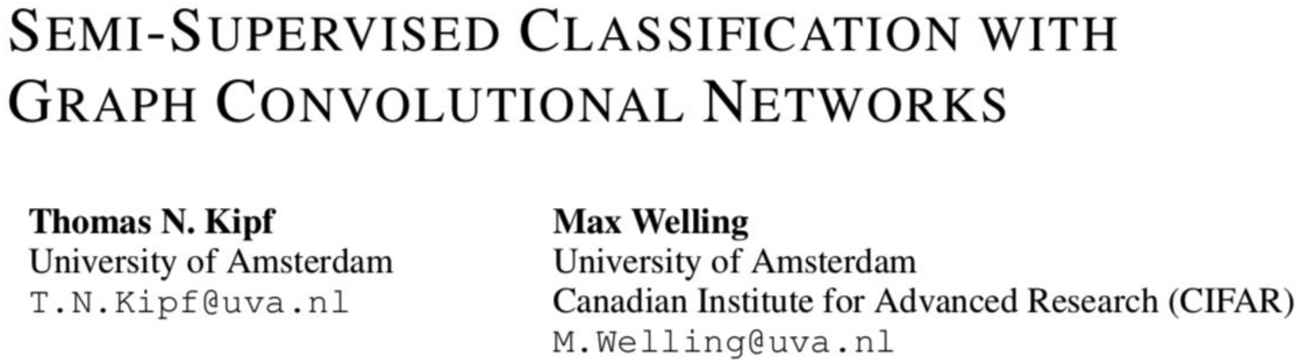


Semi-Supervised Classification with Graph Convolutional Networks

Thomas N. Kipf
University of Amsterdam
T.N.Kipf@uva.nl

Max Welling
University of Amsterdam
Canadian Institute for Advanced Research (CIFAR)
M.Welling@uva.nl

Main Idea: Generalizes the convolution operation to graph-structured data. These convolutions can be stacked with intermediate activation functions to form a model for high-performance, semi-supervised classification on graphs.

Helpful Background: Shift Operators

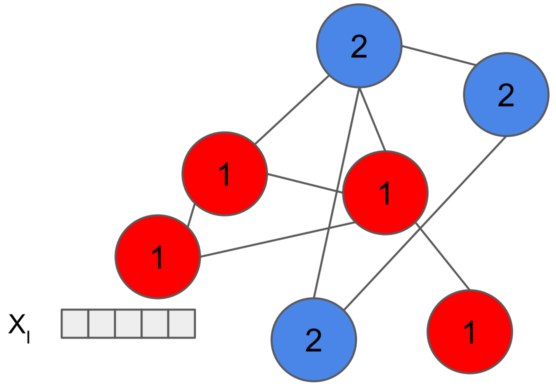
- **Shift operators:** matrix representation of the graph
  - Multiplying a signal x by a shift operator diffuses x through the graph
  - Aggregates neighboring information into each node
- **Examples:**
  - *Adjacency Matrix:* matrix to represent graph connectivity
    - If graph is undirected, this is symmetric
  - *Laplacian Matrix:* D - A (only defined for undirected graphs)
    - D = diagonal degree matrix
    - A = adjacency matrix
  - *Normalized Adjacency/Laplacian Matrix:*
    - $L' = D^{-1/2} L D^{-1/2}$
    - $A' = D^{-1/2} A D^{-1/2}$

Helpful Background: Graph Fourier Transform

- If the shift operator is symmetric, we can derive it's eigenvalue decomposition
  - Shift operator is 'normal' (i.e., we can diagonalize it with a unitary matrix)
  - $S = V\Lambda V^H$ ('H' represents the conjugate transpose)
- **Graph Fourier Transform:** $x'' = V^H x$
  - *Inverse Graph Fourier Transform:* $x = Vx''$
  - Assume we have a filter H that we apply to our graph
    - H is a polynomial of the shift operator S: $H = Vh(\Lambda)V^H$
    - The eigenvectors of the filter and the shift operator are the same!
  - What happens when we multiply our signal by this filter?
    - $Hx = Vh(\Lambda)V^H x = Vh(\Lambda)x'' = V[h(\Lambda)x'']$
    - Filtering is pointwise in the fourier domain
    - <u>The above operation applies the fourier transform, applies a pointwise multiplication, then applies the inverse fourier transform to get the output.</u>

Problem Formulation

- **Graph Node Classification**
- Labels are known for a portion of the nodes in the graph
    - Semi-supervised classification
- Each node has an input feature
    - Feature vectors x_i can be combined into a matrix X
- **Goal:** use input and graph structure information to correctly classify nodes



What did we do previously?

