Deep Learning
Week [06]: [GNNs]

Logistics
- HW2 is due today
  ○ One extra slip day because of huggingface maintenance
  ○ Run the cells in the coding assignment
  ○ Turn off GPUs after using
- Project Proposal is due March 7
- HW 3 will be released at the end of the week
- We won’t release HW solutions
- Change in office hours
  ○ Zach’s office hours will be 4-5 on Wednesday’s (instead of 6-7pm)
  ○ Varsha’s office hours will be 1-2pm Tuesday’s (instead of 9-10am)

What are Graphs?
Graphs are a general language for describing and analyzing entities with relations/interactions

Graphs = Nodes + Edges
Graph: Directed vs Undirected

How the edges link the nodes allows us to distinguish between undirected graphs vs directed graphs.

**Graph G with 3 nodes**

Directed

Examples:
- Phone Calls
- Following on Twitter

Examples:
- Academic collaborations
- Friendships on Facebook

**Adjacency Matrix - \( A \)**

\( A \) represents the edges in a given graph.

\( A_{ij} = 1 \) if an edge exists between nodes \( i \) and \( j \), else 0.

\[
A = \begin{bmatrix}
1 & 1 \\
1 & 0
\end{bmatrix}
\]

**Degree Matrix - \( D \)**

\( D \) is a diagonal matrix, where each diagonal entry represents the degree of each node in a given graph.

\( D_{ii} = \text{degree}(i) \)

\[
A = \begin{bmatrix}
0 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 0
\end{bmatrix}
\]

\[
D = \begin{bmatrix}
3 \\
1 \\
1 \\
1 \\
1 \\
1
\end{bmatrix}
\]

GraphML vs NLP vs CV

The cat sat on the mat.

No spatial locality (unlike grids)

No rank ordering or fixed reference point
Why Do We Care About Learning on Graphs?

There are many different settings where we might care about learning on graphs:

- Graph classification
- Node classification
- Link prediction
- Community detection
- Graph embedding
- Graph generation

Representation Learning > Feature Engineering

Input Graph → Structured Features → Learning Algorithm → Prediction

Feature engineering
(node-level, edge-level, graph-level features)

Downstream prediction task

Representation Learning + Classifier

Map each node to a low-dimensional vector!
Example Graph mapped into 2 dimensions

(a) Input: Karate Graph  (b) Output: Representation

Classify Embeddings!

DeepWalk: word2vec For Graphs

This is exactly the same optimization as word2vec, but we instead optimize over sequences of random walks on a graph.

Brief Review: word2vec

(words close in sentences → close in embedding space)

Can we do this on graphs?

“During the Battle of Endor, the Death Star II’s energy shield was destroyed...”

“W. W. W. W. W.”

“In the third film, Anakin becomes Vader when...”

“Samuel L. Jackson portrayed Mace Windu in the prequel trilogy...”

Given a graph and a starting point, we select a neighbor of it at random, and move to this neighbor; then we select a neighbor of this point at random, and move to it, etc. The random sequence of points visited this way is a random walk on the graph.
DeepWalk selects the next node to traverse to in each random walk purely at random (unbiased). Nodes that are close together in the random walk sequence should be embedded closer together in the embedding space.

These are the “sentences” that we generate!

Example: node2vec: The Introduction of Bias...

node2vec = DeepWalk + control over local vs global exploration (via two additional hyperparameters that we won’t discuss in detail)

Breadth First Search (BFS) \{s1, s2, s3\} Local microscopic view

Depth First Search (DFS) \{s4, s5, s6\} Global macroscopic view

Grover and Leskovec., ACM SIGKDD, 2016

GRAPH NEURAL NETWORKS

Convolutional Layer in CNN

Translation-invariant

How about for non-Euclidean data? Can we do something similar with graphs?
Can you perform convolutions on graphs? What would that look like? Will you run into any problems?

Locality vs Homophily

Image Convolutions
- Generate next layer embedding vectors for each pixel in an input image by aggregating the transformed feature vectors of each of the pixel's neighbors.

Graph Convolutions
- Generate next layer embedding vectors for each node in an input graph by aggregating the transformed feature vectors of each of the node's neighbors.

Locality: you can tell a lot about a particular pixel based on the properties of their neighbors.

Homophily: you can tell a lot about a particular node based on the properties of their neighbors.

Let's look at a single layer of a graph convolution

TARGET NODE

h_A = \sigma \left( \sum_{u \in N(A)} x_u W \right) / |N(A)|

Transform

Aggregate

INPUT GRAPH

Note: Aggregation function MUST be permutation-invariant!
- Mean()
- Sum()
- Max()
We repeat this process of transforming and aggregating neighboring embedding vectors for every node in the graph.

Example time!

