. These architectures conveniently also allow visual inspection of which aspects of a graph are important.

We demonstrate our method on node classification tasks. Our code is available on github.com/samihaija/mixhop.

2. Preliminaries and Related Work

2.1. Notation

Graph G with n nodes and m edges has a feature matrix $X \in \mathbb{R}^{n \times s_0}$ with s_0 features per node, and training labels Y_I, annotating a partial set of nodes with the c possible classes. The output of the task, Y_O, is an assignment of labels to the nodes, $Y \in [0,1]^{n \times c}$. Let A denote the adjacency matrix of G, where a non-zero entry A_{ij} indicates an edge between nodes i and j. We consider the case of a binary adjacency matrix ($A \in \{0,1\}^{n \times n}$), but this notation can be extended w.l.o.g. to weighted graphs. Let I_n be the $n \times n$ identity matrix, and D be the degree matrix, $D = \text{diag}(d)$, where $d \in \mathbb{Z}^n$ is the degree vector with $d_j = \sum_i A_{ij}$.

2.2. Message Passing

Message Passing algorithms can be used to learn models over graphs (Gilmer et al., 2017). In such models, each graph node (and optionally edge) holds a latent vector, initialized to the node's input features Each node repeatedly *passes* its current latent vector to, and aggregates incoming messages from, its immediate neighbors. After l steps of message passing and feature aggregation, every node outputs a representation which can be used for an upstream task e.g. node classification, or entire graph classification. The l steps (message passing and aggregation) can be parametrized and trained via Backprop-Through-Structure algorithms (Goller & Kuchler, 1996), to minimize an objective measured using the node representations as output by the l'th step.

2.3. Graph Convolutional Networks

We refer to the Graph Convolutional Network proposed by Kipf & Welling (2017) as the *vanilla GCN*. The vanilla GCN Graph Convolutional (GC) Layer is defined as:

$$H^{(i+1)} = \sigma(\widehat{A} H^{(i)} W^{(i)}), \tag{1}$$

where $H^{(i)} \in \mathbb{R}^{n \times s_i}$ and $H^{(i+1)} \in \mathbb{R}^{n \times s_{i+1}}$ are the input and output activations for layer i, $W^{(i)} \in \mathbb{R}^{s_i \times s_{i+1}}$ is a trainable weight matrix and σ is an element-wise activation function, and $\widehat{A}$ is a symmetrically normalized adjacency matrix with self-connections, $\widehat{A} = D^{-\frac{1}{2}}(A + I_n)D^{-\frac{1}{2}}$. A GCN model with l layers is then defined as:

$$H^{(i)} = \begin{cases} X & \text{if } i = 0 \\ \sigma(\widehat{A} H^{(i-1)} W^{(i-1)}) & \text{if } i \in [1 \,..\, l], \end{cases} \tag{2}$$

and the output Y_O can be set as to function of $H^{(l)}$. The vanilla GCN can be described as a message passing algorithm, where a node's latent representation at step i is defined as an *average* of its neighbors' representations from step $i-1$, multiplied by $W^{(i-1)}$. See Gilmer et al. (2017).

The vanilla GCN makes three simplifying assumptions: (1) it is a Chebyshev rank-2 approximation of multiplication in the Graph Fourier basis, defined to be the eigenbasis of the graph Laplacian; (2) it assumes that the two coefficients of the Chebyshev polynomials multiply to -1; (3) a *renormalization trick* adds self-connections (identity matrix) to A before, rather than after, normalization. These simplifications reduce the computational complexity and prevent exploding/vanishing gradients. However, it simplifies the definition of convolution to become a simple neighborhood-averaging operator: this is obvious from Equation 1 – the features are left-multiplied by normalized adjacency $\widehat{A}$, effectively replacing each row in the feature matrix, by the average of its neighbors (and itself, due to *renormalization*).

2.4. Semi-supervised Node Classification

We are interested in semi-supervised node classification tasks. To train a GCN model on such a task, we select row slices from the output matrix Y_O, corresponding to nodes with known labels in Y_I, on which a loss and its gradients are evaluated. The gradient of the loss is backpropagated through the GC layers where they get multiplied by $\widehat{A}^\top$, spreading gradients to unlabeled examples.

3. Our Proposed Architecture

We are interested in higher-order message passing, where nodes receive latent representations from their immediate (first-degree) neighbors and from further N-degree neighbors at every message passing step. In this section, we motivate and detail a model with trainable aggregation parameters that can choose how to mix latent information from neighbors at various distances.