- Still assume only a portion of nodes are labeled
- Use explicit regularization to smooth label information across the graph

$$\mathcal{L} = \mathcal{L}_0 + \lambda \mathcal{L}_{\text{reg}}\,, \quad \text{with} \quad \mathcal{L}_{\text{reg}} = \sum_{i,j} A_{ij} \| f(X_i) - f(X_j) \|^2 = f(X)^\top \Delta f(X)\,.$$

- Objective includes both supervised loss (L_0) and regularization (L_{reg})
    - Requires a regularization weighting to be specified
    - Regularization is based on the graph laplacian ($\Delta = D - A$
    - **Assumes connected nodes are likely to share the same label**
- Some previous work also proposed neural networks for graphs
    - Most of these approaches used RNNs
    - <u>Common Issues</u>: (1) require different weights for nodes with different degrees (2) computational complexity too high (i.e., not scalable to large graphs) (3) require an ordering of nodes to be imposed

Creating a better model...

- The following propagation rule is used: $H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$
    - Can be efficiently implemented with sparse-dense matrix multiplication
    - How do we arrive at this rule?
- Consider a spectral convolution as follows: $g_\theta \star x = U g_\theta U^\top x$
    - U contains eigenvectors of the normalized laplacian
        - Filter is a polynomial of shift operator - shares eigenvectors with the laplacian!
    - We replace the Laplacian diagonal eigenvalue matrix with g_θ
        - $Ug_\theta U^T$ is simply the diagonalization of the filter matrix
        - g_θ is a function of the eigenvalue matrix of the laplacian
- We can’t implement graph convolutions like this in practice - too expensive!
    - *Why?* It requires computing eigenvalues/vectors of the laplacian

Creating a better model...

Chebyshev Polynomials

$T_k(X) = 2xT_{k-1}(X) - T_{k-2}(X)$

$T_0 = 1, T_1 = x$

- **Solution:** represent g_θ with a truncated Chebyshev polynomial expansion!
  - Approximation for g_θ: $g_{\theta'}(\Lambda) \approx \sum_{k=0}^{K} \theta'_k T_k(\tilde{\Lambda})$
  - This can be substituted into $g_\theta \star x = U g_\theta U^\top x$ to yield $\boxed{g_{\theta'} \star x \approx \sum_{k=0}^{K} \theta'_k T_k(\tilde{L}) x}$
    - This follows from the fact that $(U\Lambda U^T)^k = U\Lambda^k U^T$
  - *Can be evaluated in O(|ε|) time!*
- Convolution is now a Kth order polynomial (only depends on k-hop nodes)
  - Assume we take k=1 (i.e., linear w.r.t. laplacian)
    - Linear layers can be stacked and separated with nonlinearities
  - With a few assumptions, our convolution becomes $g_\theta \star x \approx \theta\left(I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}}\right) x$
    - This is problematic because the eigenvalues are [0, 2]
    - To avoid this, a renormalization trick is used $I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \rightarrow \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$
- **After the above steps, the GCN layer-wise update rule is derived:**

$$\boxed{Z = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} X \Theta}$$

Graph Convolutional Network (GCN)

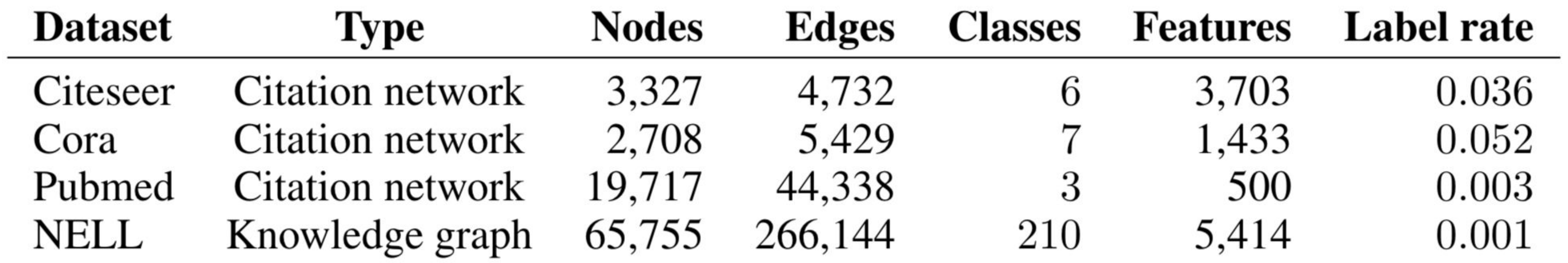
- Transform node representations, aggregate neighbors, activation function, repeat…
- **Pros:**
  - No longer biases connected nodes to be classified the same!
  - Handles all possible node degrees with only one weight matrix per layer
  - Can perform semi-supervised classification by just evaluating loss on labeled nodes
  - Efficient (adjacency matrix can be pre-computed)