1. Let's also give these some values...

2. Normalize!
You left multiply by $D^{-1}$. Compute $D^{-1}$ and then think about why we multiply by $D^{-1}$.

(As a reminder, $D$ is the diagonal matrix where each entry is the degree of each vertex)
With random walks, what we’re optimizing are the final embedding vectors themselves, not weights… so for every new/unseen node that we’re given (e.g. in a test set), we have to use SGD again to optimize their embeddings, which is computationally expensive!

THIS IS A BIG REASON WHY WE USE A WEIGHT MATRIX!

Stacking GCN Layers

\[
H^{(1)} = \sigma(D^{-1}A H^{(0)} W_0)
\]

Input to the next layer

Note: new weight matrix! Weight matrices in GNNs are layer-specific.

\[
H^{(2)} = \sigma(D^{-1}A H^{(1)} W_1)
\]

But D and A never change!

Stacking GCN Layers

Final GCN update rules:

Node-level update rule:

\[
h^{l+1}_v = \sigma\left(\sum_{u \in N(v)} \frac{h^l_u W_l}{|N(v)|}\right)
\]

Graph-level update rule:

\[
H^{(l+1)} = \sigma(D^{-1}A H^{(l)} W_l)
\]

Let’s just keep adding more layers, right? BIG problem!

What do we do with Z?

Decoder

Depends on the downstream prediction task:

- Feed Z into a MLP + Softmax decoder for node-level classification/regression
- For graph-level predictions (e.g. classifying an entire graph), can concat/sum/mean all vectors in Z, and then feed this long vector into a MLP
  - Just like in CNNs!

Stacking GCN Layers

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Citeseer</th>
<th>Cora</th>
<th>Pubmed</th>
<th>NELL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ManiReg [3]</td>
<td>60.1</td>
<td>59.5</td>
<td>70.7</td>
<td>21.8</td>
</tr>
<tr>
<td>SemiEznb [28]</td>
<td>59.6</td>
<td>59.0</td>
<td>71.1</td>
<td>26.7</td>
</tr>
<tr>
<td>LP [12]</td>
<td>45.3</td>
<td>68.0</td>
<td>63.0</td>
<td>26.5</td>
</tr>
<tr>
<td>DeepWalk [22]</td>
<td>43.2</td>
<td>67.2</td>
<td>65.3</td>
<td>58.1</td>
</tr>
<tr>
<td>RCA [18]</td>
<td>69.1</td>
<td>75.3</td>
<td>73.9</td>
<td>24.1</td>
</tr>
<tr>
<td>Planefold [29]</td>
<td>64.7 (26s)</td>
<td>75.7 (13s)</td>
<td>77.2 (25s)</td>
<td>61.9 (185s)</td>
</tr>
<tr>
<td>GCN (this paper)</td>
<td>70.3 (7s)</td>
<td>81.5 (4s)</td>
<td>79.0 (38s)</td>
<td>66.0 (486s)</td>
</tr>
<tr>
<td>GCN (rand. split)</td>
<td>67.9 ± 0.5</td>
<td>80.1 ± 0.5</td>
<td>78.9 ± 0.7</td>
<td>58.4 ± 1.7</td>
</tr>
</tbody>
</table>

With random walks, what we’re optimizing are the final embedding vectors themselves, not weights… so for every new/unseen node that we’re given (e.g. in a test set), we have to use SGD again to optimize their embeddings, which is computationally expensive!

THIS IS A BIG REASON WHY WE USE A WEIGHT MATRIX!
In order to calculate $A$'s $h^{(2)}$ vector, we need to calculate $h^{(1)}_u$ for each $u$ in $\text{Neighbors}(A)$.

In order to calculate each node $u$'s $h^{(1)}_u$ vector, we need to calculate $h^{(0)}_{u'}$ for each $u'$ in $\text{Neighbors}(u)$.

Therefore, number of layer in graph neural networks is a very important hyperparameter!

The over-smoothing problem

Here, we encounter the over-smoothing problem, where final-layer node embeddings (in $Z$) become highly similar.

GraphSAGE

2 BIG problems with GCNs:

Problem 1: $h^{(l+1)}_v$ doesn't aggregate $h^{(l)}_v$

Solution 1: Add self-loops!

$A_{\text{new}} = A + \delta$, where $\delta$ is an identity matrix.

Problem 2: Just Mean()? How about the rest?

Solution 2: Make the aggregation function a hyperparameter!

Table 1: Prediction results for the three datasets (micro-averaged F1 scores). Results for unsupervised and fully supervised GraphSAGE are shown. Analogous trends hold for macro-averaged scores.