Our analysis starts with the *Delta Operator*, a subtraction operation between node features collected from different distances. The vanilla GCN is unable to learn such a feature representation. Before introducing our model, we give one formal definition:

Definition 1 *Representing Two-hop Delta Operator: A model is capable of representing a two-hop Delta Operator if there exists a setting of its parameters and an injective mapping f, such that the output of the network becomes*

$$f\left(\sigma\left(\widehat{A}X\right) - \sigma\left(\widehat{A}^2X\right)\right), \quad (3)$$

given ***any*** *adjacency matrix $\widehat{A}$, features X, and activation function σ.*

Learning such an operator should allow models to represent feature differences among neighbors, which is necessary, for example, for learning Gabor-like filters on the graph manifold. To provide a concrete example regarding graphs, consider an online social network. In this setting, Delta Operators allow a model to represent users that live around the "boundary" of social circles (Perozzi & Akoglu, 2018). To learn an approximate feature for *American person with a popular German friend*, who might have most immediate friends speaking English, but many friends-of-friends speaking German. This person can be represented by learning a convolutional filter contrasting the English and German languages of one-hop and two-hop neighbors.

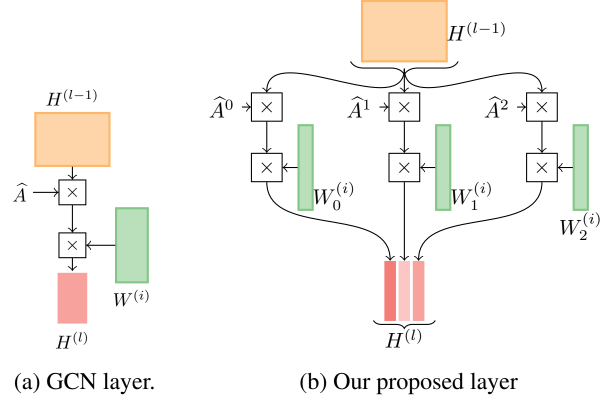
Note that in the Definition 1 we allow not learning the direct form of two-hop Delta Operators, but a transformation of it, as long as that transformation can be inverted (i.e. f is injective).

In Sections 3.1 - 3.3, we analyze the extent to which various GCN models can learn the Delta Operator. We generalize this definition and analysis in Section 3.4.

3.1. MixHop Graph Convolution Layer

We propose replacing the Graph Convolution (GC) layer defined in Equation 1, with:

$$H^{(i+1)} = \Big\Vert_{j \in P} \sigma\left(\widehat{A}^j H^{(i)} W_j^{(i)}\right), \quad (4)$$



(a) GCN layer. (b) Our proposed layer

Figure 2: Vanilla GC layer (a), using adjacency $\widehat{A}$, versus our GC layer (b), using powers of $\widehat{A}$. Orange denotes an input activation matrix, with one row per node; green denotes the trainable parameters; and red denotes the layer output. Left vs right-multiplication is specified by the relative position of the multiplicand to the $\times$ operator.

where the hyper-parameter P is a set of integer adjacency powers, $\widehat{A}^j$ denotes the adjacency matrix $\widehat{A}$ multiplied by itself j times, and $\Vert$ denotes column-wise concatenation. The difference between our proposed layer and a vanilla GCN is shown in Figure 2. Note that setting $P = \{1\}$ exactly recovers the original GC layer. Further, note that $\widehat{A}^0$ is the identity matrix I_n, where n is the number of nodes in the graph. We depict a model with $P = \{0, 1, 2\}$ in Figure 1b. In our model, each layer contains $|P|$ distinct parameter matrices, each of which can be a different size. By default, we set all $|P|$ matrices to have the same dimensionality; however, in Section 4.2, we explain how we utilize sparsifying regularizers on the learnable weight matrices to produce dataset-specific model architectures that slightly outperform our default settings.

3.2. Computational Complexity

There is no need to calculate $\widehat{A}^j$. We calculate $\widehat{A}^j H^{(i)}$ with right-to-left multiplication. Specifically, if $j = 3$, we calculate $\widehat{A}^3 H^{(i)}$ as $\widehat{A}\left(\widehat{A}\left(\widehat{A}H^{(i)}\right)\right)$. Since we store $\widehat{A}$ as a sparse matrix with m non-zero entries, an efficient implementation of our layer (Equation 4) takes $\mathcal{O}(j_{\text{max}} \times m \times s_i)$ computational time, where j_{max} is the largest element in P and s_i is the feature dimension of $H^{(i)}$. Under the realistic assumptions of $j_{\text{max}} \ll m$ and $s_l \ll m$, running an l-layer model takes $\mathcal{O}(lm)$ computational time. This matches the computational complexity of the vanilla GCN.

