Thanks for joining me for a presentation on ...

Graph Convolutional Networks

Your Presenter:

Christian McDaniel

Data Scientist & Software Engineer,

Graph Convolutional Networks

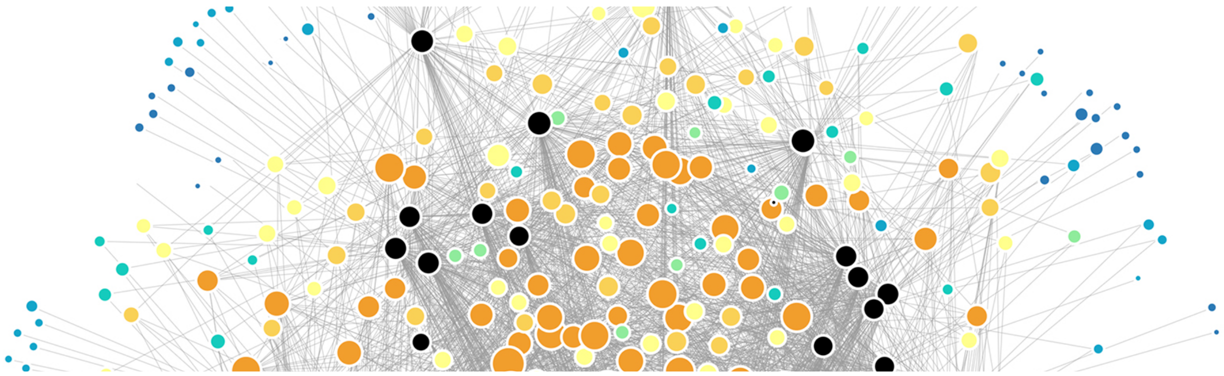
Background

- Graph Convolutional Networks are both simple and complex
- They borrow from multiple domains to arrive at an elegant analysis algorithm
    - Deep Learning - Convolutional Neural Networks
    - Spectral Graph Clustering - Graph Laplacian
    - Signal Processing - Fourier Transform

Graph Convolutional Networks

Background

- Imagine a dataset:
    - Made up many **points**
    - Each point can be described by p **features** and falls into one of c **classes**.
    - These points are **interconnected**.



Graph Convolutional Networks

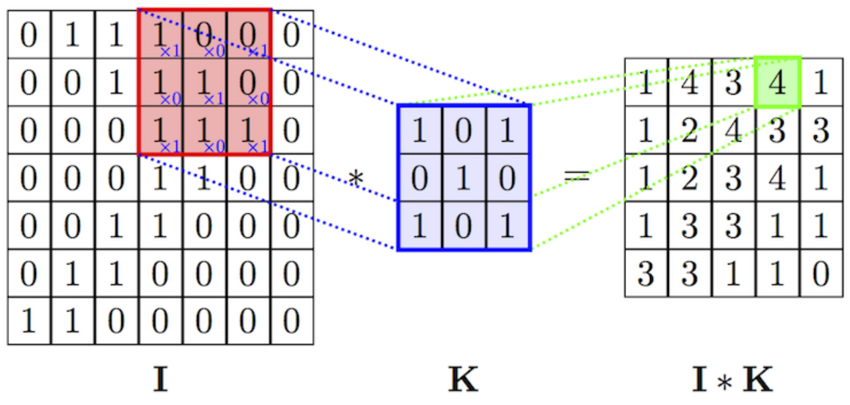
Graph Convolutional Networks

Background - Convolutional Neural Networks

+ Great success with computer vision-based applications

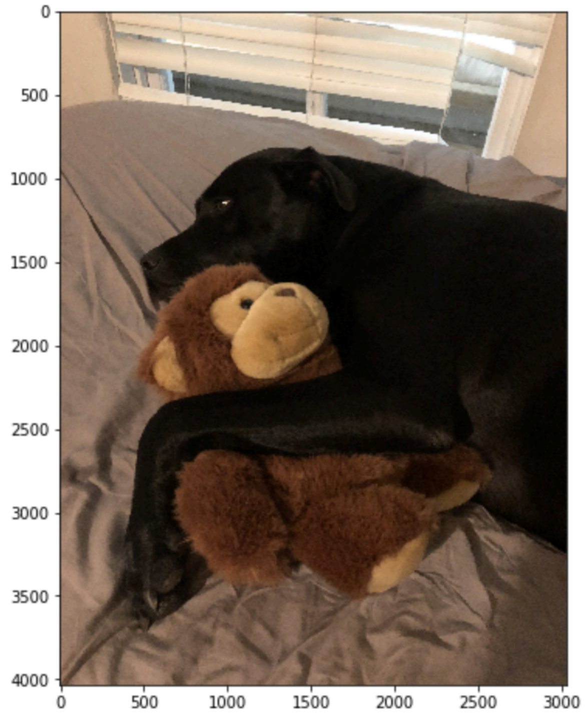
+ Some advantages of CNN's:
   + Computational efficiency (~ O(V + E))
   + Fixed number of parameters (independent of input size)
   + Localisation: acts on a local neighborhood
   + Learns the importance of different neighbors