$$\boxed{Z = f(X, A) = \mathrm{softmax}\left(\hat{A}\ \mathrm{ReLU}\left(\hat{A}XW^{(0)}\right)W^{(1)}\right)}$$

Example of a 2-Layer GCN

Experiments

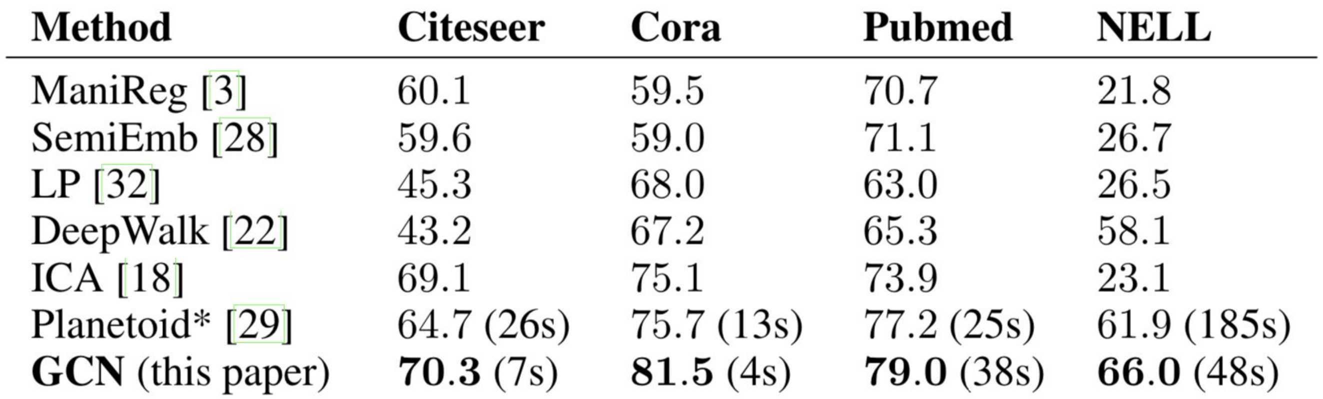
- Evaluate the GCN on many well-known semi-supervised node classification tasks for graphs
- Only consider transductive classification (i.e., no unseen graphs)
- Use 2-layer GCN for all experiments
    - The appendix contains experiments with deeper networks



Dataset	Type	Nodes	Edges	Classes	Features	Label rate
Citeseer	Citation network	3,327	4,732	6	3,703	0.036
Cora	Citation network	2,708	5,429	7	1,433	0.052
Pubmed	Citation network	19,717	44,338	3	500	0.003
NELL	Knowledge graph	65,755	266,144	210	5,414	0.001

Performance

- Train models with batch gradient descent
- GCN sets new SOTA on all tasks
- They compare GCN to numerous other propagation methods
- GCN computation time (i.e., per-epoch) is shown to scale linearly with the size of the graph



Method	**Citeseer**	**Cora**	**Pubmed**	**NELL**
ManiReg [3]	60.1	59.5	70.7	21.8
SemiEmb [28]	59.6	59.0	71.1	26.7
LP [32]	45.3	68.0	63.0	26.5
DeepWalk [22]	43.2	67.2	65.3	58.1
ICA [18]	69.1	75.1	73.9	23.1
Planetoid* [29]	64.7 (26s)	75.7 (13s)	77.2 (25s)	61.9 (185s)
GCN (this paper)	**70.3** (7s)	**81.5** (4s)	**79.0** (38s)	**66.0** (48s)

Remaining Issues

- Directed/weighted edges
    - Current GCN model assumes and unweighted symmetric adjacency matrix
- Memory/compute requirements
    - Must fit entire graph into GPU to perform an update!
    - If we use a subgraph, the receptive field expands exponentially
- Generalization to unseen graphs
- Self connection is weighted the same as neighbor connections
    - Should we make the current node representation more important? $\tilde{A} = A + \lambda I_N$