Graph Convolutional Networks (GCNs)

Dimitris Papatheodorou

Aalto University

dimitrispapatheodorou95@gmail.com

May 21, 2019

Overview

1. Introduction
   - Problem Setting
   - Graph Laplacian
2. Graph Convolutional Networks
   - The ideas behind the problem
   - GCN idea and convolutions on graphs
   - Spectral Graph Convolutions (SGC)
   - Implementation and results

Problem Setting

- **Graphs** are structured representations of data, such as citation networks, social networks, the World-Wide-Web, protein-interaction networks, and others.

Problem Setting

- **Graphs** are structured representations of data, such as citation networks, social networks, the World-Wide-Web, protein-interaction networks, and others.
- Recent work on generalizing neural networks to graphs in various ways for different tasks (graph classification, nodes classification, clustering, link prediction, node embeddings, and more).

Problem Setting

- **Graphs** are structured representations of data, such as citation networks, social networks, the World-Wide-Web, protein-interaction networks, and others.
- Recent work on generalizing neural networks to graphs in various ways for different tasks (graph classification, nodes classification, clustering, link prediction, node embeddings, and more).
- We will denote a undirected graph as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with:
  - nodes $\upsilon_i \in \mathcal{V}$ (N nodes)
  - edges $\epsilon_{ij} = (\upsilon_i, \upsilon_j) \in \mathcal{E}$ (M edges)
  - adjacency matrix $\mathcal{A} \in \mathbb{R}^{N \times N}$ (binary or weighted)
  - degree matrix $\mathcal{D}_{ii} = \sum_j \mathcal{A}_{ij}$
  - unnormalized graph Laplacian $\Delta = \mathcal{D} - \mathcal{A}$
  - normalized graph Laplacian $\mathcal{L} = \mathrm{I}_N - \mathcal{D}^{-\frac{1}{2}} \mathcal{A} \mathcal{D}^{-\frac{1}{2}}$

Intuition of graph Laplacian

Intuition of graph Laplacian

- The graph Laplacian can be considered as the discrete analogue (applied on graphs) of the Laplacian operator ∇^2 on graphs, which is differential operator given by the divergence of the gradient of a function f on Euclidean space.
  $\rightarrow \Delta f = \nabla^2 f = \text{div}(\text{grad}(f))$

Intuition of graph Laplacian

- The graph Laplacian can be considered as the discrete analogue (applied on graphs) of the Laplacian operator ∇^2 on graphs, which is differential operator given by the divergence of the gradient of a function f on Euclidean space.
  $\rightarrow \Delta f = \nabla^2 f = \mathrm{div}(\mathrm{grad}(f))$

The Gradient Operator

- For a function on the Euclidean space, the **gradient operator** gives the derivative of the function along each direction at every point.
- For a function on a discrete "graph space", the **graph gradient operator** gives the difference of the function along each edge at every vertex:
  $\rightarrow$ For edge $\epsilon = (u, v)$: $\mathrm{grad}\ (f)|_{\epsilon} = f(u) - f(v)$.
  $\Rightarrow \mathrm{grad}(f) = K^{\top} f$, where K is the incidence matrix of size $M \times N$.
  (by assigning an arbitrary orientation on the edges)

Intuition of graph Laplacian

$f_1 = 1$, $f_2 = 3$, $f_3 = 1$, $f_4 = 2$, $f_5 = 1$

e_1, e_2, e_3, e_4

$$f = \begin{bmatrix} 1 \\ 3 \\ 1 \\ 2 \\ 1 \end{bmatrix} \begin{matrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{matrix} \quad 5 \times 1 \qquad K = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & -1 \end{bmatrix} \quad 5 \times 4$$

$e_1 \quad e_2 \quad e_3 \quad e_4$

$$\text{grad}(f) = K^T f = \begin{bmatrix} -2 \\ 2 \\ 1 \\ 1 \end{bmatrix} \begin{matrix} e_1 : f_1 - f_2 \\ e_2 : f_2 - f_3 \\ e_3 : f_2 - f_4 \\ e_4 : f_4 - f_5 \end{matrix} \quad 4 \times 1$$