3.3. Representational Capability

Since each layer outputs the multiplication of different adjacency powers in different columns, the next layer's weights

Algorithm 1 MixHop Graph Convolution Layer

Inputs: $H^{(i-1)}$, $\widehat{A}$
Parameters: $\{W_j^{(i)}\}_{j \in P}$
$j_{\text{max}} := \max P$
$B := H^{(i-1)}$
for $j = 1$ **to** j_{max} **do**
  $B := \widehat{A}B$
  if $j \in P$ **then**
    $O_j := BW_j^{(i)}$
  end if
end for
$H^{(i)} := \Vert_{j \in P} O_j$
Return: $H^{(i)}$

can learn arbitrary linear combinations of the columns. By assigning a positive coefficient to a column produced by some $\widehat{A}$ power, and assigning a negative coefficient to another, the model can learn a Delta Operator. In contrast, vanilla GCNs are not capable of representing this class of operations, even when stacked over multiple layers.

Theorem 1 *The vanilla GCN defined by Equation 2 is **not** capable of representing two-hop Delta Operators.*

Theorem 2 *MixHop GCN (using layers defined in Equation 4) can represent two-hop Delta Operators.*

Proof of Theorem 1. The output of an l-layer vanilla GCN has the following form:

$$\sigma(\widehat{A}(\sigma(\widehat{A} \cdots \sigma(\widehat{A}XW^{(0)}) \cdots)W^{(l-2)})W^{(l-1)}).$$

For the simplicity of the proof, let's assume that $\forall i, s_i = n$. In a particular case, when $\sigma(x) = x$ and $X = I_n$, this reduces to $\widehat{A}^l W^*$, where $W^* = W^{(0)}W^{(1)} \cdots W^{(l-1)}$. Suppose the network is capable of representing a two-hop Delta Operator. This means that there exists an injective map f and a value for W^*, such that $\forall \widehat{A}, \widehat{A}^l W^* = f(\widehat{A} - \widehat{A}^2)$. Setting $\widehat{A} = I_n$, we get that $W^* = f(0)$. Let

$$\widehat{C}_{1,2} \triangleq \begin{bmatrix} 0.5 & 0.5 & 0 & \cdots & 0 \\ 0.5 & 0.5 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

be the symmetrically normalized adjacency matrix with self-connections corresponding to the graph having a single edge between vertices 1 and 2. Setting $\widehat{A} = \widehat{C}_{1,2}$, we get $\widehat{C}_{1,2}W^* = f(0)$. Since we already have that $f(0) = W^*$, we get that $(I_n - \widehat{C}_{1,2})W^* = 0$, which proves that the $w_1^* = w_2^*$, where w_i^* is the i-th row of W^*. Since the choice of vertices 1 and 2 was arbitrary, we have that all rows of W^* are equal to each other. Therefore, rank$(\widehat{A}^l W^*) \leq 1$, which implies that outputs of mapping f should be at most rank-one matrices. Thus, f cannot be injective, proving that vanilla GCN cannot represent two-hop Delta Operators. ■

Proof of Theorem 2. A two-layer model, defined using Equation 4 with $P = \{0, 1, 2\}$ recovers the two-hop delta operator defined in Equation 3. We start by redefining the feature vector $H^{(1)}$ learned by the first layer of the model by pulling out the element-wise activation function σ and expanding the concatenation operator found in the layer definition:

$$\begin{aligned} H^{(1)} &= \Big\Vert_{j \in \{0,1,2\}} \sigma\left(\widehat{A}^j X W_j^{(0)}\right) \\ &= \sigma\left(\Big\Vert_{j \in \{0,1,2\}} \widehat{A}^j X W_j^{(0)}\right) \\ &= \sigma\left(\left[\ I_N X W_0^{(0)} \ \vdots\ \widehat{A} X W_1^{(0)} \ \vdots\ \widehat{A}^2 X W_2^{(0)}\ \right]\right), \end{aligned}$$

We can now set $W_0^{(0)} = 0$ (zero matrix) and $W_1^{(0)} = W_2^{(0)} = I_{s_0}$. The expression above can be simplified to $H^{(1)} = \sigma\left(\left[\ 0 \ \vdots\ \widehat{A}X \ \vdots\ \widehat{A}^2 X\ \right]\right)$. The feature vector $H^{(1)}$ can be plugged into the equation for the second layer that has linear activation function:

$$H^{(2)} = \left[\ I_N H^{(1)} W_0^{(1)} \ \vdots\ \widehat{A} H^{(1)} W_1^{(1)} \ \vdots\ \widehat{A}^2 H^{(1)} W_2^{(1)}\ \right].$$