<table>
<thead>
<tr>
<th>Name</th>
<th>Citation</th>
<th>Reddit</th>
<th>P99</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervised F1</td>
<td>Unsup. F1</td>
<td>Sup. F1</td>
<td></td>
</tr>
<tr>
<td>Random</td>
<td>0.200</td>
<td>0.206</td>
<td>0.043</td>
</tr>
<tr>
<td>Raw features</td>
<td>0.573</td>
<td>0.575</td>
<td>0.565</td>
</tr>
<tr>
<td>DeepWalk</td>
<td>0.656</td>
<td>0.656</td>
<td>0.624</td>
</tr>
<tr>
<td>DeepWalk + features</td>
<td>0.701</td>
<td>0.701</td>
<td>0.691</td>
</tr>
<tr>
<td>GraphSAGE-GCN</td>
<td>0.742</td>
<td>0.772</td>
<td>0.908</td>
</tr>
<tr>
<td>GraphSAGE-mean</td>
<td>0.778</td>
<td>0.820</td>
<td>0.997</td>
</tr>
<tr>
<td>GraphSAGE-LSTM</td>
<td>0.788</td>
<td>0.832</td>
<td>0.907</td>
</tr>
<tr>
<td>GraphSAGE-pool</td>
<td>0.798</td>
<td>0.839</td>
<td>0.992</td>
</tr>
</tbody>
</table>

Simplifying GCNs

Graph-level update rule:

\[ H^{(i+1)} = \tilde{D}^{-1} A H^{(i)} W_i \]

Define:

\[ S = D^{-1} A \]

\[ H^1 = S H^{(0)} W_0 \]

Why does this work so well?

The strength of GNNs comes from their ability to **propagate node features**, not from non-linearities


Write an expression for \( H^{(i+1)} \)

Table 2. Test accuracy (%) averaged over 10 runs on citation networks. *We remove the outliers (accuracy < 75/85/75%) when calculating their statistics due to high variance.*

<table>
<thead>
<tr>
<th></th>
<th>Cora</th>
<th>Citeseer</th>
<th>Pubmed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Numbers from literature:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GCN</td>
<td>81.5</td>
<td>70.3</td>
<td>79.0</td>
</tr>
<tr>
<td>GAT</td>
<td>83.0 ±0.7</td>
<td>72.5 ±0.7</td>
<td>79.0 ±0.3</td>
</tr>
<tr>
<td>GLN</td>
<td>81.2 ±0.1</td>
<td>70.9 ±0.1</td>
<td>78.9 ±0.1</td>
</tr>
<tr>
<td>AGNN</td>
<td>83.1 ±0.1</td>
<td>71.7 ±0.1</td>
<td>79.9 ±0.1</td>
</tr>
<tr>
<td>LNet</td>
<td>79.5 ±1.8</td>
<td>66.2 ±1.9</td>
<td>78.3 ±0.3</td>
</tr>
<tr>
<td>AdaLNet</td>
<td>80.4 ±1.1</td>
<td>68.7 ±1.0</td>
<td>78.1 ±0.4</td>
</tr>
<tr>
<td>DeepWalk</td>
<td>70.7 ±0.6</td>
<td>51.4 ±0.5</td>
<td>76.8 ±0.6</td>
</tr>
<tr>
<td>DGI</td>
<td>82.3 ±0.6</td>
<td>71.8 ±0.7</td>
<td>78.8 ±0.6</td>
</tr>
<tr>
<td><strong>Our experiments:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GCN</td>
<td>81.4 ±0.4</td>
<td>70.9 ±0.5</td>
<td>79.0 ±0.4</td>
</tr>
<tr>
<td>GAT</td>
<td>83.3 ±0.7</td>
<td>72.6 ±0.6</td>
<td>78.5 ±0.3</td>
</tr>
<tr>
<td>FastGCN</td>
<td>79.8 ±0.3</td>
<td>68.8 ±0.6</td>
<td>77.4 ±0.3</td>
</tr>
<tr>
<td>GIN</td>
<td>77.6 ±1.1</td>
<td>66.1 ±0.9</td>
<td>77.0 ±1.2</td>
</tr>
<tr>
<td>LNet</td>
<td>80.2 ±3.0</td>
<td>67.3 ±0.5</td>
<td>78.3 ±0.6</td>
</tr>
<tr>
<td>AdaLNet</td>
<td>81.9 ±1.9</td>
<td>70.6 ±0.8</td>
<td>77.8 ±0.7</td>
</tr>
<tr>
<td>DGI</td>
<td>82.5 ±0.5</td>
<td>71.6 ±0.7</td>
<td>78.4 ±0.7</td>
</tr>
<tr>
<td>SGC</td>
<td>84.0 ±0.0</td>
<td>71.9 ±0.1</td>
<td>78.5 ±0.0</td>
</tr>
</tbody>
</table>

Summary

- Learning on graphs: Classify nodes and entire graphs, predict links or detect communities and even generate graphs and their embeddings
- Feature Engineering 😞 Representation Learning 😊
- Random Walks, DeepWalk + node2vec: word2vec on graphs, embed nearby nodes on the random walk closer together
- GCN: CNN on graphs, transform + aggregate neighbors. Homophily in GCNs similar to locality in CNNs.
- Over-smoothing problem: Can’t stack too many layers