Intuition of graph Laplacian

The Divergence Operator

- In the Euclidean space, divergence at a point gives the net outward flux of a vector field.
- For graphs, the vector field is just the gradient of a graph function.
- In the discrete "graph space", we define the graph divergence of a function g over the edges of a graph (eg the graph gradient) as a mapping from g to Kg.
  $\rightarrow \nabla f = \text{div}(\text{grad}(f)) = KK^\top f$, where $KK^\top$ is the Laplacian.

Intuition of graph Laplacian

The Divergence Operator

- In the Euclidean space, divergence at a point gives the net outward flux of a vector field.
- For graphs, the vector field is just the gradient of a graph function.
- In the discrete "graph space", we define the graph divergence of a function g over the edges of a graph (eg the graph gradient) as a mapping from g to Kg.
  $\rightarrow \nabla f = \text{div}(\text{grad}(f)) = KK^\top f$, where $KK^\top$ is the Laplacian.
- Notice that the laplacian $\Delta = KK^\top$ here is Cholesky decomposed, thus it's *positive semi-definite*.

Intuition of graph Laplacian

$$g = \begin{bmatrix} -2 \\ 2 \\ 1 \\ 1 \end{bmatrix} \begin{matrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{matrix} \quad 4\times 1$$

$$\mathrm{div}(g) = Kg = \begin{bmatrix} -2 \\ 5 \\ -2 \\ 0 \\ 1 \end{bmatrix} \begin{matrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{matrix} \quad 5\times 1$$

Intuition of graph Laplacian

$$
KK^T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & -1 \end{bmatrix}_{5\times 4} \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix}_{4\times 5} = \begin{bmatrix} \textcircled{1} & -1 & 0 & 0 & 0 \\ -1 & \textcircled{3} & -1 & -1 & 0 \\ 0 & -1 & \textcircled{1} & 0 & 0 \\ 0 & -1 & 0 & \textcircled{2} & -1 \\ 0 & 0 & 0 & -1 & \textcircled{1} \end{bmatrix}_{5\times 5} = L
$$

- Circled items: degrees of the vertices!
- Now the definition is more clear: $\Delta = \mathcal{D} - \mathcal{A}$

Intuition of graph Laplacian

Another example

Labeled graph	Degree matrix	Adjacency matrix	Laplacian matrix
	$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 2 & -1 & 0 & 0 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & -1 & 0 & -1 & 3 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$

Intuition of graph Laplacian

More intuition

- For continuous spaces, the Laplacian is the secord derivative, so it measures how *smooth* is a function over its domain.
- It's the same for graph laplacians: the function values *don't change by much* from one node to an adjacent one.

Intuition of graph Laplacian

More intuition

- For continuous spaces, the Laplacian is the secord derivative, so it measures how *smooth* is a function over its domain.
- It's the same for graph laplacians: the function values *don't change by much* from one node to an adjacent one.
- Formally (general case of weighted graphs):

$$E(f) = \frac{1}{2} \sum_{u \sim v} w_{uv} (f(u) - f(v))^2 = \left\| K^\top f \right\|^2 = f^\top \Delta f$$

Intuition of graph Laplacian

More intuition

- For continuous spaces, the Laplacian is the secord derivative, so it measures how *smooth* is a function over its domain.
- It's the same for graph laplacians: the function values *don't change by much* from one node to an adjacent one.
- Formally (general case of weighted graphs):

$$E(f) = \frac{1}{2} \sum_{u \sim v} w_{uv}(f(u) - f(v))^2 = \left\| K^\top f \right\|^2 = f^\top \Delta f$$

- Equivalent to Dirichlet energy, for open set $\Omega \subseteq \mathbb{R}^n$ and function $f : \Omega \to \mathbb{R}$:

$$E(f) = \frac{1}{2} \int_\Omega \|\nabla f(x)\|^2 dx$$

a measure of how *variable* a function is.