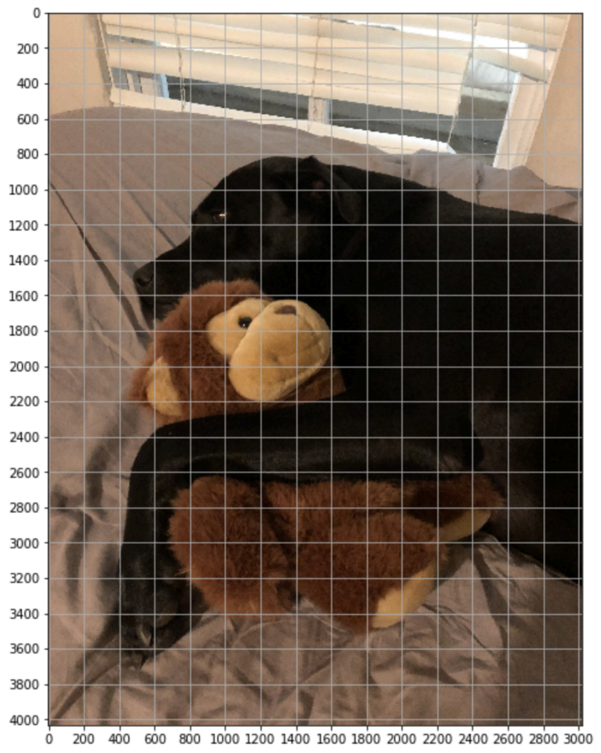
+ Images have highly regular connectivity pattern
   + each pixel is "connected" to its eight neighboring pixels
   + Convolving a kernel matrix across the "nodes" is trivial



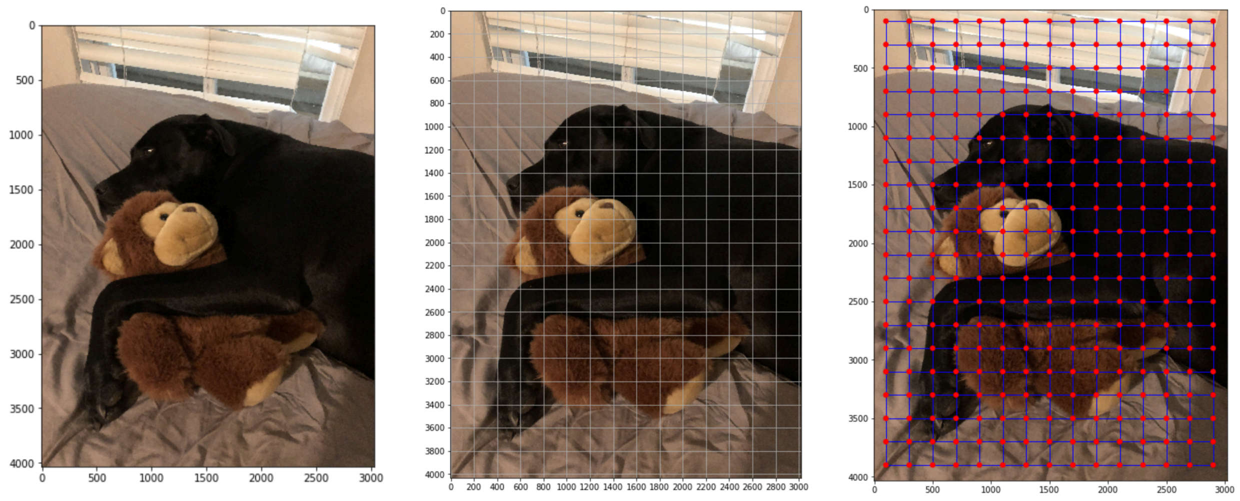
Images as Well-Behaved Graphs

Images as Well-Behaved Graphs





Images as Well-Behaved Graphs



Graph Convolutional Networks

- Generalizing the convolution operation for arbitrary graph structures is much more tricky
- We will use some *very* convenient (and awesome) rules from Signal Processing and Spectral Graph Theory
- Next we'll discuss recent advances improving performance and computational efficiency
    - (Semi-Supervised Classification with Graph Convolutional Networks, Kipf & Welling 2016)