Setting the weights for the second layer as $W_1^{(1)} = W_2^{(1)} = 0$, and

$$W_0^{(1)} = \begin{bmatrix} 0 \\ I_{s_0} \\ -I_{s_0} \end{bmatrix}, \quad (5)$$

makes

$$H^{(2)} = \left[\ \left(\sigma\left(\widehat{A}X\right) - \sigma\left(\widehat{A}^2 X\right)\right) \ \vdots\ 0 \ \vdots\ 0\ \right].$$

This shows that our GCN can successfully represent the two-hop Delta Operators according to the Definition 1. ■

3.4. General Neighborhood Mixing

We generalize Definition 1 from two-hops to multiple hops:

Definition 2 *General layer-wise Neighbor-hood Mixing: A Graph Convolutional Network is capable of representing layer-wise neighborhood mixing if for any $\alpha_0, \alpha_1, \ldots, \alpha_m$ numbers, there exists a setting of its parameters and an*

injective mapping f, such that the output of the network becomes equal to

$$f\left(\sum_{j=0}^{m}\alpha_j\sigma\left(\widehat{A}^jX\right)\right) \quad (6)$$

*for **any** adjacency matrix $\widehat{A}$, features X, and activation function σ.*

Theorem 3 *GCNs defined using Equation 1 are **not** capable of representing general layer-wise neighborhood mixing.*

Theorem 4 *GCNs defined using our proposed method (Equation 4) are capable of representing general layer-wise neighborhood mixing.*

Proof of Theorem 3. This trivially follows from Theorem 1: if the vanilla GCN cannot recover a two-hop Delta Operator, defined in Equation 3, it cannot recover the Delta Operator generalization in Equation 6. ■

Proof of Theorem 4. The proof steps closely resemble the proof of Theorem 2. Our GCN with $P = \{0, \dots, m\}$ can represent the target function, by setting the first layer weight matrices as $W_j^{(0)} = I_{s_0}$, $\forall j \in P$ and setting all but the zeroth second layer weight matrices as $W_1^{(1)} = W_2^{(1)} = \dots = W_m^{(1)} = 0$. In other words, we utilize only zero-hops in the second layer, setting the zeroth-power weight matrix the following way:

$$W_0^{(1)} = \begin{bmatrix} \alpha_0 I_{s_0} \\ \vdots \\ \alpha_m I_{s_0} \end{bmatrix} \quad (7)$$

This setting of parameters exactly recover the expression in Equation 6, for any adjacency matrix $\widehat{A}$ and features X. ■

We note that the generalized Delta Operator in Definition 2 does not explicitly specify feature differences as in Definition 1; rather, the generalized form defines linear combinations of features (which includes subtraction).

4. Learning Graph Convolution Architectures

We have discussed a single layer of our model. In practice, one would stack multiple layers and interleave them with standard neural operators such as BatchNorm (Ioffe & Szegedy, 2015), element-wise activation, and Dropout (Srivastava et al., 2014). In this section, we discuss approaches to turning the MixHop GC layer into a MixHop GCN.

4.1. Output Layer

The final layer of a GCN performs a key role for learning the learned latent space of the model on the dataset that is being trained on. As MixHop uniquely mixes features from different sets of information, we theorized that constraining the output layer may result in better outcomes for different tasks. In order to leverage this property, we define our output layer in the following way: We divide s_l columns into sets of size c and compute $\widetilde{Y}_O = \sum_{k=1}^{s_l/c} q_k H^{(l)}_{*,(id_l/c\,:\,(i+1)s_l/c)}$, then $Y_O = \text{softmax}(\widetilde{Y}_O)$. Here the subscript on $H^{(l)}$ selects c contiguous columns and the scalars $q_k \in [0, 1]$ define a valid distribution (output of a softmax). This results in the model being forced to choose which features it wants to prioritize by putting more weight on that feature. We obtain the model parameters $W_i^{(j)}$ for all i, j and $q_1, \dots q_{\frac{s_l}{c}}$, by minimizing cross-entropy loss, measured only on nodes with known labels i.e. similar to (Kipf & Welling, 2017).

4.2. Learning Adjacency Power Architectures

As mentioned, our model learns multiple weight matrices $W_j^{(i)}$, one per adjacency power used in the model. By default, we set all $W_j^{(i)}$ to be the same size, which effectively assigns the same capacity to adjacency powers $\widehat{A}^j$ for all $j \in P$. We intuit that different sizes of $W_j^{(i)}$ may be more appropriate for different tasks and datasets; as such, we are interested in learning how to automatically size $W_j^{(i)}$.