Intuition of graph Laplacian

- So, minimizing the variation of a graph function leads us to the Laplacian.
- The functions that minimize $f^\top \Delta f$ are the eigenvectors of Δ.
- This can be shown either directly, or via the *Courant-Fischer-Weyl min-max principle / variational theorem* on the *Rayleigh quotient* of the laplacian for unit norm functions.
  (See more in Algorithmic Methods of Data Mining course slides)

Intuition of graph Laplacian

Interesting Properties

- $\Delta = KK^\top$, thus the Laplacian is a Gram Matrix.

Intuition of graph Laplacian

Interesting Properties

- $\Delta = KK^\top$, thus the Laplacian is a Gram Matrix.
- The multiplicity of its zero eigenvalue λ_0 is equal to the number of components of the graph. (multiplicity: remember the characteristic polynomial $det(A - \lambda I) = 0$).

Intuition of graph Laplacian

Interesting Properties

- $\Delta = KK^\top$, thus the Laplacian is a Gram Matrix.
- The multiplicity of its zero eigenvalue λ_0 is equal to the number of components of the graph. (multiplicity: remember the characteristic polynomial $det(A - \lambda I) = 0$).
- The second smallest eigenvalue (aka Fiedler value) of the Laplacian matrix will be zero if and only if the graph is *disconnected*.
- The smaller the second smallest eigenvalue, the less 'connected' the graph.
- Interlacing property: For a graph with Laplacian Δ and eigenvalues of Δ: $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n$, if we delete an *edge*, the new eigenvalues are: $\mu_1 \geq \mu_2 \geq \cdots \geq \mu_{n-1}$. It holds that:

$$2 \geq \lambda_1 \geq \mu_1 \geq \lambda_2 \geq \mu_2 \geq \cdots \geq \mu_{n-1} \geq \lambda_n \geq 0$$

This is the same for the adjacency matrix and nodes!

The ideas behind the problem

- Kipf's and Welling's paper [Kipf and Welling(2016)] focuses on nodes classification, where node labels are available for a small number of nodes.

The ideas behind the problem

- Kipf's and Welling's paper [Kipf and Welling(2016)] focuses on nodes classification, where node labels are available for a small number of nodes.
- That's a graph-based semi-supervised learning problem.

The ideas behind the problem

- Kipf's and Welling's paper [Kipf and Welling(2016)] focuses on nodes classification, where node labels are available for a small number of nodes.
- That's a graph-based semi-supervised learning problem.
- It can be tackled by smoothing label information over the graph via some form of explicit graph-based regularization:

$$\mathcal{J} = \mathcal{J}_0 + \lambda \mathcal{J}_{\text{reg}}$$

$$\mathcal{L}_{\text{reg}} = \sum_{i,j} \mathcal{A}_{ij} \left\| f\left(X_i\right) - f\left(X_j\right) \right\|^2 = f(X)^\top \Delta f(X)$$

◦ $\mathcal{J}_{reg}$ is the graph laplacian regularization term
◦ $\mathcal{J}_0$ is the supervised loss wrt the labeled parts of the graph
◦ $f(\cdot)$ is a diffentiable function (eg a neural network)
◦ $X = \{X_i | i = 1, \ldots, N\}$ is a matrix of node feature vectors

Are they good enough?

It that any good?

Are they good enough?

It that any good?

- Yes, but it assumes that connected nodes are likely to share the same label.

Are they good enough?

It that any good?

- Yes, but it assumes that connected nodes are likely to share the same label.
- Edges do not necessarily encode node similarity! They may contain additional information.

Are they good enough?

It that any good?

- Yes, but it assumes that connected nodes are likely to share the same label.
- Edges do not necessarily encode node similarity! They may contain additional information.
- This assumption restrict the modeling capacity of our classifier.

GCNs main idea

A better idea?

- GCNs encode the graph structure directly using a neural network $f(X, \mathcal{A})$

GCNs main idea

A better idea?