Graph Convolutional Networks

Before we learn the ituitions, let's first look at what a simple GCN may look like:

The graph-based convolution:

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

Add in a Nonlinear Activation

- Graph structure is encoded directly into the neural network model by incorporating the adjacency matrix: $f(X, A)$
- We can do so by wrapping the above equation in a nonlinear activation function:

$$H^{(l+1)} = \sigma(\tilde{D}^{\frac{-1}{2}} \tilde{A} \tilde{D}^{\frac{-1}{2}} H^l W^l)$$

- Where W^l is a layer-specific trainable weight matrix, $\sigma(\cdot)$ is an activation function, $H^l \in \mathbb{R}^{NxF^l}$ is the matrix of activations from the l^{th} layer, and $H^0 = X$
- A two-layer Graph Convolutional Network may look something like

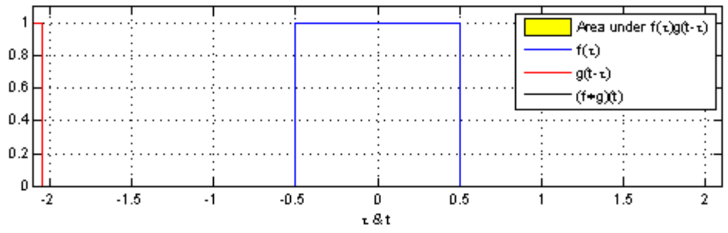
$$Z = f(X, A) = \text{softmax}(\hat{A}\text{ReLU}(\hat{A}XW^0)W^1)$$

- Where $\hat{A}$ is the precalculated $D^{\frac{-1}{2}} A D^{\frac{-1}{2}}$

So... Where did this implimentation come from??

1. Generalizing the Convolution

- A **convolution operation** combines one function f with another function g such that the first function is transformed by the other

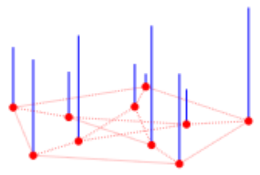


- In Convolutional Neural Networks, the second function g is *learned* for a given data set so that the transformations on f are meaningful w.r.t. some class values
  - e.g., a set of pixels showing a dog, f_d may be transformed to values near 0
  - while a set of pixels showing a cat, f_c may be transformed to values near 1
- Let's see how we might do this for our graph-based data...

Graph Convolutional Networks

Signal Processing

- With the **nodes** of the graph representing individual examples from the dataset,
- And some data at each node
- E.g.,
    - scalar intensity values at each pixel of an image
    - feature vectors for higher dimensional data
- We can consider the data values as **signals** on each node



Graph Convolutional Networks

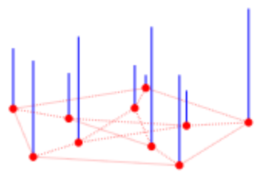
Signal Processing

- With the **nodes** of the graph representing individual examples from the dataset,
- And some data at each node

Graph Convolutional Networks

Signal Processing

- With the **nodes** of the graph representing individual examples from the dataset,
- And some data at each node
    - scalar intensity values at each pixel of an image
    - feature vectors for higher dimensional data
- We can consider the data values as **signals** on each node

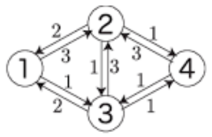


- The changing of the signals across the edges of the graph resemble the fluctuation of a signal over time
- Borrowing from signal processing, these oscillations could be characterized by their component frequencies, via the **Fourier transform**
- It just so happens that graphs have their own version of the FT...

Graph Convolutional Networks

From Spectral Graph Theory ... The Normalized Graph Laplacian

$$L = I - D^{\frac{-1}{2}} A D^{\frac{-1}{2}}$$



$$\mathcal{A} = \begin{bmatrix} 0 & 3 & 1 & 0 \\ 2 & 0 & 1 & 1 \\ 2 & 3 & 0 & 1 \\ 0 & 3 & 1 & 0 \end{bmatrix}$$

+ Where I is the diagonal **identity matrix**
D is the diagonal **degree matrix**
and A is the weighted **adjacency matrix**

$$\mathcal{D} = \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix} \quad \mathcal{L} = \begin{bmatrix} 4 & -3 & -1 & 0 \\ -2 & 4 & -1 & -1 \\ -2 & -3 & 6 & -1 \\ 0 & -3 & -1 & 4 \end{bmatrix}$$