For vanilla GCNs, such an architecture search is relatively inexpensive - the parameters are the number of layers and their widths. In contrast, searching over the architecture space of our model is multiplicatively $\mathcal{O}(l \times |P|)$ more expensive, as each architecture involves choices on how to divide each layer width s_i among the adjacency powers. To address this limitation, we propose using a lasso regularization to automatically learn an architecture for our model (Gordon et al., 2018). In particular, we train our architecture in stages:

1. Construct a wide network (e.g. 200 dimensions for each adjacency power, at each layer), only making choices on the depth.
2. Train the network on the task while applying L2 Group Lasso regularization over each column of each $W_j^{(l)}$. This will drop values of entire columns (close) to zero.
3. At the peak validation accuracy, measure the L2 norm of each $W_j^{(l)}$. Pick a threshold, and count the number of columns in each $W_j^{(l)}$ with norm higher than the threshold. In our experiments, we pick a threshold such that the size of the shrunken model equals size of our baseline model (i.e. with $P = \{1\}$).
4. Shrink the weight matrices by removing columns with norms below the k'th percentile.
5. Substitute L2 Group Lasso with standard L2 regularization. Restart training.

We discuss the learned architectures in Section 6.3.

5. Experimental Design

Given the model described above, a number of natural questions arise. In this section, we aim to design experiments which answer the following hypothesises:

- **H1**: The MixHop model learns delta operators.
- **H2**: Higher order graph convolutions using neighborhood mixing can outperform existing approaches (e.g. vanilla GCNs) on real semi-supervised learning tasks.
- **H3**: When learning a model architecture for MixHop the best performing architectures differ for each graph.

To answer these questions, we design three experiments.

- **Synthetic Experiments**: This experiment uses a family of synthetic graphs which allow us to vary the correlation (or *homophily*) of the edges in a generated graph, and observe how different graph convolutional approaches respond. As homophily is decreased in the network, nodes are more likely to connect to those with different labels, and a model that better captures delta operators should have superior performance.
- **Real-World Experiments**: This experiment evaluates MixHop's performance on a variety of noisy real world datasets, comparing against challenging baselines.
- **Model Visualization Experiment**: This experiment shows how an appropriately regularized MixHopmodel can learn different, task-dependent, architectures.

5.1. Datasets

We conduct semi-supervised node classification experiments on synthetic and real-world datasets.

Synthetic Datasets: Our synthetic datasets are generated following Karimi et al. (2017). We generate 10 graphs, each with a different homophily coefficient (ranging from 0.0 to 0.9 at 0.1 intervals) that indicates the likelihood of a node forming a connection to a neighbor with the same label. For example, a node in the $homophily = 0.9$ graph with 10 edges, will have on average 9 edges to a same-label neighbor. All graphs contain 5000 nodes. The features for all synthetic nodes were sampled from overlapping multi-Gaussian distributions. We randomly partition each graph into train, test, and validation node splits, all of equal size. See Appendix for more information.

Real World Datasets: The experiments with real-world datasets follow the methodology proposed in Yang et al. (2016). In addition to using the classic dataset split, (which have 20 samples per label), we evaluate against against a set of random splits with 100 samples per label. We will release our test splits.

5.2. Training

For all experiments, we construct a 2-layer network of our model using TensorFlow (Abadi et al., 2016). We train our

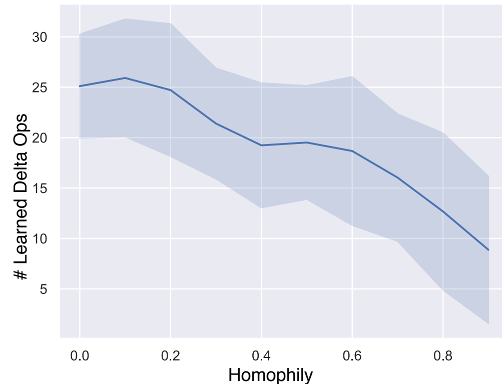


Figure 3: Amount of model capacity devoted to learning delta operators at different levels of homophily.

models using a Gradient Descent optimizer for a maximum of 2000 steps, with an initial learning rate of 0.05 that decays by 0.0005 every 40 steps. We terminate training if validation accuracy does not improve for 40 consecutive steps; as a result, most runs finish in less than 200 steps. We use 5×10^{-4} L2 regularization on the weights, and dropout input and hidden layers. We note that the citation datasets are extremely sensitve to initializations; as such, we run all models 100 times, sort by the validation accuracy, and finally report the test accuracy for the top 50 runs. For all models we ran (our models in Tables 1 & 3, and all models in Table 3), we use a latent dimension of 60; Our default architecture evenly divided 60 dimensions are divided evenly to all $|P|$ powers. Our learned architectures spread them unevenly, see Section 6.3.