- GCNs encode the graph structure directly using a neural network $f(X, \mathcal{A})$
- using the unregularized supervised loss $\mathcal{J}_0$

GCNs main idea

A better idea?

- GCNs encode the graph structure directly using a neural network $f(X, \mathcal{A})$
- using the unregularized supervised loss $\mathcal{J}_0$
- and by conditioning f on $\mathcal{A}$ they distribute gradient information from $\mathcal{J}_0$ and will enable it to learn representations of nodes both with and without labels.

Convolutions on graphs

- For a multi-layer $f(X, \mathcal{A})$ GCN, a simple propagation rule could be:

$$H^{(0)} = X$$

$$H^{(\ell+1)} = f\left(H^{(\ell)}, \mathcal{A}\right) = \alpha\left(\mathcal{A}H^{(\ell)}W^{(\ell)}\right)$$

where α is an activation function and W^{ℓ} the trainable weight matrix of the ℓ-th layer.

Convolutions on graphs

- For a multi-layer $f(X, \mathcal{A})$ GCN, a simple propagation rule could be:

$$H^{(0)} = X$$

$$H^{(\ell+1)} = f\left(H^{(\ell)}, \mathcal{A}\right) = \alpha\left(\mathcal{A}H^{(\ell)}W^{(\ell)}\right)$$

  where α is an activation function and W^{ℓ} the trainable weight matrix of the ℓ-th layer.

- For better results: ◦ Enforce self-loops: $\tilde{\mathcal{A}} = \mathcal{A} + I_N$
  ◦ Symmetrical normalization: $\tilde{D}^{-\frac{1}{2}}\tilde{\mathcal{A}}\tilde{D}^{-\frac{1}{2}}$

$$H^{(\ell+1)} = f\left(H^{(\ell)}, \mathcal{A}\right) = \alpha\left(\tilde{D}^{-\frac{1}{2}}\tilde{\mathcal{A}}\tilde{D}^{-\frac{1}{2}}H^{(\ell)}W^{(\ell)}\right)$$

Spectral Graph Convolutions (SGC)

$\rightarrow$ To understand this idea we take a look at Spectral Graph Convolutions.

Spectral Graph Convolutions (SGC)

$\rightarrow$ To understand this idea we take a look at Spectral Graph Convolutions.
What we need:

- Signal $x \in \mathbb{R}^N$ (scalar for every node)

Spectral Graph Convolutions (SGC)

$\rightarrow$ To understand this idea we take a look at Spectral Graph Convolutions.
What we need:

- Signal $x \in \mathbb{R}^N$ (scalar for every node)
- Filter $g_\theta = \text{diag}(\theta), \ \theta \in \mathbb{R}^N$

Spectral Graph Convolutions (SGC)

$\rightarrow$ To understand this idea we take a look at Spectral Graph Convolutions. What we need:

- Signal $x \in \mathbb{R}^N$ (scalar for every node)
- Filter $g_\theta = \text{diag}(\theta),\ \theta \in \mathbb{R}^N$
- Eigendec normalized laplacian: $\mathcal{L} = \mathrm{I}_N - \mathcal{D}^{-\frac{1}{2}} \mathcal{A} \mathcal{D}^{-\frac{1}{2}} = U \Lambda U^\top$

Spectral Graph Convolutions (SGC)

$\rightarrow$ To understand this idea we take a look at Spectral Graph Convolutions.
What we need:

- Signal $x \in \mathbb{R}^N$ (scalar for every node)
- Filter $g_\theta = \text{diag}(\theta),\ \theta \in \mathbb{R}^N$
- Eigendec normalized laplacian: $\mathcal{L} = \mathrm{I}_N - \mathcal{D}^{-\frac{1}{2}} \mathcal{A} \mathcal{D}^{-\frac{1}{2}} = U \Lambda U^\top$
- SGC multiplication operator $\star$ in the Fourier (frequency) domain

Spectral Graph Convolutions (SGC)

$\rightarrow$ To understand this idea we take a look at Spectral Graph Convolutions.
What we need:

- Signal $x \in \mathbb{R}^N$ (scalar for every node)
- Filter $g_\theta = \text{diag}(\theta),\ \theta \in \mathbb{R}^N$
- Eigendec normalized laplacian: $\mathcal{L} = \mathrm{I}_N - \mathcal{D}^{-\frac{1}{2}} \mathcal{A} \mathcal{D}^{-\frac{1}{2}} = U \Lambda U^\top$
- SGC multiplication operator $\star$ in the Fourier (frequency) domain
- $U^\top x$ graph Fourier transform of the Signal
- We can understand g_θ as function of Λ: $g_\theta(\Lambda)$

What we get:

$$g_\theta \star x = U g_\theta U^\top x$$

Spectral Graph Convolutions (SGC)

$\rightarrow$ To understand this idea we take a look at Spectral Graph Convolutions.
What we need:

- Signal $x \in \mathbb{R}^N$ (scalar for every node)
- Filter $g_\theta = \text{diag}(\theta), \ \theta \in \mathbb{R}^N$
- Eigendec normalized laplacian: $\mathcal{L} = \mathrm{I}_N - \mathcal{D}^{-\frac{1}{2}} \mathcal{A} \mathcal{D}^{-\frac{1}{2}} = U \Lambda U^\top$
- SGC multiplication operator $\star$ in the Fourier (frequency) domain
- $U^\top x$ graph Fourier transform of the Signal
- We can understand g_θ as function of Λ: $g_\theta(\Lambda)$

What we get:

$$g_\theta \star x = U g_\theta U^\top x$$

$\rightarrow$ This procedure is computationally expensinve though.
Eigendecomposition is expensive and multiplication with U is $\mathcal{O}(N^2)$.

Spectral Graph Convolutions (SGC)

Solution by
[Hammond et al.(2011)Hammond, Vandergheynst, and Gribonval]:
Approximate $g_\theta(\Lambda)$ using Chebyshev polynomials $T_k(x)$ of Kth order.

Spectral Graph Convolutions (SGC)

Solution by
[Hammond et al.(2011)Hammond, Vandergheynst, and Gribonval]:
Approximate $g_\theta(\Lambda)$ using Chebyshev polynomials $T_k(x)$ of Kth order.

Chebyshev Polynomials Review:

- Recurrence Formula:

$$
\begin{aligned}
T_0(x) &= 1 \\
T_1(x) &= x \\
T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x)
\end{aligned}
$$

- Using Rodrigues' formula:

$$T_n(x) = \frac{(-2)^n n!}{(2n)!}\sqrt{1-x^2}\frac{d^n}{dx^n}\left(1-x^2\right)^{n-1/2}$$

Spectral Graph Convolutions (SGC)

So we get this approximation:

$$g_{\theta'}(\Lambda) \approx \sum_{k=0}^{K} \theta'_k T_k(\tilde{\Lambda}),$$

where ∘ Λ is rescaled: $\tilde{\Lambda} = \frac{2}{\lambda_{\max}} \Lambda - I_N$,
∘ λ_{max} is the largest eigenvalue of $\mathcal{L}$,
∘ $\theta' \in \mathbb{R}^K$ Chebyshev coefficients

Thus for the convolution we get:

$$g_{\theta'} \star x \approx \sum_{k=0}^{K} \theta'_k T_k(\tilde{L}) x, \quad \text{which is } \mathcal{O}(|\mathcal{E}|)$$

with $\tilde{L} = \frac{2}{\lambda_{\max}} L - I_N$
$\rightarrow$ K-localized expression, thus depends only on nodes that are at maximum K steps away from the central node.

Stacked SGCs $\rightarrow$ Profit?

- Simply stacking SGCs gives as a neural network.

Stacked SGCs $\rightarrow$ Profit?

- Simply stacking SGCs gives as a neural network.
- What if we only use $K = 1$? We get a linear function wrt to $\mathcal{L}$ and on the graph laplacian spectrum. Is that any good? :S

Stacked SGCs $\rightarrow$ Profit?