Graph Convolutional Networks

The Normalized Graph Laplacian in the Spectral Domain

- The Normalized Graph Laplacian L is a real symmetric positive semidefinite matrix
  - $z^T Lz \geq 0$ for all columns z
  - it has a complete set of orthonormal eigenvectors $\{u_l\}_{l=0}^{n-1} \in \mathbb{R}^n$ a.k.a. the **graph Fourier modes**
  - and it has the associated ordered real nonnegative eigenvalues $\{\lambda_l\}_{l=0}^{n-1}$, the frequencies of the graph
- L is diagonalized in the Fourier basis $U = [u_0, u_1, \ldots, un - 1] \in \mathbb{R}^{nxn}$ such that

$$L = U^T \Lambda U$$

- Where $\Lambda = \mathrm{diag}([\lambda_0, \lambda_1, \ldots, \lambda_{n-1}] \in \mathbb{R}^{nxn})$

I.e.,

$$L = I - D^{-1/2} A D^{-1/2} = U^T \Lambda U$$

Graph Convolutional Networks

The Normalized Graph Laplacian in the Spectral Domain

$$L = I - D^{-1/2} A D^{-1/2} = U^T \Lambda U$$

The Graph Fourier Transform

- The **Fourier transform** in the graph domain, where eigenvectors denote Fourier modes and eigenvalues denote frequencies of the graph is defined

$$\hat{x} = U^T x \in \mathbb{R}^n$$

- As on Euclidean spaces, this transform enables the formulation of fundamental operations.
- E.g., the **convolution operation** becomes multiplication, and we can define a **convolution** of a data vector x with a filter g_Θ as

$$g_\Theta * x = U((U^T g_\Theta) \bigodot (U^T x)) = g_\Theta(U \Lambda U^T)x$$

Graph Convolutional Networks

The Normalized Graph Laplacian in the Spectral Domain

$$L = I - D^{-1/2} A D^{-1/2} = U^T \Lambda U$$

The Graph Fourier Transform

$$g_\Theta * x = U((U^T g_\Theta) \bigodot (U^T x)) = g_\Theta(U \Lambda U^T)x$$

Graph Convolutional Networks

The Normalized Graph Laplacian in the Spectral Domain

$$L = I - D^{-1/2} A D^{-1/2} = U^T \Lambda U$$

The Graph Fourier Transform

$$g_\Theta * x = U((U^T g_\Theta) \bigodot (U^T x)) = g_\Theta(U \Lambda U^T)x$$

Signal Processing

$$g_\Theta * x = g_\Theta(U \Lambda U^T)x = U \hat{G} U^T x$$

- As computing the eigenspectrum of a matrix can be computationally exepensive, we can approximate the Fourier coefficients using a Kth order Chebychev Polynomial

$$g_{\Theta\prime} \approx \sum_{k=0}^{K} \Theta\prime_k T_k(\tilde{\Lambda})$$

- Where rescaled $\tilde{\Lambda} = \frac{2}{\lambda_{max}} \Lambda - I$

Graph Convolutional Networks

The Normalized Graph Laplacian in the Spectral Domain

$$L = I - D^{-1/2} A D^{-1/2} = U^T \Lambda U$$

The Graph Fourier Transform

$$g_\Theta * x = U((U^T g_\Theta) \bigodot (U^T x)) = g_\Theta(U \Lambda U^T)x$$

Signal Processing

$$g_\Theta * x = g_\Theta(U \Lambda U^T)x = U\hat{G}U^T x$$

$$g'_\Theta \approx \sum_{k=0}^{K} \Theta'_k T_k(\tilde{\Lambda})$$

- This lets us define

$$g_\Theta * x \approx \sum_{k=0}^{K} \Theta'_k T_k(\tilde{L})x$$

- with $\tilde{L} = \frac{2}{\lambda_{max}} L - I$
- The convolution is now K-localized, operating only on the nodes a distance of K away from any given node