6. Experimental Results

6.1. Results on Synthetic Graphs

We present our results on the synthetic datasets in Figure 4. We show average accuracy for each baseline against the homophily of the graph. We use a dense (MLP) model that does not ingest any adjacency information as a control. As expected, all models perform better as the homophily of the synthetic graph increases. At low levels of homophily, when nodes are rarely adjacent to neighbors with the same label, we observe that MixHop performs significantly better than the most competitive baseline. Interestingly, we notice that the GAT model performs significantly worse than the features-only control. This suggests that the added attention mechanism of the GAT model relies heavily on homophily in node neighborhoods.

For each level of homophily, we measured the number of delta operators learned by our model. We present these metrics in Figure 3. We observe that for low levels of homophily, our model uses 2.5X of its model capacity on

Model	Citeseer	Cora	Pubmed
ManiReg (Belkin et al., 2006)	60.1	59.5	70.7
SemiEmb (Weston et al., 2012)	59.6	59.0	71.1
LP (Zhu et al., 2003)	45.3	68.0	63.0
DeepWalk (Perozzi et al., 2014)	43.2	67.2	65.3
ICA (Lu & Getoor, 2003)	69.1	75.1	73.9
Planetoid (Yang et al., 2016)	64.7	75.7	77.2
Vanilla GCN (Kipf & Welling, 2017)	70.3	81.5	79.0
MixHop with $P = \{1\}$ (baseline)	70.7±0.73	81.1±0.84	79.9±0.78
MixHop: default architecture (ours)	**71.4**±0.81	81.8±0.62	80.0±1.1
MixHop: learned architecture (ours)	**71.4**±0.81†	**81.9**±0.40	**80.8**±0.58

Table 1: Experiments run on Node Classification citation datasets created by Yang et al. (2016). †The learned architecture for Citeseer is equivalent to default architecture, so the results are the same.

Dataset	nodes	edges	features	c	$\lvert Y_I^P \rvert$	$\lvert Y_I^R \rvert$
Citeseer	3,327	4,732	3,703	6	120	600
Cora	2,708	5,429	1,433	7	140	700
Pubmed	19,717	44,338	500	3	60	300

Table 2: Dataset statistics. Numbers of nodes (n), edges (m), features, classes (c), and labeled nodes ($|Y_I^P|$ from the Planetoid splits, $|Y_I^R|$ from our random splits).

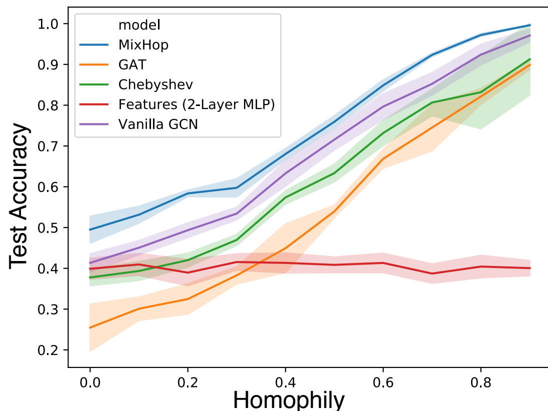


Figure 4: Synthetic dataset results. MLP does not utilize graph with (homophilic) edges, but only node features.

learning delta operators compared with higher homophily. This follows intuition: as the nodes cluster around like-labeled neighbors, the need to identify meaningful feature differences between neighbors at different distances drops significantly. These results strongly suggest that the learned delta operators play a role in the success of MixHop in Figure 4. For this experiment, we trained our model over the synthetic datasets under one constraint: input layer weights $W_0^{(j)}$ are shared across all powers $j \in P$. This allows us to examine sub-columns in the following layer $W_1^{(j)}$. Specifically, we count the number of times a feature, coming out of the first layer, is assigned values of opposite signs in $W_1^{(j)}$. We restrict the analysis to only values of $W_1^{(j)}$ with magnitude larger than the median in the corresponding column.