- Simply stacking SGCs gives as a neural network.
- What if we only use $K = 1$? We get a linear function wrt to $\mathcal{L}$ and on the graph laplacian spectrum. Is that any good? :S
- Yes! Stacking simple functions still lets us explore a rich class of convolutional filters and comes with some extra benefits:

Stacked SGCs $\rightarrow$ Profit?

- Simply stacking SGCs gives as a neural network.
- What if we only use $K = 1$? We get a linear function wrt to $\mathcal{L}$ and on the graph laplacian spectrum. Is that any good? :S
- Yes! Stacking simple functions still lets us explore a rich class of convolutional filters and comes with some extra benefits:
  - Such filters are also not limited by the explicit form and parameterization of the approximation,

Stacked SGCs $\rightarrow$ Profit?

- Simply stacking SGCs gives as a neural network.
- What if we only use $K = 1$? We get a linear function wrt to $\mathcal{L}$ and on the graph laplacian spectrum. Is that any good? :S
- Yes! Stacking simple functions still lets us explore a rich class of convolutional filters and comes with some extra benefits:
  - Such filters are also not limited by the explicit form and parameterization of the approximation,
  - less likely to overfit on local neighborhoods

Stacked SGCs $\rightarrow$ Profit?

- Simply stacking SGCs gives as a neural network.
- What if we only use $K = 1$? We get a linear function wrt to $\mathcal{L}$ and on the graph laplacian spectrum. Is that any good? :S
- Yes! Stacking simple functions still lets us explore a rich class of convolutional filters and comes with some extra benefits:
  - Such filters are also not limited by the explicit form and parameterization of the approximation,
  - less likely to overfit on local neighborhoods
  - but still able to convolve a kth order neighborhood through k layers,

Stacked SGCs $\rightarrow$ Profit?

- Simply stacking SGCs gives as a neural network.
- What if we only use $K = 1$? We get a linear function wrt to $\mathcal{L}$ and on the graph laplacian spectrum. Is that any good? :S
- Yes! Stacking simple functions still lets us explore a rich class of convolutional filters and comes with some extra benefits:
  - Such filters are also not limited by the explicit form and parameterization of the approximation,
  - less likely to overfit on local neighborhoods
  - but still able to convolve a kth order neighborhood through k layers,
  - and less expensive, so we can STACK MORE LAYERS! to increase the model's capacity

Stacked SGCs $\rightarrow$ Profit?

Furthermore, because we are *engineers* we can assume $\lambda_{max} \approx 2$ and hope the gods of neural networks help the parameters adapt this change in scale during training.

Stacked SGCs $\rightarrow$ Profit?

Furthermore, because we are *engineers* we can assume $\lambda_{max} \approx 2$ and hope the gods of neural networks help the parameters adapt this change in scale during training.

◦ The simplified version is:

$$g_{\theta'} \star x \approx \theta'_0 x + \theta'_1 \left(\mathcal{L} - I_N\right) x = \theta'_0 x - \theta'_1 \mathcal{D}^{-\frac{1}{2}} \mathcal{A} \mathcal{D}^{-\frac{1}{2}} x$$

Stacked SGCs $\rightarrow$ Profit?

Furthermore, because we are *engineers* we can assume $\lambda_{max} \approx 2$ and hope the gods of neural networks help the parameters adapt this change in scale during training.

◦ The simplified version is:

$$g_{\theta'} \star x \approx \theta'_0 x + \theta'_1 \left(\mathcal{L} - I_N\right) x = \theta'_0 x - \theta'_1 \mathcal{D}^{-\frac{1}{2}} \mathcal{A} \mathcal{D}^{-\frac{1}{2}} x$$

◦ Constraining the numbers of parameters even more address overfitting and computational cost:

$$g_{\theta} \star x \approx \theta \left(I_N + \mathcal{D}^{-\frac{1}{2}} \mathcal{A} \mathcal{D}^{-\frac{1}{2}}\right) x, \text{ where } \theta = \theta'_0 = -\theta'_1$$

Stacked SGCs $\rightarrow$ Profit?