Graph Convolutional Networks

The Normalized Graph Laplacian in the Spectral Domain

$$L = I - D^{-1/2} A D^{-1/2} = U^T \Lambda U$$

The Graph Fourier Transform and Signal Processing

$$g_\Theta * x = U((U^T g_\Theta) \bigodot (U^T x)) = g_\Theta(U \Lambda U^T) x = U \hat{G} U^T x$$

$$g_\Theta * x \approx \sum_{k=0}^{K} \Theta'_k T_k(\tilde{L}) x$$

Graph Convolutional Networks

The Normalized Graph Laplacian in the Spectral Domain

$$L = I - D^{-1/2} A D^{-1/2} = U^T \Lambda U$$

The Graph Fourier Transform and Signal Processing

$$g_\Theta * x = U((U^T g_\Theta) \bigodot (U^T x)) = g_\Theta(U \Lambda U^T)x = U \hat{G} U^T x$$

$$g_\Theta * x \approx \sum_{k=0}^{K} \Theta'_k T_k(\tilde{L})x$$

Graph Convolutional Networks

The Normalized Graph Laplacian in the Spectral Domain

$$L = I - D^{-1/2} A D^{-1/2} = U^T \Lambda U$$

The Graph Fourier Transform and Signal Processing

$$g_\Theta * x = U((U^T g_\Theta) \bigodot (U^T x)) = g_\Theta(U \Lambda U^T)x = U \hat{G} U^T x$$

$$g_\Theta * x \approx \sum_{k=0}^{K} \Theta'_k T_k(\tilde{L})x$$

Subsequent Computational Advancements

1) $K = 1$

2) When calculating the rescaled $\tilde{\Lambda}$ and $\tilde{L}$, we can approximate $\lambda_{max} \approx 2$, expecting the neural network paramaters to adapt accordingly during training.

$$\tilde{L} = \frac{2}{\lambda_{max}} L - I, \text{where } L = I - D^{-1/2} A D^{-1/2} \rightarrow \tilde{L} \approx L - I = D^{\frac{-1}{2}} A D^{\frac{-1}{2}}$$

$$g_\Theta * x \approx \theta'_0 x + \theta'_1 (L - I)x = \theta'_0 x + \theta'_1 D^{\frac{-1}{2}} A D^{\frac{-1}{2}} x$$

- with two free parameters λ'_0 and λ'_1

Graph Convolutional Networks

The Normalized Graph Laplacian in the Spectral Domain

$$L = I - D^{-1/2} A D^{-1/2} = U^T \Lambda U$$

The Graph Fourier Transform and Signal Processing

$$g_\Theta * x = U((U^T g_\Theta) \bigodot (U^T x)) = g_\Theta(U \Lambda U^T)x = U\hat{G}U^T x$$

$$g_\Theta * x \approx \sum_{k=0}^{K} \Theta'_k T_k(\tilde{L})x$$

Subsequent Computational Advancements

1) $K = 1$

2) $g_\Theta * x \approx \theta'_0 x + \theta'_1 D^{\frac{-1}{2}} A D^{\frac{-1}{2}} x$

3) We can further constrain the problem to learning a *single* parameter (per dimension in x): $\theta = \theta'_0 = -\theta'_1$

- This further prevents overfittings and reduces computations

Subsequent Computational Advancements

4) The previous revisions have left L with eigenvalues in $[0, 2]$, potentially leading to numerical instabilities and exploding/vanishing gradients due to repeated operations

- A "renormalization trick" has been devised:

$$D^{\frac{-1}{2}} A D^{\frac{-1}{2}} \rightarrow \tilde{D}^{\frac{-1}{2}} \tilde{A} \tilde{D}^{\frac{-1}{2}}$$

- With $\tilde{A} = A + I$ and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ (i.e., self-loops have been added)
- We generalize this to a signal $X \in \mathbb{R}^{NxP}$ with P-dimensional vectors at each node and F filters as