6.2. Node Classification Results

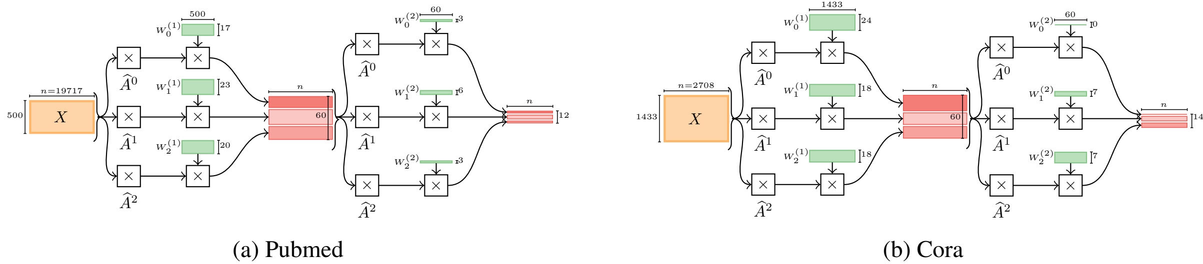
We show two sets of semi-supervised node classfication results using different splits of our datasets. Because these datasets are taken from the real world, they are inherently noisy, and it is unlikely that achieving 100% classification accuracy is possible even when given a significant amount of labeled training data. Instead, we are interested in the sparse classification task, namely how well our model is able to improve on previous work while being resilient to noise, even with limited information.

In Table 2, we demonstrate how our model performs on common splits taken from Yang et al. (2016). Accuracy numbers above double-line are copied from Kipf & Welling (2017). Numbers below the double-line are our methods, with $P = \{1\}$ being equivalent to vanilla GCNs. ± represents the standard deviation of 50 runs with different random initializations. All MixHop models are of same capacity. These splits utilize only 20 labeled nodes per class during training. We achieve a test accuracy of 71.4%, 81.9%, and 80.8% on Citeseer, Cora, and Pubmed respectively. Interestingly, for Citeseer, we see that the learned architecture was equal to the original architecture (and so the models performed the same). In Table 3, we demonstrate how our model performs using random splits with more training information available. These splits utilize 100 nodes per class during training. We achieve a test accuracy of 77.0%, 87.2%, and 83.9% on Citeseer, Cora, and Pubmed respectively.

As MixHop is able to pull in linear combinations of features from farther distances, it can extract meaningful signals in extremely sparse settings. We believe this explains why MixHop outperforms baseline methods in both sets of dataset

Model	**Citeseer**	**Cora**	**Pubmed**
2-Layer MLP	70.6±1	69.0±1.1	78.3±0.54
Chebyshev (Defferrard et al., 2016)	74.2±0.5	85.5±0.4	81.8±0.5
Vanilla GCN (Kipf & Welling, 2017)	76.7±0.43	86.1±0.34	82.2±0.29
GAT (Velickovic et al., 2018)	74.8±0.42	83.0±1.1	81.8±0.18
MixHop: default architecture (ours)	76.3±0.41	87.0±0.51	83.6±0.68
MixHop: learned architecture (ours)	**77.0**±0.54	**87.2**±0.32	**83.8**±0.44

Table 3: Classification results on random partitions of (Yang et al., 2016) datasets.



(a) Pubmed (b) Cora

Figure 5: Learned MixHop Architectures. Note how different parameter sizes (green boxes) are learned for the two datasets. For example, Group-Lasso regularization on Cora removes all capacity for the zeroth power in the second GC layer. For space, all matrices are plotted transposed and output layer (Section 4.1) has been ommitted.

splits. The results of these experiments confirm our hypothesis (H2) that higher order graph convolution methods with neighborhood mixing can outperform existing methods on real datasets.

6.3. Visualizing Learned Architectures

Figure 5 depicts the learned architectures for two of the citation datasets. We note that each dataset prefers its own architecture. For example, Cora prefers to have zero-capacity on the 0th power of the adjacency matrix (effectively ignoring the features of each node) in the second layer. Not shown (for space reasons) is Citeseer, which prefers the default parameter settings with the same weight capacity across all powers. All three real datasets had different final architectures, which confirms our hypothesis (H3) that different architectures are optimal for different graph datasets.

7. Related Work

Abu-El-Haija et al. (2018b) uses adjacency powers but for embedding learning. Abu-El-Haija et al. (2018a); Atwood & Towsley (2016) use adjacency powers for feature propagation on graphs, but they combine the powers at the end of the network (right before classification), and Lee et al. (2018) combine them at the input. We intermix information from the powers layer-wise, enabling our method to learn neighborhood mixing e.g. delta operators, which contrast the features of immediate neighbors from those further away. Defferrard et al. (2016) uses more Chebyshev polynomials (i.e. higher-rank) Graph Convolution, but their model underperforms our baseline (Kipf & Welling, 2017), allowing us to hypothesize that message passing along edges outperforms explicit alignment onto the graph Fourier Basis.