Furthermore, because we are *engineers* we can assume $\lambda_{max} \approx 2$ and hope the gods of neural networks help the parameters adapt this change in scale during training.

◦ The simplified version is:

$$g_{\theta'} \star x \approx \theta'_0 x + \theta'_1 \left(\mathcal{L} - I_N\right) x = \theta'_0 x - \theta'_1 \mathcal{D}^{-\frac{1}{2}} \mathcal{A} \mathcal{D}^{-\frac{1}{2}} x$$

◦ Constraining the numbers of parameters even more address overfitting and computational cost:

$$g_\theta \star x \approx \theta \left(I_N + \mathcal{D}^{-\frac{1}{2}} \mathcal{A} \mathcal{D}^{-\frac{1}{2}} \right) x, \text{ where } \theta = \theta'_0 = -\theta'_1$$

◦ Eigenvalues of $I_N + \mathcal{D}^{-\frac{1}{2}} \mathcal{A} \mathcal{D}^{-\frac{1}{2}} \in [0, 2] \rightarrow$ exploding/vanishing gradients. Solution: apply the renormalization trick again:

$$g_\theta \star x \approx \theta \left(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} \right) x$$

Stacked SGCs $\rightarrow$ Profit?

General form for signal $X \in \mathbb{R}^{N \times C}$ (C channels) and F filters with $\Theta \in \mathbb{R}^{C \times F}$ filter parameters:

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

$Z \in \mathbb{R}^{N \times F}$ being convolved signal matrix.

The complexity of the operation is $\mathcal{O}(|\mathcal{E}|FC)$, as $\tilde{A}X$ can be efficiently implemented as a product of a sparse matrix with a dense matrix.

An example

2-layer GCN:

- Preprocess: $\hat{\mathcal{A}} = \tilde{\mathcal{D}}^{-\frac{1}{2}} \tilde{\mathcal{A}} \tilde{\mathcal{D}}^{-\frac{1}{2}}$
- Neural Network:

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

- Cross-Entropy loss:

$$\mathcal{J} = -\sum_{\ell \in \mathcal{Y}_L} \sum_{f=1}^{F} Y_{\ell f} \ln Z_{\ell f},$$

  where $\mathcal{Y}_L$ is the set of node indices that have labels.

- Adam on a full dataset batch + early stopping
- Dropout for all layers and L_2 regularization for the first one.
- Glorot weight initialization (aka Xavier normal)

Experiments

Datasets

Table 1: Dataset statistics, as reported in Yang et al. (2016).

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

Classification

Table 2: Summary of results in terms of classification accuracy (in percent).

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)
GCN (rand. splits)	67.9 ± 0.5	80.1 ± 0.5	78.9 ± 0.7	58.4 ± 1.7

Experiments

Propagation models evaluation on classification accuracy using random weight initialization

Table 3: Comparison of propagation models.

Description		Propagation model	Citeseer	Cora	Pubmed
Chebyshev filter (Eq. 5)	$K = 3$	$\sum_{k=0}^{K} T_k(\tilde{L}) X \Theta_k$	69.8	79.5	74.4
	$K = 2$		69.6	81.2	73.8
1st-order model (Eq. 6)		$X\Theta_0 + D^{-\frac{1}{2}} A D^{-\frac{1}{2}} X \Theta_1$	68.3	80.0	77.5
Single parameter (Eq. 7)		$(I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}}) X \Theta$	69.3	79.2	77.4
Renormalization trick (Eq. 8)		$\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} X \Theta$	**70.3**	**81.5**	**79.0**
1st-order term only		$D^{-\frac{1}{2}} A D^{-\frac{1}{2}} X \Theta$	68.7	80.5	77.8
Multi-layer perceptron		$X\Theta$	46.5	55.1	71.4

Node Embeddings

$$Z = \tanh\left(\hat{A}\tanh\left(\hat{A}\tanh\left(\hat{A}XW^{(0)}\right)W^{(1)}\right)W^{(2)}\right)$$

(a) Karate club network

0.00
−0.05
−0.10
−0.15
−0.15 −0.10 −0.05 0.00 0.05

(b) Random weight embedding

Figure 3: *Left*: Zachary's karate club network (Zachary, 1977), colors denote communities obtained via modularity-based clustering (Brandes et al., 2008). *Right*: Embeddings obtained from an untrained 3-layer GCN model (Eq. 13) with random weights applied to the karate club network. Best viewed on a computer screen.