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

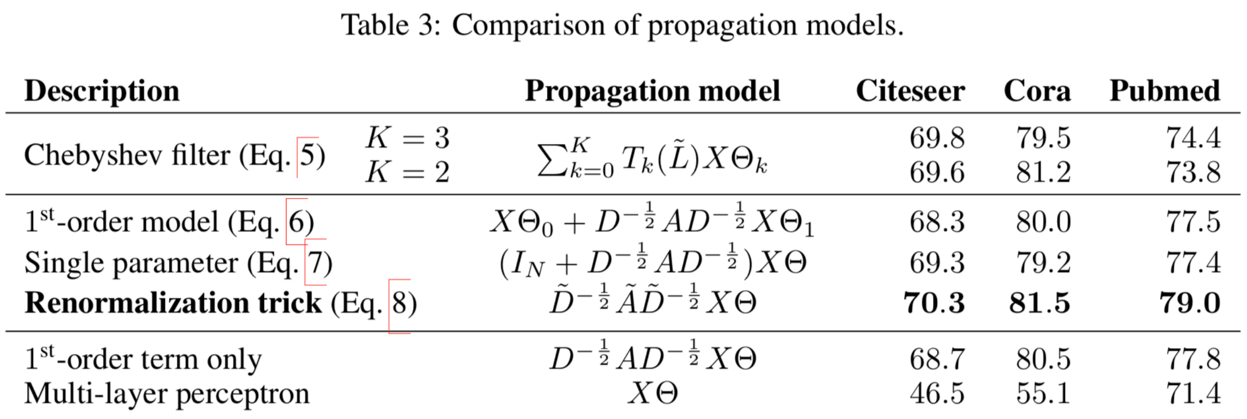


Table 3: Comparison of propagation models.

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

Graph Convolutional Networks

The graph-based convolution:

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

Nonlinear Activation

- Graph structure is encoded directly into the neural network model by incorporating the adjacency matrix: $f(X, A)$
- We can do so by wrapping the above equation in a nonlinear activation function:

$$H^{(l+1)} = \sigma(\tilde{D}^{\frac{-1}{2}} \tilde{A} \tilde{D}^{\frac{-1}{2}} H^l W^l)$$

- Where W^l is a layer-specific trainable weight matrix, $\sigma(\cdot)$ is an activation function, $H^l \in \mathbb{R}^{NxF^l}$ is the matrix of activations from the l^{th} layer, and $H^0 = X$
- A two-layer Graph Convolutional Network may look something like

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

- Where $\hat{A}$ is the precalculated $D^{\frac{-1}{2}} A D^{\frac{-1}{2}}$

In [13]:

```
""" Github: tkipf/pygcn """
import math
import torch
from torch.nn.parameter import Parameter
from torch.nn.modules.module import Module

class GraphConvolution(Module):
    """
    Simple GCN layer, similar to https://arxiv.org/abs/1609.02907
    """
    def __init__(self, in_features, out_features, bias=True):
        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()

    def reset_parameters(self):
        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)

    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
```

In [ ]:
```
""" Github: tkipf/pygcn """
import torch.nn as nn
import torch.nn.functional as F
from layers import GraphConvolution


class GCN(nn.Module):
    def __init__(self, nfeat, nhid, nclass, dropout):
        super(GCN, self).__init__()

        self.gc1 = GraphConvolution(nfeat, nhid)
        self.gc2 = GraphConvolution(nhid, nclass)
        self.dropout = dropout

    def forward(self, x, adj):
        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)
```

Graph Convolutional Networks

Time for Some Examples!

Graph Convolutional Networks

Example

- The Cora Dataset
    - 2708 Machine Learning papers --> the Nodes
    - 7 classes:
        - Case_Based
        - Genetic_Algorithms
        - Neural_Networks
        - Probabilistic_Methods
        - Reinforcement_Learning
        - Rule_Learning
        - Theory
    - Each paper is cited by at least one other paper --> the Edges
    - Each paper is described by the frequency of 1433 unique and meaningful words --> the Features

In [18]:
```
import pandas as pd

edges    = pd.read_csv('/Users/cm185255/Documents/pygcn/data/cora/cora.cites',se
p='\t',header=None,names=["cited paper ID","ID of paper cited"])
features = pd.read_csv('/Users/cm185255/Documents/pygcn/data/cora/cora.content',
sep='\t',header=None)

print('Table for the Edges (citations between papers)')
print(edges.head())
print('\nTable for the Features (word counts for each paper; last column = class
)')
print(features.head())

...
```