8. Conclusion

In this work, we analyzed the expressive power of popular methods for semi-supervised learning with Graph Neural Networks and we showed they cannot learn general neighborhood mixing functions. To address this, we have proposed a graph convolutional layer that utilizes multiple powers of the adjacency matrix. Repeated application of this layer allows a model to learn general mixing of neighborhood information, including averaging and delta operators in the feature space, without additional memory or computational complexity. Utilizing L2 group lasso regularization on these stacked layers allows us to learn a unique architecture that is optimized for each dataset. Our experimental results showed that higher order graph convolution methods can achieve state of the art performance on several node classification tasks. Our analysis of the experimental results showed that neighborhood difference operators are especially useful in graphs which do not have high homophily (correlation between edges and labels). While we focused this paper on applying our proposal to the most popular models for graph convolution, it is possible to implement our method in more sophisticated frameworks including the recent GAT (Velickovic et al., 2018). Other recent work like (Ying et al., 2018), which focuses on hierarchical pooling for community-aware graph representation might also be extended to use general neighborhood mixing layers.

Acknowledgements

The authors acknowledge support from the Defense Advanced Research Projects Agency (DARPA) under award FA8750-17-C-0106, and acknowledge discussions with Jesse Dodge and Leto Peel, respectively, on Group Lasso regularization and synthetic experiments.

References

Abadi, M., Agarwal, A., et al. TensorFlow: Large-scale machine learning on heterogeneous systems. In *OSDI'16*, 2016.

Abu-El-Haija, S., Kapoor, A., Perozzi, B., and Lee, J. N-gcn: Multi-scale graph convolution for semi-supervised node classification. In *MLG KDD Workshop*, 2018a.

Abu-El-Haija, S., Perozzi, B., Al-Rfou, R., and Alemi, A. A. Watch your step: Learning node embeddings via graph attention. In *NeurIPS*, 2018b.

Atwood, J. and Towsley, D. Diffusion-convolutional neural networks. In *NeurIPS*, 2016.

Belkin, M., Niyogi, P., and Sindhwani, V. Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. In *JMLR*, 2006.

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

Daugman, J. Two-dimensional spectral analysis of cortical receptive field profiles. In *Vision Research*, 1980.

Daugman, J. Uncertainty relation for resolution in space, spatial frequency, and orientation optimized by two-dimensional visual cortical filters. In *Journal of the Optical Society of America*, 1985.

Defferrard, M., Bresson, X., and Vandergheynst, P. Convolutional neural networks on graphs with fast localized spectral filtering. In *NeurIPS*, 2016.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *ICML*, 2017.

Goller, C. and Kuchler, A. Learning task-dependent distributed representations by backpropagation through structure. In *International Conference on Neural Networks*, 1996.

Gordon, A., Eban, E., Nachum, O., Chen, B., Wu, H., Yang, T.-J., and Choi, E. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *CVPR*, June 2018.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *arXiv:1502.03167*, 2015.

Karimi, F., Genois, M., Wagner, C., Singer, P., and Strohmaier, M. Visibility of minorities in social networks. In *arxiv/1702.00150*, 2017.

Kipf, T. and Welling, M. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *NeurIPS*, 2012.

Lee, H., Grosse, R., Ranganath, R., and Ng, A. Y. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *ICML*, 2009.

Lee, J. B., Rossi, R. A., Kong, X., Kim, S., Koh, E., and Rao, A. Higher-order graph convolutional networks. In *arxiv/1809.07697*, 2018.

Lu, Q. and Getoor, L. Link-based classification. In *ICML*, 2003.

Perozzi, B. and Akoglu, L. Discovering communities and anomalies in attributed graphs: Interactive visual exploration and summarization. *KDD*, 12(2):24:1–24:40, January 2018.

Perozzi, B., Al-Rfou, R., and Skiena, S. Deepwalk: Online learning of social representations. In *Knowledge Discovery and Data Mining*, 2014.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. In *JMLR*, 2014.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *CVPR*, 2015.

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

Weston, J., Ratle, F., Mobahi, H., and Collobert, R. Deeplearning via semi-supervised embedding. In *Neural Networks: Tricks of the Trade*, pp. 639–655, 2012.

Yang, Z., Cohen, W., and Salakhutdinov, R. Revisiting semi-supervised learning with graph embeddings. In *ICML*, 2016.

Ying, Z., You, J., Morris, C., Ren, X., Hamilton, W., and Leskovec, J. Hierarchical graph representation learning with differentiable pooling. In *NeurIPS*, 2018.

Zhu, X., Ghahramani, Z., and Lafferty, J. Semi-supervised learning using gaussian fields and harmonic functions. In *ICML*, 2003.