Experiments

Node Embeddings for classification

Adding a softmax layer to the previous model:

Some code

Graph Convolution Layer (1/2)

```
class GraphConvolution(Module):
    def __init__(self, in_features, out_features, bias=True, init_method='xavier'):
        super(GraphConvolution, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = Parameter(torch.FloatTensor(in_features, out_features))
        if bias:
            self.bias = Parameter(torch.FloatTensor(out_features))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters(method=init_method)

    def forward(self, input, adj):
        support = torch.mm(input, self.weight)
        output = torch.spmm(adj, support)
        if self.bias is not None:
            return output + self.bias
        else:
            return output
```

Some code

Graph Convolution Layer (2/2)

```
def reset_parameters(self, method='xavier'):
    if method == 'uniform':
        stdv = 1. / math.sqrt(self.weight.size(1))
        self.weight.data.uniform_(-stdv, stdv)
        if self.bias is not None:
            self.bias.data.uniform_(-stdv, stdv)
    elif method == 'kaiming':
        nn.init.kaiming_normal_(self.weight.data, a=0, mode='fan_in')
        if self.bias is not None:
            nn.init.constant_(self.bias.data, 0.0)
    elif method == 'xavier':
        nn.init.xavier_normal_(self.weight.data, gain=0.02)
        if self.bias is not None:
            nn.init.constant_(self.bias.data, 0.0)
    else:
        raise NotImplementedError
```

Some code

Graph Convolution Network

```
class GCN(nn.Module):
    def __init__(self, nfeat, nhid, nclass, dropout, init_method='xavier', dropout_input=False):
        super(GCN, self).__init__()
        self.gc1 = GraphConvolution(nfeat, nhid, init_method=init_method)
        self.gc2 = GraphConvolution(nhid, nclass, init_method=init_method)
        self.dropout = dropout
        self.dropout_input = dropout_input

    def forward(self, x, adj):
        if self.dropout_input:
            x = F.dropout(x, self.dropout, training=self.training)
        x = F.relu(self.gc1(x, adj))
        x = F.dropout(x, self.dropout, training=self.training)
        x = self.gc2(x, adj)
        return F.log_softmax(x, dim=1)
```

Some code

Training (1/2)

```
def lr_scheduler(epoch, opt):
    return opt.lr * (0.5 ** (epoch / opt.lr_decay_epoch))


# Train
def train(epoch):
    global best_acc

    t = time.time()
    model.train()
    optimizer.lr = lr_scheduler(epoch, opt)
    optimizer.zero_grad()

    output = model(features, adj)
    loss_train = F.nll_loss(output[idx_train], labels[idx_train])
    acc_train = accuracy(output[idx_train], labels[idx_train])

    loss_train.backward()
    optimizer.step()
```

Some code

Training (2/2)

```python
# Validation for each epoch
model.eval()
output = model(features, adj)
loss_val = F.nll_loss(output[idx_val], labels[idx_val])
acc_val = accuracy(output[idx_val], labels[idx_val])

if acc_val > best_acc:
    best_acc = acc_val
    state = {
        'model': model,
        'acc': best_acc,
        'epoch': epoch,
    }
    torch.save(state, os.path.join(save_point, '%s.t7' % (opt.model)))
```

References

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.

Thomas N Kipf and Max Welling.
Semi-supervised classification with graph convolutional networks.
arXiv preprint arXiv:1609.02907, 2016.

The End