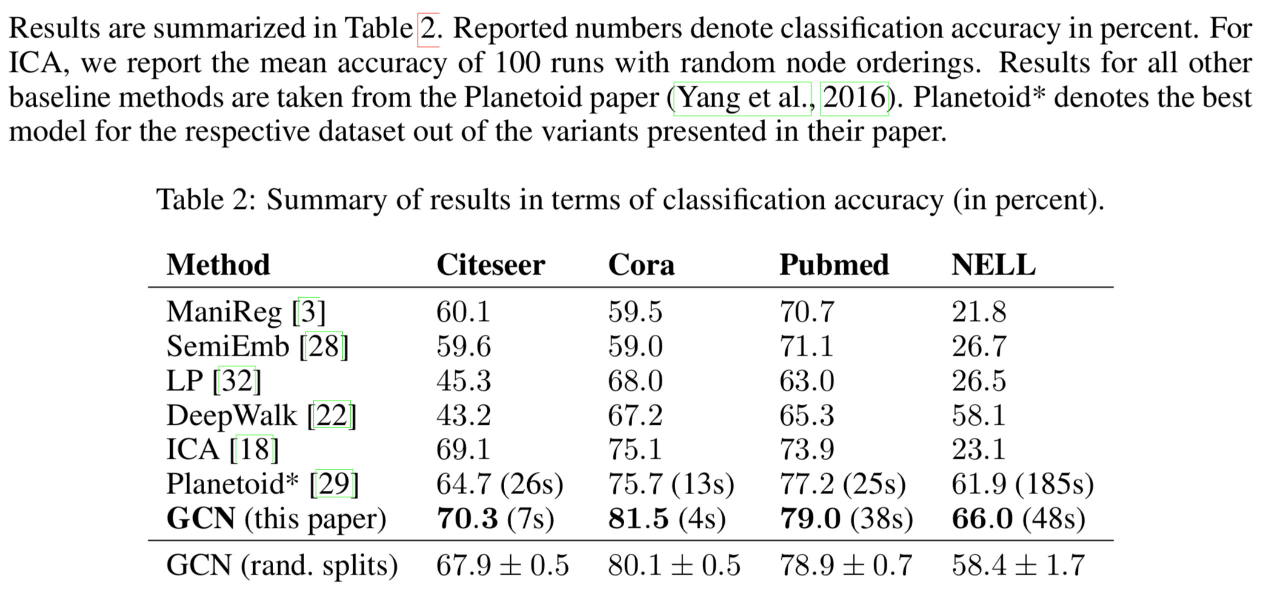

Results are summarized in Table 2. Reported numbers denote classification accuracy in percent. For ICA, we report the mean accuracy of 100 runs with random node orderings. Results for all other baseline methods are taken from the Planetoid paper (Yang et al., 2016). Planetoid* denotes the best model for the respective dataset out of the variants presented in their paper.

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

- ManiReg = Manifold regularization (Belkin et al 2006)
- SemiEmb = semi-supervised embedding (Watson et al, 2012)
- LP = label propagation (Zhu et al, 2003)
- DeepWalk = skip-gram graph embeddings (Perozzi et al, 2014)
- ICA = iterative classification algorithm (Lu & Getoor, 2003)
- Planetois (Yang et al, 2016)

Graph Convolutional Networks

Example #2 - Adding Attention coefficients for whole-graph classification

- Graph Attention Networks (2018)
    - Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, Yoshua Bengio
- Utilizes self-attention to compute a concise representation of a signal sequence

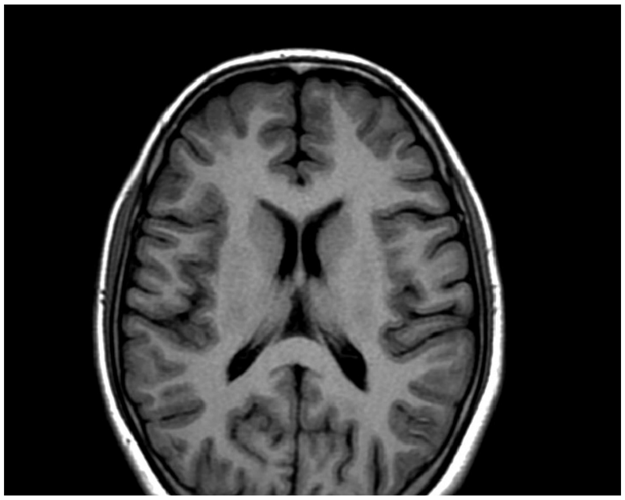
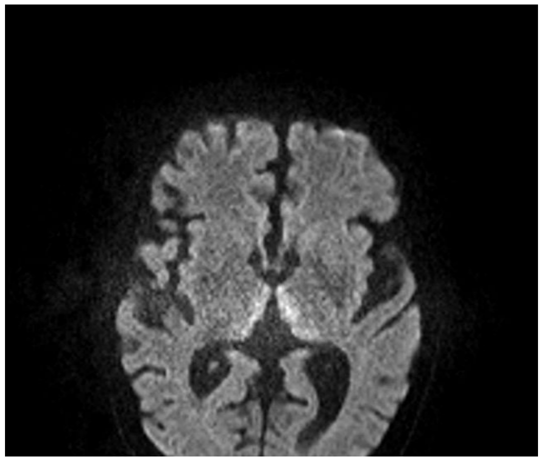
Graph Convolutional Networks

Graph Convolutional Networks

Example #2:

Developing a Graph Convolution-Based Analysis Pipeline for Multi-Modal Neuroimage Data: An Application to Parkinson's Disease

Christian McDaniel & Shannon Quinn

Anatomical MRI	Diffusion MRI
	

Hypothesis:

- Parkinson's disease results in structural and functional brain changes related to decreased dopamine production
- Neuroimaging has shown promise (though limited) for PD detection and research
- Combining insights from multiple modalities may help reveal new discoveries and improve detection

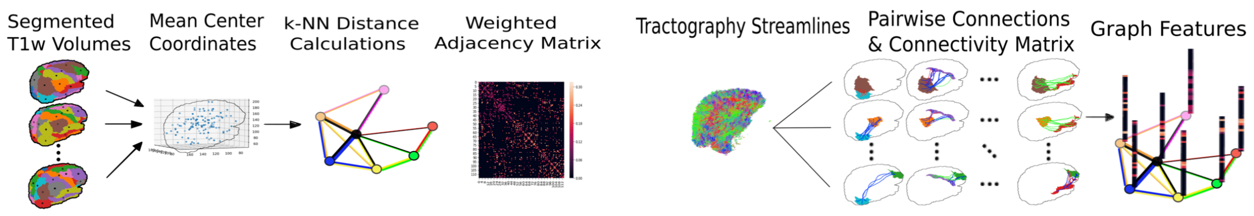
The data

- Anatomical MRI helps identify regions of the brain; doesn't say much about PD pathology
- There are multiple **tractography** algorithms; each offers unique information about structural connectivity
    - Each tractography algorithm generates a new set of features for the same graph

The GCN Implimentation

- The algorithm will need to consolidate data from multiple tractography algorithms
- The outputs from the GCN on each node will need to be consolidated to a single output (PD vs HC) for each graph

Graph Convolutional Networks


Segmented T1w Volumes
Mean Center Coordinates
k-NN Distance Calculations
Weighted Adjacency Matrix
Tractography Streamlines
Pairwise Connections & Connectivity Matrix
Graph Features

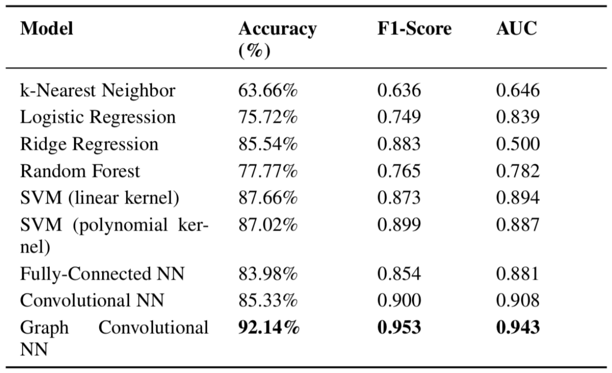
Graph Convolutional Networks

Example #2

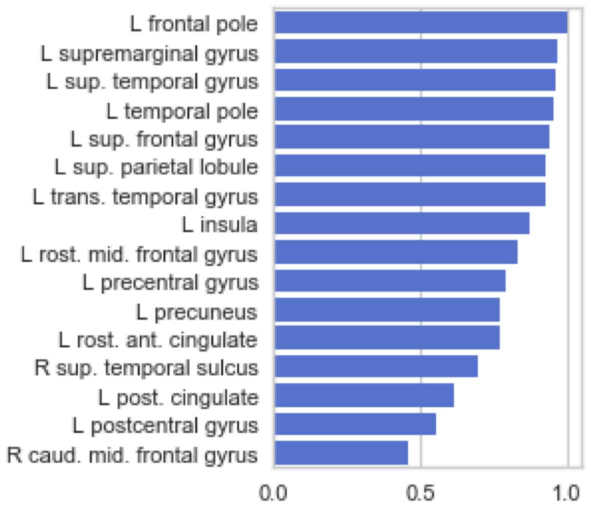
Graph Convolutional Networks

Example #2

Results	Attentions



Model	Accuracy (%)	F1-Score	AUC
k-Nearest Neighbor	63.66%	0.636	0.646
Logistic Regression	75.72%	0.749	0.839
Ridge Regression	85.54%	0.883	0.500
Random Forest	77.77%	0.765	0.782
SVM (linear kernel)	87.66%	0.873	0.894
SVM (polynomial kernel)	87.02%	0.899	0.887
Fully-Connected NN	83.98%	0.854	0.881
Convolutional NN	85.33%	0.900	0.908
Graph Convolutional NN	**92.14%**	**0.953**	**0.943**



Fin!

Questions?